CoRM: Compactable Remote Memory over RDMA

Konstantin Taranov
ETH Zurich
konstantin.taranov@inf.ethz.ch

Salvatore Di Girolamo
ETH Zurich
salvatore.digirolamo@inf.ethz.ch

Torsten Hoefler
ETH Zurich
htor@inf.ethz.ch

Abstract

Distributed memory systems are becoming increasingly important since they provide a system-scale abstraction where physically separated memories can be addressed as a single logical one. This abstraction enables memory disaggregation, allowing systems as in-memory databases, caching services, and ephemeral storage to be naturally deployed at large scales. While this abstraction effectively increases the memory capacity of these systems, it faces additional overheads for remote memory accesses. To narrow the difference between local and remote accesses, low latency RDMA networks are a key element for efficient memory disaggregation. However, RDMA acceleration poses new obstacles to efficient memory management and particularly to memory compaction: network controllers and CPUs can concurrently access memory, potentially leading to inconsistencies if memory management operations are not synchronized. To ensure consistency, most distributed memory systems do not provide memory compaction and are exposed to memory fragmentation. We introduce CoRM, an RDMA-accelerated shared memory system that supports memory compaction and ensures strict consistency while providing one-sided RDMA accesses. We show that CoRM sustains high read throughput during normal operations, comparable to similar systems not providing memory compaction while experiencing minimal overheads during compaction. CoRM never disrupts RDMA connections and can reduce applications' active memory up to 6x by performing memory compaction.

ACM Reference Format:
Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452817

1 Introduction

The widespread availability of fast low-cost networks with Remote Direct Memory Access (RDMA) has encouraged modern database management systems to adapt RDMA for improving query performance [7, 38]. RDMA already empowers replication [23, 42, 45], index structures [48], distributed transactions [11, 43, 44], and processing of analytical workloads [5, 26]. The emergence of RDMA has further sparked interest in distributed shared memory (DSM) systems that combine memory of interconnected nodes as a shared remotely-accessible memory space [1, 2, 9, 14, 31]. In-memory databases [8], caching services [17], and ephemeral storage [41] are only some examples of systems enabled by this paradigm. However, while DSM systems provide a global memory view and resource disaggregation, they need to efficiently handle management tasks such as memory allocation and memory defragmentation.

RDMA-capable Network Interface Cards (RNICs) empower systems to access the main memory of remote peers without involving the host CPUs, providing up to 22x shorter latency [35], and up to 20x higher throughput [30], compared to traditional TCP/IP networking. However, the use of RDMA can prevent memory optimization strategies, such as memory compaction. In fact, remote objects are accessed by specifying their virtual addresses at the remote host: if the remote host relocates an object, its virtual address might change, requiring to propagate this update to the other nodes. To avoid this issue, some RDMA systems do not expose the virtual addresses of the stored objects, distributing instead objects' handles to the clients [32, 39]. While this indirection hides the process of updating pointers of relocated objects, it can significantly hinder performance because of the pointer chasing overheads [13, 14].

Memory fragmentation is a serious concern across the spectrum of modern computing platforms and databases [24, 28, 36]. Traditional memory allocators without compaction can suffer from catastrophic memory fragmentation [37, 39]. Furthermore, while fragmentation increases memory usage of in-memory data stores by up to 69% (e.g., Redis, MongoDB, and VoltDB) [24, 28, 33, 34, 47], it also has a negative impact on their performance due to memory sparsity [24]. For distributed systems, the fragmentation problem is particularly severe as the memory space may consist of hundreds of physical nodes and the stored data can be replicated multiple times for fault tolerance. Each additional machine can potentially increase the amount of wasted fragmented memory (§2.1.2).

We propose CoRM, a shared memory system that exploits RDMA for fast remote accesses and supports memory compaction. Additionally, CoRM's compaction is RDMA-safe: objects are still accessible via RDMA (§3.5) and the user is guaranteed to observe their consistent state (§3.2.3) even if they have been relocated by the compaction algorithm. To facilitate compaction, objects in CoRM are associated with block-local object IDs that are randomly generated at object allocation time. Our compaction algorithm is probabilistic: two memory blocks can be compacted into one only if they are conflict-free, that is, the objects in the two blocks do not have the same IDs. This enables a trade-off between compaction probability and space overhead: the larger the object ID space, the higher the compaction probability (§3.4). In some cases, relocated objects can experience higher access times because of indirections. Clients can detect this situation and fix the pointers to recover efficient one-sided RDMA access (§3.2).

CoRM is designed to provide memory compaction to RDMA-accelerated DSM systems such as FaRM [14] without compromising

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452817

strict consistency and one-sided RDMA accesses. It can reduce memory occupation of FaRM by up to 6x for synthetic workloads and up to 2.9x for real applications. We conduct a theoretical study on the memory compaction guarantees provided by CoRM and compare it with Mesh [36], a compaction algorithm for C/C++ applications. We extensively benchmark CoRM, showing that it matches the read throughput of FaRM (up to 380 Kreq/s per client) and that its compaction can increase the read throughput by 25%.

2 Distributed Shared Memory Systems

Distributed Shared Memory (DSM) systems [9, 10, 14, 46] provide an abstraction where the memory of multiple different physical nodes is viewed as a single unified memory space. Applications running in a DSM can randomly access their local memory or the memory of remote nodes. To implement this abstraction, DSMs provide APIs for managing (i.e., allocating and freeing) and accessing (i.e., reading and writing) memory. Memory accesses are translated to load/stores if they target local memory, otherwise, they lead to requests that are sent over the network to the target nodes.

2.1 Concurrent memory allocators

In a DSM system, the memory of a process can be allocated by the application tasks running locally or by remote ones. Hence, a DSM node needs to manage concurrent memory allocations, as it would be in normal multi-threaded applications with the addition that now allocation requests can also come from the network.

Concurrent Memory Allocators (CMAs) have two main requirements: (1) scale with the number of threads managing memory; (2) maintain low memory fragmentation, maximizing memory efficiency. We define memory fragmentation as the ratio between the amount of memory granted by the operating system to a process and the amount of memory that the process is effectively using.

2.1.1 Scalability. To improve scalability, most CMAs [6, 9, 14, 16, 25, 36] adopt a two-level architecture, as depicted by Figure 1. In this model, each thread is served by a thread-local allocator that has its free memory heap: the memory allocation requests are served from this heap without the need for global synchronization. If a thread-local allocator runs out of memory, it requests new memory from the process-wide allocator. The process-wide allocator may allocate memory directly from the operating system.

To avoid frequent accesses to the process-wide allocator, which can potentially require synchronization, the thread-local allocators fetch more than one free page at a time. The set of free pages that are fetched from the process-wide allocator in a single access is defined as *block*. Blocks are used to store objects belonging to predefined size classes: a given memory block can be used only for storing objects of a certain size. An object is allocated in the smallest size class that can fit it. Therefore, the size classes must be carefully chosen to limit internal fragmentation.

The block-based approach introduces a trade-off between synchronization overhead and memory efficiency: the larger the block size, the less the number of accesses to the process-wide allocator, hence the better the scalability. On the other hand, larger blocks can lead to memory inefficiency if the allocated blocks are not fully utilized by the allocating thread.

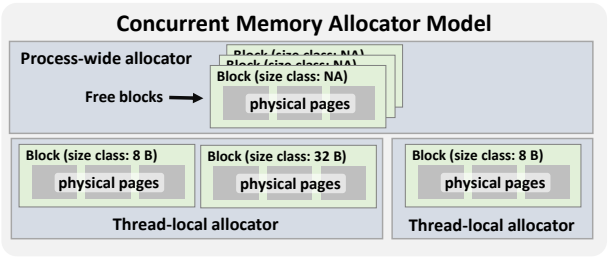

Figure 1: Concurrent Memory Allocator: each thread allocates memory from its thread-local allocator, which requests memory blocks from the process-wide allocator.

2.1.2 Memory Fragmentation. High memory fragmentation can be caused by irregular allocation spikes or low usage of particular size classes [4]. Allocation spikes happen when an allocator experiences a high volume of allocations followed by only a partial set of deallocations. The issue is that it is not guaranteed that these deallocations will lead to memory being freed up: In fact, blocks containing at least one object cannot be released back to the process-wide allocator. In this scenario, the threads can be left with many blocks that are scarcely utilized and cannot be released, causing memory fragmentation.

Low occupancy of some size classes can also be a source of memory fragmentation. Consider an example where an application allocates an object of size s on each of its T threads: the thread-local allocators will allocate the object from their local memory, increasing the memory requirement to $(T \cdot s)$. However, if the thread-local allocators do not have blocks of the requested size-class, they will request a new block from the process-wide allocator, potentially allocating up to $(T \cdot B)$, where B is the block size. If objects of size s are uncommon, most of the newly allocated blocks will have very low occupancy: e.g., if there is only one object per thread of that size, the overall unused memory is $T \cdot (B - s)$ bytes.

