SwordBox: Accelerating Shared Access in RDMA-based Disaggregated Memory

NSDI ’25 Fall Submission #470

Abstract

Effectively sharing passive remote memory remains an open problem. Despite its promise, experience with commercial RDMA hardware has shown that 1-sided verbs are incapable of managing concurrent updates at scale. Faced with this reality, most prior systems either disallow sharing, retreat to 2-sided verbs—which require co-locating computational resources with remote memory—or propose new RDMA hardware. We present SwordBox, an alternate approach that leverages the *de facto* serialization point in rack-scale disaggregated systems—the top-of-rack switch—to transparently resolve RDMA races in flight. Our P4-based prototype dramatically improves the performance of Clover, an RDMA-based disaggregated key-value store. Under a YCSB-A workload, throughput rises by nearly 35× while bandwidth usage and tail latency drop by 16 and 300×, respectively.

1 Introduction

There has been tremendous interest in resource disaggregation in recent years, with both academic and industrial researchers chasing the potential for increased scalability, power efficiency, and cost savings [30, 7, 12, 13, 16, 24, 32, 38, 39, 41, 44]. By physically separating compute from memory across a network, it is possible to dynamically adjust hardware resource allocations to suit changing workloads. A large number of proposals [15, 2, 43, 47, 49, 11, 1, 45] have leveraged the remote direct memory access (RDMA) support found in modern network interface cards (NICs) due to its low latency, high throughput, and simple verb-based interface. Yet, most share a common drawback: they do not support sharing remote memory across compute nodes. For systems that do [47, 49, 45], even modest levels of write contention crater performance.

The reason behind this limitation is easy to identify: sharing remote memory requires coordinating access across multiple clients, yet the RDMA protocol—like TCP—provides a connection-based abstraction; while connection-less operation is possible, much like UDP it provides essentially no semantic guarantees. Fundamentally, remote memory operations must be ordered to provide coherent access, and the RDMA protocol provides two basic mechanisms to do so remotely (i.e., in a 1-sided fashion): reliable connections that ensure ordering and atomic operations that deliver mutual exclusion. While the performance of RDMA connection handling has received considerable attention [11, 39, 8], connections remain an end-to-end abstraction, and do not provide any guarantees regarding operations from distinct clients. For that, systems must rely on atomic operations like compare-and-swap (CAS), but their enhanced semantics dictate expensive implementation choices on the NIC, dramatically restricting their performance compared to simple verbs like read and write [22]. Moreover, atomic operations are available only over reliable connections.

As a result, most existing systems that deliver scalable, high-performance shared remote access depend on the presence of computational resources collocated with the remote data [21, 34, 11, 33, 39]. In particular, a memory-local CPU can employ 2-sided RDMA operations to orchestrate operations between multiple clients [21, 23], avoiding the need for atomic operations. Unfortunately, such RPC-like approaches are infeasible in the passive memory setting. Alternatively, organizations with significant resources have considered redesigning the RDMA protocol itself to better support the needs of the disaggregated usage case—by, e.g., removing the connection abstraction and providing more powerful verbs [52, 46, 50]—but such hardware is not yet available.

In this work, we explore an alternative dimension: rather than relying exclusively on end-to-end solutions, we consider leveraging in-network resources—specifically programmable switches that are located between clients and the remote memory servers—to accelerate systems based on existing 1-sided RDMA verbs. Concretely, we observe that in rack-scale disaggregated settings, the top-of-rack (ToR) switch serves as a single serialization point for all RDMA requests. As a result, it is possible to transparently rewrite RDMA operations in flight to orchestrate requests from multiple clients to passive memory servers, sidestepping the fundamental bottlenecks present in the current connection-based RDMA protocol.

We present SwordBox, a top-of-rack switch that implements two separate yet complimentary approaches to accelerating RDMA-based passive memory [4]. Client-driven schemes must rely either on mutual exclusion (i.e., locks) or optimistic concurrency control (which require multiple round trips to resolve conflicts). SwordBox removes the performance bottlenecks of both by 1) multiplexing multiple clients' RDMA operations onto shared connections to leverage the ordering semantics delivered by reliable connections [35], and 2) caching small amounts of metadata to dynamically steer in-flight RDMA updates to serialize concurrent operations to remote-memory indexing structures.