2.1.3 Memory Compaction. To reduce memory fragmentation, CMA systems can adopt memory compaction strategies. In principle, these strategies consist of taking a set of scarcely utilized blocks and merging them into one, releasing the others. This process needs to preserve the accessibility of the compacted objects so clients could still access them. A common strategy is to employ an indirection table that maps object keys to their current memory location [32] allowing systems to move objects freely in memory by updating corresponding entries in the table. However, the use of indirection tables results in revoking direct RDMA access to stored objects (§2.2.1) leading to a 2x reduction in read throughput [13, 14].

An alternative approach has been recently proposed by Mesh [36]: it does not use indirection tables and, instead, exploits virtual memory functionality to compact memory without the need for changing virtual addresses of relocated objects. Mesh merges the content of scarcely utilized blocks into one block and then updates the virtual-to-physical mapping of the affected blocks making their virtual addresses to point to the single resulting block holding the relocated objects. Mesh requires the relocated objects to reside at the same offset as they did in their original blocks (i.e., no conflicts) to preserve their virtual addresses. In Section 3.1.2, we discuss how this constraint limits the probability of compacting two blocks and show how our new compaction strategy can avoid this issue.

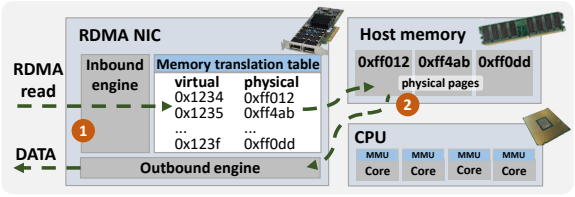

Figure 2: Accessing remote memory via RDMA read.

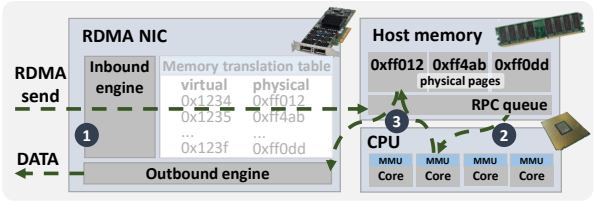

Figure 3: Accessing remote memory via RPC read.

2.2 RDMA-accelerated DSM systems

To efficiently enable the shared memory abstraction, DSM systems must minimize the overheads of remote memory accesses. To narrow the performance gap between local and remote accesses, modern DSM systems employ RDMA to ensure low latency and high throughput [1, 2, 9, 14]. RDMA is a mechanism that allows one machine to directly access the memory of other remote machines across the network. The RNIC on the sender side reads data directly from the sender's memory and injects it into the network. On the receiver side, the RNIC receives the data and writes it directly to the host memory, bypassing the operating system and minimizing the end-to-end latency. RDMA-enabled hosts communicate through either reliable or unreliable Queue Pairs (QPs). In this work, we consider only reliable QPs, as it is the only type of QP that supports one-sided RDMA read operations.

In principle, all DSM operations (i.e., memory allocation and freeing, read and writes) can be implemented as RDMA one-sided operations. However, most of them would require multiple round-trips, hindering the performance gains given by RDMA: e.g., allocating memory with only RDMA one-sided operations would require to read, modify, and write back the allocation state of the target node (without considering that multiple nodes of the DSM can be targeting the same memory at the same time). We take the approach of FaRM [14], which accelerates remote reads with RDMA, while implementing other operations with Remote Procedure Calls (RPCs). We now describe how RDMA can be used to accelerate read operations and how it can help to have low-latency RPCs.

2.2.1 RDMA reads. To expose memory over the network and allow other nodes to read it, a DSM node must register the memory on its RNIC. Memory registration consists of pinning the associated memory pages in physical memory and copying the related page table entries to the Memory Translation Table (MTT) of the RNIC. The RNIC generates keys for local and remote accesses, namely *l_key* and *r_key*. The memory region can be accessed by any local QP which has the *l_key* and by remote endpoints having the *r_key*. Figure 2 shows an RDMA read example: when the RNIC receives an RDMA request, it translates the target virtual address using its MTT into the corresponding physical page (1), then it uses the computed physical address to issue a DMA read towards the host memory (2), sending back the read data.

RDMA reads and memory compaction. RDMA accesses are issued by specifying the virtual addresses of the data in the memory of the target machine. This characteristic often limits the memory compaction capabilities of DSM systems exploiting RDMA. Memory compaction requires to move allocated data in memory, changing the virtual-to-physical mapping and potentially changing the virtual addresses themselves. When this happens, remote peers cannot access that data via RDMA anymore because the virtual addresses they hold may become invalid. For this reason, DSM systems like FaRM [14], do not support compaction and therefore can suffer from memory fragmentation, which can be especially dangerous in scenarios like the ones discussed in Section 2.1.2.

2.2.2 RPC operations. Memory management operations and accesses can be handled via RPC. On a DSM node there is a number of worker threads that are in charge of running application-defined tasks and handling RPC calls. RPC requests are pushed into an RPC queue that is shared between the worker threads. The handling of RPC requests can be accelerated with RDMA by letting remote peers push the RPC requests directly to the RPC queue [21].

Figure 3 shows an example of an RPC read operation: the memory read request arrives at the RNIC and is copied directly into the RPC queue (1). The DSM worker threads regularly poll the RPC queue to check for new requests. When a thread gets a memory request (2), it serves the request and sends the result back to the initiator (3). In this case, the virtual-to-physical address translation is done by the MMU of the core where the worker thread is executing.

3 CoRM

CoRM[1] is a memory management system that exploits RDMA to improve both latency and throughput of memory accesses. With CoRM, we show that it is possible to enable memory compaction in RDMA-accelerated DSM systems without introducing indirection to take full advantage of one-sided RDMA operations.

Relation to other systems. DSM systems like FaRM sacrifice memory compaction in order to employ RDMA to accelerate remote communications. CoRM is designed to provide memory compaction to RDMA-accelerated DSM systems such as FaRM without compromising strict consistency, but requiring storing extra metadata in object headers (§4.4). Since CoRM's API mimics FaRM's API and only adds a maintenance call for releasing unused virtual addresses (§3.3), we believe our compaction strategy can be integrated to FaRM without extra effort.

System	Type	RDMA	Mem. Compaction	Vaddr Reuse
Mesh [36]	Allocator	✗	✓	✗
FaRM [14]	DSM	✓	✗	-
CoRM	DSM	✓	✓	✓

Table 1: Comparison of FaRM, CoRM, and Mesh.

The compaction strategy of CoRM is similar to Mesh [36], which is a memory allocator supporting memory compaction for C/C++ applications. Unlike Mesh, CoRM's memory compaction strategy can additionally merge blocks having objects placed at the same offsets, improving the compaction probability (§3.4). Furthermore,

[1] https://github.com/spcl/CoRM

Table 2: CoRM APIs

API	Type	Pointer Correction (§3.2)	Description
`ctx* CreateCtx(char* ip,int port)`	Initialization	N/A	connect to a remote memory allocator
`addr_t ctx::Alloc(size_t size)`	RPC	N/A	allocate object with a given *size*
`int ctx::Free(addr_t &addr)`	RPC	Yes	free object at a given *address*
`int ctx::Read(addr_t &addr,char* buf,size_t size)`	RPC	Yes	read object to *buffer* with a given *size* using RPC
`int ctx::DirectRead(addr_t &addr,char* buf,size_t size)`	RDMA	No	read object using one-sided RDMA read
`int ctx::ScanRead(addr_t &addr,char* buf,size_t size)`	RDMA	Yes	read object by reading and scanning the whole block which contains the object
`int ctx::Write(addr_t &addr,char* buf,size_t size)`	RPC	Yes	write content of *buffer* to the remote object using RPC
`int ctx::ReleasePtr(addr_t &addr)`	RPC	Yes	an explicit call to release old object *address* using RPC

Mesh does not solve the problem of virtual space exhaustion, which is addressed in CoRM's design by tracking the block in which each object was initially allocated (§3.3). For that, CoRM requires users to perform additional actions, called pointer correction (§3.2) and pointer release (§3.3), that have little effect on performance (§4.3.2).

Table 1 recaps the characteristics of Mesh, FaRM, and CoRM. *RDMA* indicates if the system supports RDMA-accelerated remote accesses; *mem. compaction* indicates if the system supports memory compaction; *vaddr reuse* tells if the system can reuse virtual addresses after compaction, avoiding virtual address space exhaustion (§3.3). CoRM improves over Mesh by introducing a new compaction strategy that increases the probability that two memory blocks can be compacted and it extends FaRM by introducing support to memory compaction, making it resilient to memory fragmentation while still preserving strong consistency and one-sided RDMA accesses.

Interface. The API is shown in Table 2. Users can allocate and free objects using *Alloc* and *Free*. Allocations return 128-bit pointers that can be used to access objects. Those pointers include the actual 64-bit object address and RDMA-related metadata such as the *r_key*. Read operation can be used to read an object given its pointer. CoRM supports two types of reads: via RPC (*read*) and via one-sided RDMA (*DirectRead*). One-sided RDMA reads are lock-free and are performed without involving the remote CPU. The application is guaranteed to observe a consistent object state even in case of concurrent writes to the same object. To support lock-free consistent RDMA reads, we embed versioning information into the object itself [14]: a version number is stored with each cacheline, allowing the reader to check consistency by verifying that all cachelines that have been remotely read have the same version number. This strategy relies on cache-coherent DMA and requires cacheline-aligned allocation. If the consistency check fails, the RDMA read needs to be issued again. To update an object, the user can use the *Write* call to write a local buffer to a remote one.

3.1 Memory allocation and compaction

CoRM supports compaction that does not compromise the object pointers of the clients. Our system exploits RDMA-aware memory remapping to silently move objects across physical memory blocks while preserving their virtual addresses and RDMA access keys.

3.1.1 Allocation algorithm. CoRM uses a concurrent memory allocator as described in §2.1, similar to most memory systems [9, 14, 16, 25, 36]. The allocator supports a list of distinct 8-byte aligned sizes, that are chosen to reduce the average internal fragmentation due to round up to the nearest size class. The process-wide block allocator in CoRM can allocate blocks with sizes that are multiples of 4 KiB (i.e., a normal-sized page). However, CoRM can easily be extended to work with huge pages to reduce the number of pages.

Block allocation is performed in two steps: first, we allocate a physical page using the *memfd_create* system call [29]; then the allocated physical page is mapped to virtual space using *mmap*. The process-wide allocator keeps track of all virtual-to-physical mappings. The *memfd_create* call creates an anonymous file that lives in RAM, which can be modified, truncated, and memory-mapped as a regular file. To reduce the number of allocated file descriptors, CoRM allocates files of 16 MiB and uniquely identifies physical blocks as a tuple of the file descriptor and the page offset in the file. The block allocator is also responsible for registering allocated blocks with the RNIC to enable remote access (§3.5).

3.1.2 Compaction algorithm. CoRM can compact blocks of the same size class belonging to the same machine. The high-level idea of the compaction algorithm is to find two blocks of the same class with low utilization and copy objects from a block (i.e., source) to the other (i.e., destination), as illustrated in Figure 4. Once all objects have been copied, the source block can be deallocated and its virtual address is remapped to the physical address of the destination block. At this point, we have two virtual addresses pointing to the same physical page. The mapping is updated also on the RNIC in order to preserve RDMA access to the objects of the source block (§3.5).

A similar approach to memory compaction has been proposed by Mesh [36]. However, in Mesh compaction is only possible when no objects in the blocks occupy the same offsets. The limitation comes from the fact that Mesh is a plug-in replacement for malloc in C/C++ programs where the applications can freely read and write memory by using virtual addresses with load/store instructions. In that case, the page virtual address can be remapped transparently (i.e., the translation is performed by the MMU) but the object offsets cannot change. Thus, Mesh can compact blocks only if their objects do not conflict in offsets. In DSM systems, however, users always use explicit read/write functions to access memory: these are needed to resolve remote pointers and to enable concurrency control.

CoRM takes advantage of the DSM programming model and relaxes the requirement of the compacted objects keeping the same offsets after compaction. To achieve that, CoRM assigns identifiers (IDs) to each object in the block. The ID is unique only within a single block and is generated randomly using a uniform distribution. An object is uniquely identified in memory by the block address and the object ID, which is stored in the header of the object. This design choice allows CoRM to compact two blocks only if the objects in them do not have the same IDs. Differently from Mesh, where the compaction condition is based on offsets, in CoRM the object

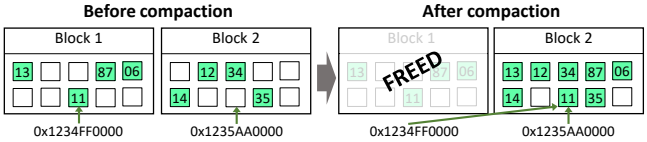

Figure 4: Compaction of two blocks without conflicts on object offsets or IDs. After compaction, the virtual addresses of block 1 point to block 2.

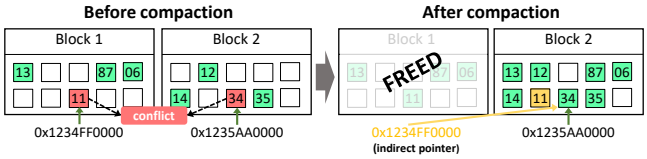

Figure 5: Compaction with offset conflicts. CoRM can compact the blocks by moving objects. Accesses to moved objects with indirect pointers need additional pointer correction.

IDs are random and the IDs size can be tuned (16 bits by default), improving the compaction probability. In fact, while blocks can also have conflicts in the object IDs, they are less probable than offset conflicts (§3.4). Figure 5 shows an example where the blocks to compact have conflicting offsets. In this case, a Mesh-like approach would not be able to compact, while CoRM can move the conflicting objects to a different offset, preserving their IDs.

During compaction, it is preferable to preserve the offset of the objects as it preserves the virtual addresses of compacted objects. When it is not possible (i.e., because of offset conflicts), CoRM is free to move objects to new offsets within the block. The moved objects can still be found by looking for their object IDs which is included in the 128-bit pointers returned by the *Alloc* function. A pointer that points to an object that has been moved to a different offset is defined as **indirect**. Instead, pointers to objects that have not been moved even after compaction are defined as **direct**. When accessing an object with an indirect pointer, the virtual address translation will point to the correct block but the object will not be found at the given offset. In this case, CoRM will need to perform an additional action, called *pointer correction* (§3.2), in order to retrieve the requested object. The pointer correction is implicitly performed during all API calls but one-sided *DirectRead* (see Table 2).

3.1.3 Compaction policy. CoRM calculates a fragmentation ratio for each size class. CoRM triggers compaction for a size class if its fragmentation ratio exceeds a fragmentation threshold. The fragmentation threshold can be tuned for each size class depending on its compaction probability (§3.4). CoRM can additionally start compaction when an allocation fails due to shortage of memory.

3.1.4 Compaction mechanism. In CoRM, each thread has its private memory allocator. Therefore, the blocks that can be compacted may belong to different threads, preventing efficient lockless memory compaction. To address this issue, CoRM selects one of the worker threads as a compaction leader, that performs compaction in two stages: *block collection* and *block compaction*. During the block collection, the leader broadcasts a collection request to all other threads, asking for sufficiently low-occupancy blocks of a certain size-class. In the second stage, after all threads reply to the collection request, the leader can start the compaction algorithm. Our two-stage design removes the need to have costly concurrent

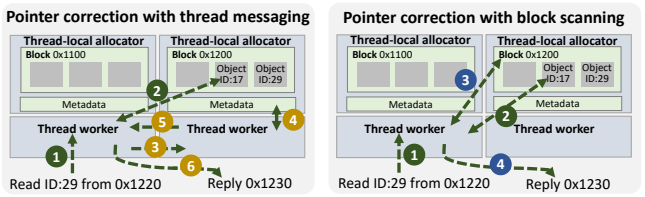

Figure 6: CoRM supports two approaches for pointer correction: thread messaging, and memory scanning. The worker received a request to read object ID:29 from address 0x1220, which belongs to block 0x1200, but the object has been moved to address 0x1230.

data structures since CoRM holds an invariant that any block is owned by at most one thread. CoRM tries first to compact the least utilized blocks, as they have fewer elements and induce fewer offset collisions. During compaction of two blocks, besides coping objects, CoRM also merges metadata of the affected blocks. The metadata is a hash table that keeps a mapping between object IDs and offsets used only for fast pointer correction (§3.2).

3.2 Pointer correction

As the compacted objects can move within memory after compaction, the virtual pointer held by the user may not directly point to the desired object. To avoid searching an object in its block, each user's object pointer contains the *offset hint* where the object is expected to be in the block. CoRM always optimistically accesses the object at the hinted offset (using a load instruction) and then checks its ID. If the ID of the accessed object does not match the one in the used pointer, then CoRM performs a search to find the requested object. Once the object is found, the hint inside the object pointer is updated to the new offset, making the pointer *direct*.

3.2.1 RPC calls. Pointer correction is transparent to the user when RPC-based calls are used to read and write objects. We show two approaches that can be used to find objects accessed with indirect pointers: the first approach uses inter-thread communication, while the second directly scans the block with the requested object in order to find it. Figure 6 illustrates both approaches. Whenever an RPC read call is served (1), CoRM checks if the object ID of the hinted object matches the ID of the requested one (2). If this check fails and the solution with inter-thread communications is employed (left), then the thread serving the RPC request forwards it to the thread owning the requested block (3). In this way, the owner thread can quickly query metadata of the block to determine the position of the object (4). For each block we keep a thread-local mapping between object IDs and offsets stored in it. Once the object is found, the owner thread sends the corrected pointer back to the thread handling the RPC request (5) allowing it to complete the request and reply to the client (6). While this approach is more efficient for large block sizes since it avoids expensive scans, it can delay the request processing if the owner thread is busy with other activities (e.g., compaction). Instead, the *block-scanning* approach does not require inter-thread communications by letting the thread serving the RPC call scan the block with the requested object, even if it belongs to another thread. The thread compares the IDs of all allocated objects with the ID of the requested one to find the new offset (3). After the object is found, it is sent back to the client (4).

3.2.2 RDMA calls. RDMA read accesses are not served by the CoRM worker threads but are directly performed by the RNIC. This implies that the pointer correction mechanisms we described above cannot be applied for *DirectRead* calls and that we need to move the pointer-correction activity to the client side. Similarly to the RPC case, a client performing an RDMA read can detect if the read object is correct by comparing its ID to the one stored in the accessed pointer. If the two do not match, then the client has two options: (1) issue an RPC-read, triggering the pointer correction mechanisms of above; (2) issue an RDMA read of the entire block where the object is stored. With (2), the CoRM client-side library scans the block in order to find the requested object. We define (2) as *ScanRead*.

3.2.3 Consistency. Clients interfaced with CoRM are guaranteed to observe consistent objects even in the case reads are interleaved with writes or happen while memory compaction is in progress. While RPC calls can directly ensure consistency by employing explicit locking of object headers, this is not true for RDMA reads.

When performing a *DirectRead*, CoRM issues a one-sided RDMA read to retrieve the object and then checks if the read object is valid. Other than the case described in Section 3.2.2, there are two reasons for which a read object might be invalid: (1) the read object is being updated by a concurrent write (i.e., the object is corrupt); (2) the read object is under compaction. To detect the first case, we store the object version into the header of the object and in the first byte of each cacheline, as proposed by FaRM [14]. Writes to the object increase the object version. Clients can verify the validity of the read objects by checking that the version numbers match. To detect the second case, we store a lock state into the object header (2 bits). At the beginning of a compaction process, CoRM locks all objects that are going to be compacted. If a client reads a locked object, then the object is invalid. In case the read object is detected as invalid, then the read is repeated after a backoff period.

3.2.4 Fault Tolerance. The current implementation of CoRM is not fault tolerant. Thus, we assume that if any thread fails then the whole process fails. Fault tolerance is an interesting area of future work. CoRM could employ a fault-tolerant replication protocol (e.g., [15, 18, 22, 42]) to withstand failures.

3.3 Pointer release

The main effect of the CoRM compaction scheme is to reduce the physical memory utilization by reducing fragmentation. However, this does not automatically translate to lower utilization of the virtual address space since all virtual addresses are preserved after compaction. As a result, if virtual addresses are not released in the long run, solutions like CoRM or Mesh can run out of virtual space.

To address this problem, CoRM stores the address of the virtual block where the object has been initially allocated in the header of each object. This allows us to keep track of how many objects that have been moved out from an old virtual address are still valid and can still be accessed. Once there are no more of such objects, i.e., they have been deallocated with *Free* calls, then CoRM can safely assume that the virtual address can be reused.

Additionally, CoRM provides the *ReleasePtr* call allowing clients to explicitly release an old object pointer without actually freeing the corresponding object. This call can be used by the clients to communicate that all copies of the old pointer have been corrected

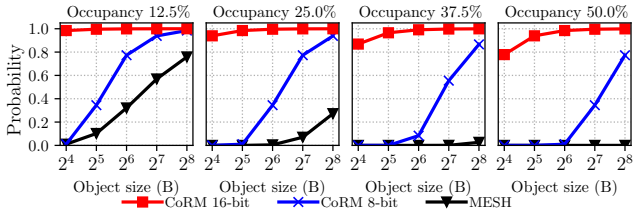

Figure 7: Compaction probability of two random blocks depending on occupancy and size class. CoRM probabilities are reported for 8-bit and 16-bit object IDs.

and it is safe to reuse that address. The clients must ensure that the pointer is not reused to address the same object once it has released it. CoRM always notifies the user if it uses an old pointer. Note an old pointer can be corrected to become direct (i.e., having correct offset), but it will still reference the old block address. We expect *ReleasePtr* calls to be rarely used (only when virtual space is almost exhausted) as *Free* calls can implicitly free unused virtual space.

3.4 Probability of compaction

A key threat to CoRM's memory compaction capabilities are collisions in object IDs. This issue is similar to the one that Mesh has for allocation offsets, where a conflict in the offsets prevents two blocks from being compacted. We now define the probability of compaction of two blocks B_1 and B_2 depending on their occupancy.

We denote $p(B_1, B_2)$ as the probability of compacting the block B_2 into the block B_1. Note that the probability $p(B_1, B_2)$ is equal to $p(B_2, B_1)$. As only blocks of the same type are compactable, we denote s as the total number of objects that can be stored in a block. We denote n as the total number of different object identifiers a block can have. For Mesh, n is the number of objects a block can store, which is equal to s. For CoRM, n is the total number of possible object IDs, which is 2^x, where x is the number of bits used to store the object IDs. The value of x is a parameter of CoRM and can be used to tune the compaction probability. The larger object IDs, the lower the probability of ID conflicts but increases memory usage, as they are stored in the header of each object (§3.3). This is a key difference from Mesh, where n depends solely on the block and the object class size. E.g., for 16 byte objects, a 4 KiB block can store 256 objects, whereas for 128 byte objects the same block can fit only 32 objects. In this case, if CoRM would use 8-bit IDs, then it would have the same compaction probability of Mesh. However, already for larger size classes, the compaction probability of CoRM with 8-bit IDs will become higher than Mesh, because the number of offsets that Mesh can use would decrease.

We define b_1 and b_2 as the number of objects stored by B_1 and B_2, respectively. Assuming that object IDs are randomly generated with a uniform distribution, the probability of no collisions is:

$$p(B_1, B_2) = \begin{cases} \frac{\binom{n-b_1}{b_2}}{\binom{n}{b_2}}, & \text{if } b_1 + b_2 \leq s \\ 0, & \text{otherwise,} \end{cases}$$

where $\binom{n}{k}$ is the binomial coefficient, $\binom{n-b_1}{b_2}$ is the total number of blocks not using any ID of the objects stored in B_1, and $\binom{n}{b_2}$ is the total number of blocks that have b_2 allocated objects. When the

sum of objects in two blocks is greater than the total number of slots, then the blocks are not compactable.

Figure 7 compares the probability of two random blocks of 4 KiB being compactable depending on their occupancy (four sub-figures) and size classes (x-axis). CoRM performs better than Mesh in all situations. In particular, for large object sizes, CoRM succeeds even for high occupancy using only 8-bits for identifiers, whereas Mesh has near-zero probability. With 16-bit IDs, CoRM consistently provides a higher chance of compaction regardless of block occupancy. We conclude that CoRM is a better choice for memory compaction in DSM systems since it has a higher likelihood of compacting memory blocks even with 50% utilization.

3.5 Preserving RDMA access

When an RNIC receives an RDMA request, it translates the requested virtual address to a physical one using its Memory Translation Table (MTT), which contains virtual-to-physical page translation entries. Whenever a new memory region is registered, a new entry is installed in the MTT, enabling RDMA access to that region. However, if the page is remapped because of compaction (i.e., the virtual address is associated with a different physical page), then also the corresponding entry in the RNIC's MTT must be updated. Otherwise, RDMA accesses referencing that virtual address will access the wrong physical page. One possible solution is to re-register the pages every time they get remapped. However, according to the RDMA specification, this will cause the invalidation of the *r_key*: all clients would need to be informed of this event and update the *r_key* to the remote objects. A client making an RDMA access with an invalid *r_key* causes the RDMA QP disconnection, potentially leading to high overheads for recovering (e.g., re-establishing the connection), which can take few milliseconds.

To avoid these overheads, CoRM supports three approaches for restoring RDMA accesses after page remapping, both preserving the *r_key* of the original registration. The first approach relies on the *ibv_rereg_mr* call, which re-registers the memory and preserves its access keys. While this approach works on any commodity RNIC, we observed that RDMA accesses to memory regions under re-registration break the QP connection, which complies with the InfiniBand specification [3]. The second and third approaches rely on the On-Demand-Paging (ODP) capabilities of RNICs. ODP does not compromise the connection but only works on modern RDMA devices. ODP is a technique that allows RNICs to implicitly request the latest address translation entries from the OS when pages are invalid in the MTT or if their mapping changed. ODP enables consistency between OS and RNIC translation entries. The second approach solely relies on ODP, whereas the third approach also exploits ODP prefetching to reduce the overhead of MTT misses.

Figure 8 shows the latency of the three solutions on a Mellanox ConnectX-5 card. In all cases, the address must be first mapped with an *mmap* call, which takes around 2 μs. With the first approach, the *ibv_rereg_mr* call takes approximately 9 μs to update the translation entry on the RNIC. During that time, the virtual address is unavailable and all remote accesses to it will cause QP disconnections. Using the second approach, the CPU does not need to explicitly fix entries in the MTT as the RNIC will resolve inconsistencies via ODP. However, we observe that the first RDMA read from a page that has been remapped incurs in a 63 μs overhead: this is the cost

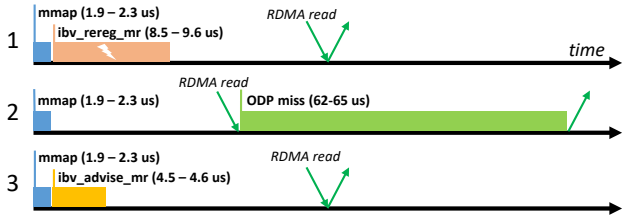

Figure 8: RDMA remapping latencies for three strategies.

paid by ODP to invalidate and install the new mapping into the MTT. Subsequent reads have a 2 μs latency. To reduce the overhead of ODP MTT updates, verbs support the prefetching of MTT entries with the *ibv_advise_mr* call. The latency of prefetching is 4.5 μs, which can potentially save the ODP overhead if the read requests arrive after the prefetching completed. This is the default solution adopted by CoRM for memory registration and remapping.

4 Evaluation

We evaluate the performance of memory access and management operations with CoRM and its memory compaction capabilities. We first study the performance of CoRM with direct pointers and indirect pointers to evaluate performance degradation when objects move. Then we measure the performance of CoRM during compaction under synthetic and YCSB [12] workloads. Later, we evaluate CoRM's ability to compact memory and compare it with Mesh for synthetic workloads and real applications.

Experimental setup. The experiments are conducted on an isolated cluster with 12 machines interconnected with an FDR InfiniBand network. Each machine is equipped with ConnectX-3 InfiniBand network cards and two 3.40 GHz Intel Xeon E5-2630 v3 CPUs (16 hardware threads). CoRM is implemented in C++ and depends on: *libibverbs*, an implementation of the RDMA verbs; *librdmacm*, an implementation of the RDMA connection manager; and *libev*, a high-performance event loop.

We deploy CoRM on a dedicated machine and spawn clients on other interconnected machines: all clients connect to CoRM remotely and send requests over the network. If not stated differently, we configure CoRM with blocks of 4 KiB and 8 worker threads.

4.1 Operations Latency

CoRM enables lock-free RDMA-accelerated DSMs with memory compaction capability. To achieve this, all memory accesses specified by the applications must be issued through the CoRM API and cannot directly use load/store instructions for local accesses or raw RDMA calls (e.g., ibverbs) for remote ones. We now discuss the latency of the memory access and management operations of CoRM. The operation latency is defined as the time observed by a client to complete the operation (i.e., round-trip time).

Latency of direct accesses. Figure 9 shows the median latency of different CoRM functions when all pointers are direct (i.e., no object has been relocated because of memory compaction). To show the overhead introduced by CoRM, we also report the round-trip latencies of RPC and raw one-sided RDMA reads. RPC operations are implemented using raw Send/Recv RDMA operations. The benchmark first loads CoRM with 10,000 objects of each size-class (≈40 MiB in total), then starts the client to issue the different operations.

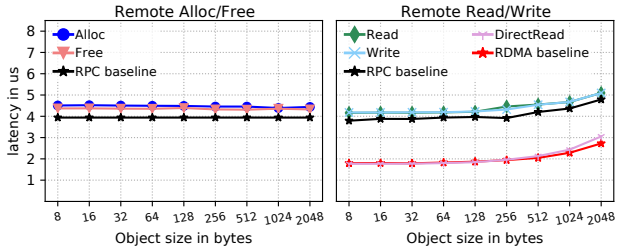

Figure 9: Median latency of CoRM with direct pointers.

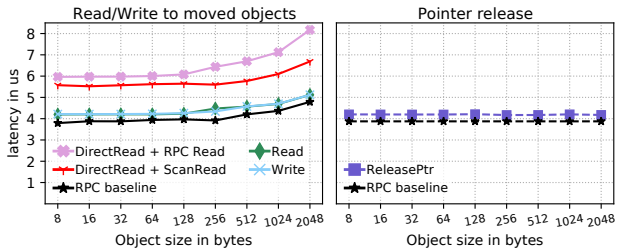

Figure 10: Median latency of CoRM with indirect pointers.

The round trip latencies of RDMA requests are under 4 μs. For comparison, the TCP/IP traffic over the same link using IPoIB has a latency of 17 μs. Alloc and Free calls add about 0.5 us to the base RPC call to manipulate the memory. However, this number accounts for the allocation case when the thread-local allocator always has a block of the requested size-class. If this is not the case, the thread-allocator needs to request a new block, increasing the allocation latency by an additional 5 us (i.e., to allocate a new block and register its memory on the RNIC). Read and write over RPC have similar performance as they communicate the same amount of data. The round-trip latency of the raw RDMA reads can be as low as 1.7 us. The consistency protocol which checks the integrity of the read data in DirectReads only increases the latency for large objects. For objects smaller than 256 bytes, the DirectRead call has approximately the same latency as a raw RDMA read. These results show the benefit of using one-sided RDMA operations over Send/Recv calls for latency-sensitive read-only workloads.

Latency of indirect accesses. To measure the effects of indirect accesses, we measure the latency of read and write calls targeting compacted objects that have been moved to a different offset. In this case, the client is using an indirect pointer for accessing the objects (i.e., the pointer has an incorrect offset). Figure 10 (left) shows that there are no significant differences in latency between direct and indirect pointers for RPC requests. In throughput-intensive workloads, however, we observed a 5% drop in performance for RPC requests (§4.3). While RPC requests using indirect pointers can be fully handled by the CoRM workers, a failing DirectRead requires the client to perform an additional action to recover the requested object (i.e., pointer correction). We show the costs of the two pointer correction strategies: ScanRead and RPC read (see §3.2). We observe that with this configuration (i.e., blocks of 4 KiB) using an RPC call to backup a failed DirectRead is more expensive than a ScanRead. However, for large block sizes, the first approach can be more efficient because it avoids to move the whole block over the network but it would still require more CPU time on the CoRM workers in order to perform the fix.

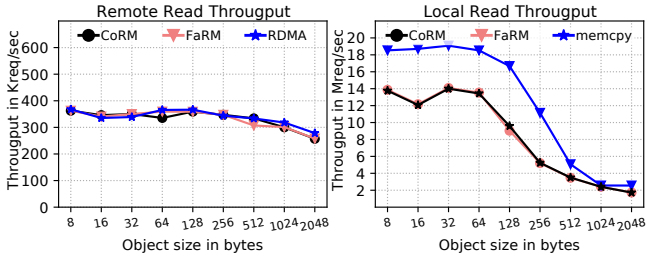

Figure 11: Read Throughput of CoRM, FaRM for remote and local accesses compared to direct memory accesses using RDMA and load/store instructions.

Once a client realizes that an object has been moved and it has updated all its references to that object, it can communicate to CoRM that it is now safe to reuse the old object's virtual address (§3.3). This is done with the *ReleasePtr* call. Figure 10 (right) shows the latency of this operation and the one of an RPC call for a reference. The pointer release costs about 0.3 us for indirect pointers and does not depend on the object size. This latency includes the time needed by CoRM to find the moved object (§3.2).

4.2 Read throughput

We now show the throughput achieved by CoRM when objects are remotely or locally accessed. We compare the read throughput achieved by CoRM with the one of FaRM[2] and the one achieved by load instructions for local reads and raw RDMA calls for remote accesses. The former scenario can be compared to an ideal Mesh-based DSM system where applications are free to use load/store instructions for local accesses. This case represents a best-case scenario for Mesh because it does not consider the additional synchronization overheads that should be introduced to keep consistency in case of concurrent read and writes.

4.2.1 Synthetic workload. We load the systems with a total of 8 GiB of data for each size class. The objects are accessed uniformly to ensure that data is accessed from DRAM and the clients have at most one outstanding request at a time. Figure 11 shows the results.

For remote accesses, raw RDMA shows the best performance (380 Kreq/sec for small objects) as clients do not need to verify cache versions. FaRM and CoRM have approximately the same performance as they both share the same approach for checking consistency. For small object sizes, the consistency check has negligible overhead, while it causes up to 2% slowdown w.r.t. raw RDMA for large objects. This is expected as large objects require more cachelines to be checked. CoRM's cache version approach for consistency checks was a deliberate choice to mimic FaRM. An alternative approach is to store a single checksum in the header of a record or after the record [30], which is potentially a better strategy for large records.

For local accesses, we compare the throughput of FaRM and CoRM to the one achieved by a *memcpy* call. FaRM is not more than 1.01x faster than CoRM for all sizes. FaRM and CoRM are both 1.33 times slower than *memcpy* because of the additional software layer. For larger object sizes, the performance is approximately the same for all three approaches as object accesses are memory bound.

[2]FaRM is not open-source, therefore, we emulated FaRM (including its cacheline consistency check) following the publicly available information.

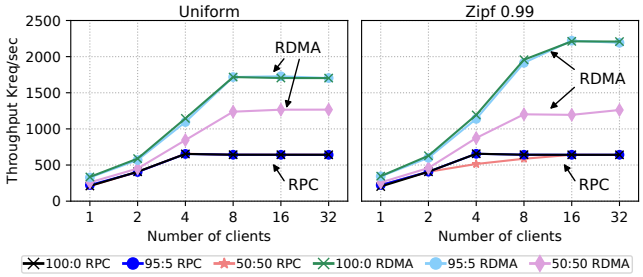

Figure 12: Aggregate throughput of CoRM under YCSB and uniform workloads for different read:write access ratios.

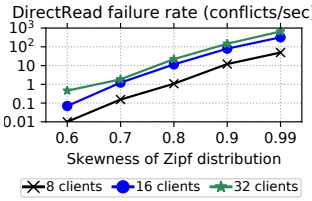

Figure 13: Aggregated failure rate for YCSB workloads with 50:50 access ratio.

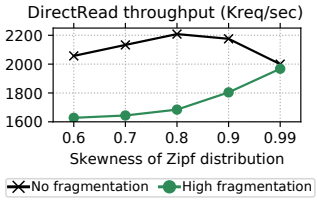

Figure 14: Aggregated read throughput for YCSB workloads with 100:0 access ratio.

4.2.2 YCSB workloads. To understand the read throughput that CoRM achieves under realistic workloads, we benchmark it under different YCSB [12] workloads. We load CoRM with 8,000,000 objects of 32 bytes and measure the throughput achieved by RPC and DirectReads. In Figure 12, we compare the performance of CoRM under Zipf ($\theta = 0.99$) and uniform distributions, while varying the number of clients. We report the total throughput of CoRM averaged over one minute period after a steady state under different read:write access ratios and numbers of clients. The lines tagged with RDMA use DirectRead to read objects, whereas RPC tagged lines use RPC reads. Writes are always performed using RPC.

RPC reads achieve lower throughput than RDMA reads, and the difference becomes larger for workloads with more reads. The aggregate throughput grows as we add more clients. However, in the RPC calls, the throughput stabilizes at 700K req/sec for more than 4 clients. DirectReads allows clients to achieve 1,250K req/sec for the 50:50 ratio, which is a 2x improvement over RPC reads. For read-dominant workloads, the throughput goes even higher up to 1,750K req/sec and 2,200K req/sec for uniform and Zipf distributions respectively, which shows the superiority of RDMA-based accesses over RPC. The Zipf workload shows a higher throughput because it has a better memory locality compared to the uniform distribution. This is beneficial for the destination RNIC: in fact, RNICs have limited cache for address translation entries, and once the cache is full the MTT will swap and incur in more misses [14].

4.2.3 DirectReads under contention. To evaluate the failure rate of lock-free one-sided reads, we measure the number of failed DirectReads over a period of one minute for the YCSB workload with 50:50 read:write ratio, while varying the skewness of Zipf distribution and the number of clients. The experiment was performed in the same setting as the previous experiment (§4.2.2).

Figure 13 shows that the number of conflicts increases with the number of clients and the skewness of the distribution. Nonetheless, even for the highly skewed workload ($\theta = 0.99$) and 32 clients, clients observed only 659 failed DirectReads per second, which is less than 0.1% of the total request rate.

4.2.4 Performance under fragmentation. To understand the impact of fragmentation on CoRM's throughput we benchmark its performance under YCSB workloads for two settings: *no fragmentation* (as in the previous experiment) and *high fragmentation*. To create the high fragmentation setting, we load CoRM with 16,000,000 objects of 32 bytes and then randomly deallocate the 50% of them. In Figure 14, we compare the performance of CoRM under the load of 8 clients, while varying the skewness factor of Zipf distribution.

Figure 14 shows that the unfragmented memory provides a 1.25x increase in throughput of DirectReads for moderately skewed access patterns over the fragmented one. For the highly skewed pattern ($\theta = 0.99$), CoRM's performance is approximately the same for both settings, as the clients mostly access the same small set of keys.

4.3 Compaction performance

To characterize the performance the CoRM compaction process we first study its latency (i.e., the time needed to perform the memory compaction), then analyze how the throughput of remote clients is affected when the server is busy in performing the compaction.

4.3.1 Compaction latency. The compaction process is composed of two phases: block collection and block compaction. During the block collection phase, the thread performing the compaction gathers blocks that are candidates for compaction (§3.1.4): i.e., non-full and belonging to the same size class. Figure 15 shows the time for these two phases. Since block compaction involves page remapping, which depends on the specific RNIC and remapping strategy (§3.5), we compare the compaction latencies measured on machines equipped with ConnectX-3 and ConnectX-5 with *ibv_rereg_mr*, and ConnectX-5 with ODP and prefetching. The block collection phase involves inter-thread communications, hence we compare two different CPUs: Intel Xeon (§4) and AMD EPYC 7742 @ 2.25GHz. Each experiment consists of allocating a single object of 32 bytes on each thread and then triggering the compaction process in order to measure its latencies.

Block collection. The collection time depends on the number of threads as all threads must reply to the collection request before the compaction can happen. In this experiment, each thread replies with its only allocated block to the compaction request. The results are shown in Figure 15 (left). On the Intel cluster, the collection takes 10 μs for 2 threads and 31 μs for 16 threads We notice that the collection takes on 2 μs on the AMD cluster, when 2 threads are used, which is 5x times faster than Intel. The two clusters show similar latencies when increasing the number of threads.

Block compaction. The compaction time depends on the block size and the number of blocks that can be compacted. In particular, it involves the checks of compactability requirements, the data copy, and the virtual address remapping on both the OS and the RNIC. The compaction time of a single block corresponds to the unavailability period of that block as the data in this block is not accessible because of compaction. In this experiment, blocks under compaction have only one object and are always compactable (i.e., no conflicts). Thus, the number of deallocated blocks after compaction is equal to the number of threads minus one.

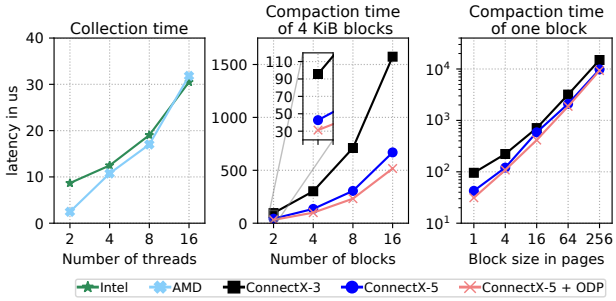

Figure 15: Latencies for collection and compaction for different numbers of threads and block sizes.

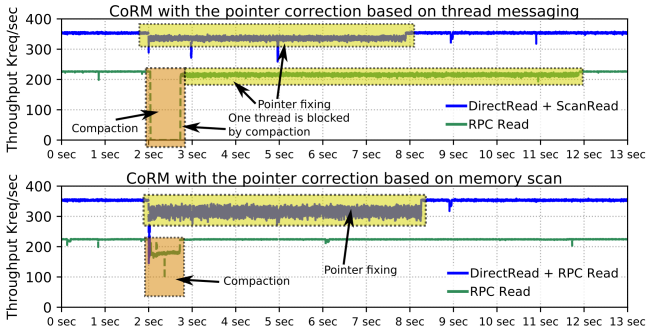

Figure 16: Read throughput of RPC and RDMA clients before, during, and after compaction. During the compaction, CoRM compacts 5,794 blocks. *(top)* Pointers are corrected with thread messaging; *(bottom)* pointers are corrected with the block scan strategy.

Figure 15 (center) shows how the block compaction latency varies with the number of blocks to compact. The block size is fixed to 4 KiB (i.e., one page). The compaction time grows linearly with the number of blocks (note that the x-axis is exponential). With ConnectX-3, the block compaction takes about 100 μs, with most of the time (70 μs) spent into the *ibv_rereg_mr* call. The same call on ConnectX-5 takes 7 μs: since this call is executed for each remapping, the difference in performance increases with the number of blocks to compact. The ODP strategy provides the best results as it does not require explicit memory re-registration.

Figure 15 (right) shows the compaction latency of a single block for different block sizes. The bigger the block size, the more physical pages need to be remapped. For 1-page blocks, the block compaction takes 100 μs for ConnectX-3, and the time grows linearly with the number of pages. For 1 MiB blocks, consisting of 256 pages, the compaction takes 12 *ms* for ConnectX-3. ConnectX-5 can remap pages faster, but the cost of the re-registration has the same trend as ConnectX-3. In case large blocks are used, the remapping time can be significantly reduced by using huge pages. For example, a 2 MiB page has the same remapping and re-registration latency as a 4 KiB page. Therefore, all reported data is applicable to huge pages.

4.3.2 Throughput during compaction. We evaluate the performance degradation of CoRM during compaction. In this experiment, we populate CoRM with 8,000,000 objects of 32 bytes and randomly deallocate the 75% of them. Then we start the throughput workload that repeatedly and sequentially reads all objects. After the warm-up, we trigger the compaction algorithm. In all experiments, the compaction is invoked after two seconds. We measure the throughput of a single client *before*, *during*, and *after* compaction.

Figure 16 shows the read throughput observed by clients accessing data with different types of read calls (i.e., RPC and RDMA) and different pointer-fixing strategies. We also study the performance of the two different approaches for pointer-correction, which is used when an object is accessed with an indirect pointer (see §3.2).

Thread messaging. We measure the throughput of two types of clients: one using RPC reads, and the other using DirectReads (RDMA). In the first experiment (top sub-figure), CoRM is configured to use thread messaging to find objects which are requested using RPC calls. The RDMA client needs to recover by itself in case the read fails (i.e., because the object has been moved). Here we show the case where the client issues a ScanRead to back up a failed DirectRead. The RPC client observes 700 *ms* of unavailability, as a requested object has been moved to a different offset and CoRM could not find it by using the passed pointer. The reason for that is the thread that owns the block with the moved object was busy with compaction and could not reply to correction requests from other threads. It happens because all blocks under compaction belong to the same thread that performs the compaction. We intentionally configured CoRM to perform long compaction without breaks to observe that scenario. During this long compaction, the thread managed to deallocate 5794 blocks. The unavailability period could be shorter if the compaction was configured with an upper bound on the number of compacted blocks. The RDMA client, on the other hand, does not observe the unavailability as it corrects the pointers using ScanRead, which reads the whole block with a requested object using RDMA and then scans it. After 8 seconds the RDMA client managed to correct 77,000 indirect pointers. The RPC client spends almost 9 seconds for all corrections. Overall, both clients observe 5% performance drop during pointer correction.

Memory scan. In the second experiment (bottom sub-figure), CoRM's worker threads opt for scanning the whole block to find the object requested using RPC. The RDMA client in this experiment uses RPC to correct pointers, instead of ScanRead. Compared to the previous experiment, the RCP client does not experience large unavailability periods, showing a 22% slowdown in throughput due to the fact that blocks under compaction are not readable during object migration and remapping. Once the compaction finishes, the client does not observe any significant performance drops even using indirect pointers, since a small percentage of pointers were indirect. In a similar experiment where all pointers are indirect (not reported here), the client observed only 5% slowdown. The RDMA client experiences performance degradation as it needs to issue an RPC to correct pointers to moved objects. This approach is slower than the one with ScanRead, which complies with our latency experiments. Overall, DirectReads provides 1.6x faster performance than RPC reads even in the presence of indirect pointers and when CoRM performs compaction.

Conclusion. The current experiments show that CoRM's pointer correction introduces only a *temporal* slowdown of 5% on read performance, whereas the use of indirection tables and RPCs has a *constant* 40% slowdown since it prevents the use of one-sided RDMA reads for fetching the data.

4.4 Compaction overheads and benefits

To enable its compaction strategy and reuse virtual addresses, CoRM stores an object identifier with each object. The object identifiers serve two scopes: (1) they enable CoRM to find relocated objects; (2) they allow detecting reads using pointers to relocated objects (i.e., same offset, different object ID). The size of the object identifiers determines the space-overhead per block and the capacity of CoRM of compacting blocks with a large number of objects. We define CoRM-*n* as the instance of CoRM using *n* bits for the object identifiers. In this experiment, we study the effects of the CoRM compaction strategy for synthetic and real-world workloads using different object identifier sizes and comparing the results with the compaction strategy proposed by Mesh.

4.4.1 Object identifiers overhead. Table 3 summarizes the space overheads for different CoRM configurations for 1 MiB blocks, which is the size of blocks in FaRM. In CoRM-0, object IDs are disabled and the compaction strategy is based on object offsets, as in Mesh. However, while the overhead of the object IDs is zero, CoRM still stores the virtual block address in the header of each object to be able to reuse virtual addresses (§3.3). Assuming a system with 48-bit physical memory pointers and 20-bits aligned blocks, the virtual block address that CoRM needs to store is 28 bits.

The CoRM's compaction algorithm cannot compact blocks storing more objects than the ones that can be addressed with a given identifier size. For example, CoRM-8, with 8 bits for object identifiers, can address up to 256 objects in a block, hence it cannot be used for 1 MiB blocks storing 2 KiB objects, which can hold up to 512 objects. To handle these cases, CoRM can be configured to use a hybrid compaction scheme where class sizes that cannot be compacted by CoRM-*n* are compacted with CoRM-0.

Mesh	CoRM-0	CoRM-8	CoRM-12	CoRM-16
0 bits	28 bits	28+8 bits	28+12 bits	28+16 bits

Table 3: Memory overheads for different compaction algorithms per object for 1 MiB blocks.

4.4.2 Synthetic traces. Figure 17 shows the amount of memory currently allocated (i.e., active memory) after a sequence of memory operations. We generate synthetic traces that first allocate 8 M objects of a given size (sub-figures) and then randomly deallocate a fixed portion (x-axis) of them. We also plot the active memory in case an ideal memory compactor, which always frees the non-utilized memory, and in the case in which no compaction is performed. We study the compaction capability of CoRM for 8, 12, and 16-bit object IDs and include the memory overhead introduced by CoRM to store object IDs in the reported data.

As the object sizes increase, both Mesh and CoRM are able to compact more memory. Mesh works effectively for large objects with high fragmentation but incurs in many conflicts for smaller messages and low deallocation rations. CoRM-8 and CoRM-12 perform always better than Mesh for the object sizes where they can be applied (e.g., ≥ 4 KiB objects for CoRM-8). CoRM-16 matches the ideal compactor already for 2 KiB objects and low deallocation rates (i.e., 0.5): this is because larger object IDs reduce the probability of conflict, hence increase the probability of compaction (object IDs are randomly chosen). For 256-byte objects, CoRM-16 requires

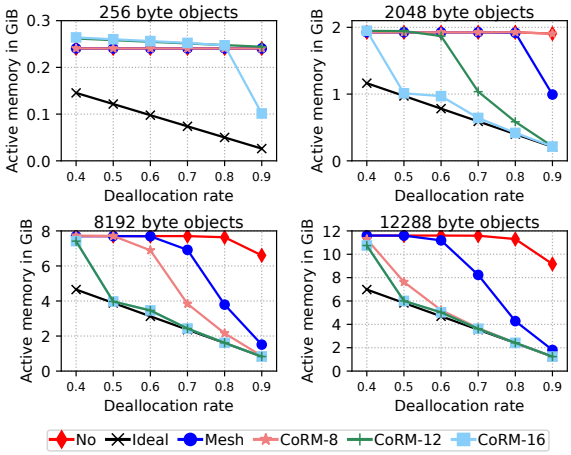

Figure 17: Active memory under synthetic workloads. CoRM is configured with 1 MiB blocks.

more active memory than the non-compacting case. This overhead comes from the fact that blocks with 256 byte objects have a very high probability of collision in object identifiers. Therefore, CoRM's overheads overwhelm its compaction capabilities.

4.4.3 Redis traces. Redis [40] is a popular in-memory data structure server. We extract memory traces from the *memefficency* unit test of the Redis test suite (v5.0.7). The traces are:

- *redis-mem-t1*: Default Redis configuration. It allocates 10'000 keys of 8 bytes each with values of sizes ranging from 1 to 16 KiB.
- *redis-mem-t2*: Redis is configured as an LRU cache with a capacity of 100 MiB. The trace first allocates 700,000 8-byte keys with values of 150 bytes each; then allocates 170,000 8-byte keys with values of size 300 bytes each.
- *redis-mem-t3*: Default Redis configuration. The trace allocates 5 keys containing data structures of 160 KiB each; then it allocates 50,000 keys with values of 150 bytes. It then removes 25,000 keys from the last batch of allocated keys.

For each trace, we report memory usage for a time point when the system had the highest fragmentation. To show the effects of the number of threads on the active memory, we run the traces multiple times, varying the number of threads used by the memory allocator: i.e., these are the treads serving RPC requests in CoRM. For each allocation request, the thread is selected randomly.

Vanilla CoRM. Figure 18 shows the memory usage of the Redis traces under different compaction strategies. In this experiment, CoRM disables compaction for blocks that can store more objects than the ones that can be addressed with the given identifier size. For reference, we include the active memory kept by an ideal compaction strategy and in case no compaction is performed. CoRM introduces up to 4 MiB memory overhead (i.e., for CoRM-16) to store object IDs. The reported data includes this overhead.

Redis workloads do not experience allocation spikes as in the synthetic workload. Therefore, none of the algorithms can significantly reduce the active memory for a single-threaded allocator. However, the traces exhibit high fragmentation due to the low usage of some size classes. Depending on the trace, the difference in active memory between 1-thread and 32-thread allocators ranges from 3x to 12x. This increase of fragmentation is explained by the fact that

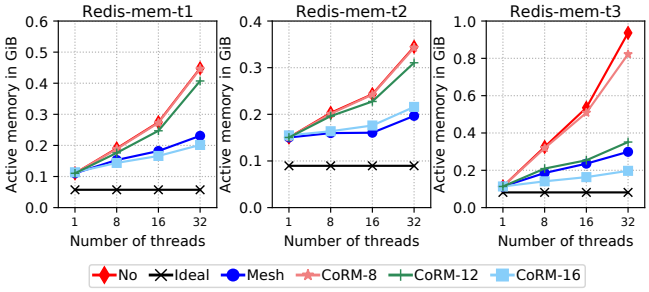

Figure 18: Active memory under Redis workloads. CoRM is configured with 1 MiB blocks.

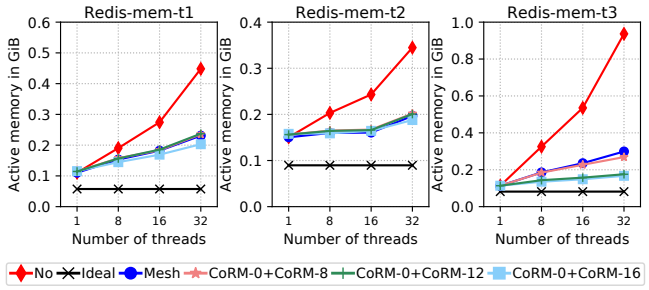

Figure 19: Active memory under Redis workloads. CoRM is configured with 1 MiB blocks and in hybrid mode.

with more threads there is a higher possibility of conflicts (either on the block IDs or on the offsets). The cases where Mesh performs better than CoRM are the ones where CoRM cannot compact blocks because of their large object count: e.g., corm-12 can compact only blocks with objects larger than 256 bytes. With 16-bit per object ID, CoRM-16 provides better memory efficiency for *redis-mem-t1* and *redis-mem-t3*. For *redis-mem-2*, Mesh manages to compact more memory than CoRM-16. Also in this case, this happens because the trace allocates objects that CoRM-16 cannot compact (i.e., 8 bytes, while CoRM-16 can compact blocks with at least 16 bytes objects). CoRM-20, which is not plotted here, manages to compact more memory than Mesh since it supports the aforementioned size class. This suggests that the object IDs should be tuned for particular workloads in order to maximize memory efficiency.

Hybrid CoRM. To remove the negative effects of blocks that cannot be compacted with a given object identifier size, we adopt the hybrid compaction strategy described in §4.4.1. Figure 19 shows the active memory kept by the different compaction strategies with this setting, using the same Redis traces of Figure 18. The data shows that hybrid configuration has better compaction performance compared to pure CoRM and pure Mesh algorithms.

In all experiments, hybrid CoRM is at least as good as Mesh in terms of active memory. For *redis-mem-t1* and *redis-mem-t2*, CoRM-16 provides an improvement of 12% and 5% over Mesh, respectively. The key difference is that CoRM, other than being at least as good as Mesh for memory compaction, it enables RDMA-accelerated DSMs while being able to release and reuse virtual addresses.

Discussion. To take full advantage of CoRM's compaction capabilities, users can tune object ID sizes for different size-classes, according to the specific workloads. Ideally, applications would label class sizes with an indication of how frequently they are used. Highly-used classes would likely not benefit for compaction since their frequent allocations and deallocations would already avoid fragmentation. Instead, class types that are not frequently allocated, can be managed by CoRM which will be able to compact them while introducing a space overhead given by the object IDs. We consider an auto-labeling strategy of class sizes as future work.

5 Related Work

RDMA-accelerated DSM systems: FaRM [14] is a distributed memory computing platform that exploits RDMA to read remote objects. FaRM does not support memory compaction and addresses the problem of unpopular size-classes by pinning them to specific thread allocators. While this solution mitigates memory fragmentation due to unpopular size classes, it does not help to limit fragmentation due to allocation/deallocation spikes.

GAM [9] is a shared memory system that exploits RDMA to accelerate its cache coherence protocol. Unlike FaRM, GAM does not allow objects to be read using RDMA but it used RDMA for updating its shared buffer state. Like FaRM, GAM does not support memory compaction and can benefit from CoRM's compaction algorithm without compromising its RDMA functionalities.

RDMA-enabled systems with compaction: RamCloud [39] and MICA [27] are key-value stores employing log-structured memory allocators to limit fragmentation. The main drawback of this approach is that deleted objects occupy memory until they are garbage-collected. To free a memory block, the garbage collector copies alive objects from it to the tail block of the log-structured memory. As objects frequently move in memory, RamCloud and MICA use indirection tables. Therefore, MICA's RDMA-accelerated extension, HERD [19, 20], and RamCloud cannot directly access objects using one-sided RDMA and focus on accelerating RPC calls.

6 Conclusion

We introduce CoRM, a prototype of a shared memory system that employs RDMA to accelerate remote reads and, at the same time, supports memory compaction to provide high memory efficiency. CoRM's memory accesses are strongly consistent even in the presence of concurrent compaction. CoRM shows the same read throughput of other RDMA-accelerated DSMs like FaRM under normal operations and it can quickly recover indirect pointers created as a result of memory compaction. In case of fragmented memory, CoRM is at least as good as Mesh in compacting memory, while saving up to 2.8x more memory w.r.t. Mesh in cases where allocation/deallocation spikes occur for large objects. The novel compaction algorithm of CoRM, based on object IDs instead of offsets, does not use indirection tables and completely relies on OS virtual address translation. All in all, CoRM fills a gap in RDMA-accelerated shared memory systems by avoiding the need for compromising memory efficiency for performance.

Acknowledgement. We thank Miguel Castro and the anonymous reviewers for their helpful feedback. We gratefully acknowledge support from ETH Zurich. This work was supported by Microsoft Research through its Swiss Joint Research Centre. This project also received funding from the European Research Council under the Horizon 2020 programme grant agreements DAPP, No. 678880.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC 18)*. USENIX Association, USA, 775–787.

[2] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS 19)*. Association for Computing Machinery, New York, NY, USA, 120–126. https://doi.org/10.1145/3317550.3321433

[3] InfiniBand Trade Association et al. 2000. The InfiniBand Architecture Specification. *http://www.infinibandta.org/specs/* (2000).

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 53–64. https://doi.org/10.1145/2318857.2254766

[5] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD 15)*. Association for Computing Machinery, New York, NY, USA, 1463–1475. https://doi.org/10.1145/2723372.2750547

[6] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[7] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[8] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. 2020. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD 20)*. Association for Computing Machinery, New York, NY, USA, 329–344. https://doi.org/10.1145/3318464.3386135

[9] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (July 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[10] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 1–3.

[11] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast In-Memory Transaction Processing Using RDMA and HTM. *ACM Trans. Comput. Syst.* 35, 1, Article 3 (July 2017), 37 pages. https://doi.org/10.1145/3092701

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[13] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2017. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.* 40, 1 (2017), 3–14.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[15] Aleksandar Dragojeviundefined, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP 15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[16] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004).

[17] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI 17)*. USENIX Association, USA, 649–667.

[18] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (April 2019), 49 pages. https://doi.org/10.1145/3302258

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM 14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI 16)*. USENIX Association, USA, 185–201.

[22] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 20)*. Association for Computing Machinery, New York, NY, USA, 201–217. https://doi.org/10.1145/3373376.3378496

[23] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM 18)*. Association for Computing Machinery, New York, NY, USA, 297–312. https://doi.org/10.1145/3230543.3230572

[24] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[25] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP 17)*. Association for Computing Machinery, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[26] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD 16)*. Association for Computing Machinery, New York, NY, USA, 355–370. https://doi.org/10.1145/2882903.2882949

[27] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444.

[28] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 20)*. Association for Computing Machinery, New York, NY, USA, 541–556. https://doi.org/10.1145/3373376.3378525

[29] Linux Programmer's Manual. 2019. memfd_create - create an anonymous file. *http://man7.org/linux/man-pages/man2/memfd_create.2.html* (2019).

[30] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC 13)*. USENIX Association, USA, 103–114.

[31] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[32] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The Case for RAMCloud. *Commun. ACM* 54, 7 (July 2011), 121–130. https://doi.org/10.1145/1965724.1965751

[33] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS 19)*. Association for Computing Machinery, New York, NY, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[34] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. *SIGPLAN Not.* 53, 2 (March 2018), 679–692. https://doi.org/10.1145/3296957.3173203

[35] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC 15)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/2749246.2749267

[36] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 333–346. https://doi.org/10.1145/3314221.3314582

[37] John M Robson. 1977. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.* 20, 3 (1977), 242–244.

[38] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 228–239. https://doi.org/10.14778/2856318.2856319

[39] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA, 1–16. https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble

[40] Salvatore Sanfilippo. 2009. Redis. *http://redis.io* (2009).

[41] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of Temporary Storage in the NodeKernel Architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 767–782. https://www.usenix.org/conference/atc19/presentation/stuedi

[42] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 851–863. https://www.usenix.org/conference/atc18/presentation/taleb

[43] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[44] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[45] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1637–1650. https://doi.org/10.14778/3342263.3342639

[46] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: a PGAS extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1105–1114.

[47] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi

[48] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD 19)*. Association for Computing Machinery, New York, NY, USA, 741–758. https://doi.org/10.1145/3299869.3300081