We apply SwordBox to two remote memory systems that natively support sharing: Sherman [49], which uses locking, and Clover [47] that relies on optimistic concurrency. We show that both systems natively collapse under contention due to RDMA's limitations, but SwordBox can remove their bottlenecks. Concretely, by multiplexing all acquire and release operations for a shared lock in Sherman onto a single reliable connection, SwordBox can replace the client-issued compare-and-swap operations with a lightweight writes, delivering a potential 10× throughput gain. Performance gains are even higher in the case of Clover, where we resolve update conflicts to Clover's internal, append-only metadata index structure by steering requests to an advancing set of locations, as if they had been issued by a single serialized client. Our evaluation shows that under a 50:50 read-write workload, throughput rises by almost 35× while bandwidth usage and tail latency drop by 16 and 300×.

2 Background

We briefly overview RDMA, use of programmable switches in disaggregated settings, and passive remote memory.

2.1 RDMA protocol

RDMA was designed as a protocol for accessing remote memory. It defines a set of zero-copy instructions—known as *verbs*—that are invoked by clients to read or write memory physically located on remote servers, typically without kernel involvement on either end. When run over Ethernet using the RoCEv2 standard, RDMA NICs located on client and server can cooperate to implement congestion [29, 55] and flow control, reliable delivery, and at-most-once delivery semantics. Before exchanging data, RDMA end points establish a queue pair (QP) which defines the region(s) of memory each is able to access. Like Internet transport protocols, RDMA queue pairs provide a configurable set of semantics depending on the transport mode selected: UDP-like semantics are provided by unreliable datagram (UD) and unreliable connections (UC), while reliable connections (RC) are similar to TCP, ensuring reliable, in-order delivery. Moreover, reliable connections support so-called *1-sided* verbs (e.g., read, write, and compare-and-swap) that are executed autonomously by the remote NIC without any remote CPU involvement.

The benefits of the various transport modes and 1-vs-2-sided verbs has been a topic of intense debate. While reliable connections provide enhanced guarantees, their implementation requires on-NIC memory, a precious resource, and researchers have observed scalability bottlenecks due to memory and cache limitations in the Mellanox ConnectX family of RDMA NICs [11, 23, 20, 48, 22]. Recent work has shown how to overcome limits in terms of the number of connections [39, 35], but the ordering guarantees provided by RC remain restricted to individual queue pairs. While unreliable transport modes can deliver superior performance and scalability [23], they require the use of 2-sided verbs—i.e., involvement of a CPU at the memory server—to ensure ordering, a non-starter for passive disaggregated settings. Unless hardware support for more sophisticated 1-sided verbs [52, 46, 50] becomes available, another approach is required.

2.2 Programmable switches

Most prior proposals for disaggregation consider rack-scale deployments where servers are partitioned into roles: compute, memory, and storage, all of which are interconnected by a top-of-rack switch [3, 12, 17, 24, 43]. The central role of the ToR in this architecture has not gone unnoticed, and researchers have observed that a programmable switch can off-load a wide variety of traditional operating system services [3, 26, 53, 54, 18, 19]. The constraints in each case are similar: programmable switches have limited memory and processing capabilities. If the computational task is too large packets must be recirculated adding additional latency and reducing aggregate bandwidth. Ideal applications for programmable switches use little memory, require minimal processing and deliver outsized performance benefit.

Specifically, prior work has shown that programmable switches are able to provide rack-scale serialization at low cost [27, 28, 40, 4], manage locks [53], and track the state required to maintain an RDMA reliable connection [25]. Most relatedly, researchers have even used a programmable switch to implement a centralized memory controller including a unified TLB and cache for passive remote memory [26]. Their approach is limited, however, by the resource constraints of the switch. Inspired by performance-enhancing TCP proxies of old [5, 14], we consider a slightly different design point where the authoritative state and control logic remain at the end points. In our approach, the ToR simply observes and, at times, selectively modifies RDMA packets [42, 4] in-flight to improve performance while preserving the semantics of the underlying RDMA connections.

2.3 Disaggregated memory

While 1-sided RDMA allows system designers to place memory on passive remote servers, the question of how best to access remote memory remains open; existing systems fall into one of two categories. Transparent systems present far memory to the application as if it were local, using traditional memory virtualization techniques to cache pages locally and swap in and out on demand [36, 15, 32, 2, 43]. Other systems require applications to access remote memory explicitly through RPC-like APIs [41, 1, 47, 49, 23, 45] which allow the runtimes to apply a variety of optimizations including batching and scheduling. Regardless of the method of access, sharing remote memory between clients is fundamentally expensive, and at the time of writing no RDMA-based transparent system exposes shared memory. Rather, the few systems that attempt to support sharing do so in the context of specific datastructures which allows the runtimes to manage concurrent accesses. We apply SwordBox to two distinct contention management schemes—locks and optimistic concurrency control—and provide brief overviews below.

Locks. Sherman is a write-optimized B+Tree for passive remote memory implemented using per-node locks [49]. The tree is augmented using entirely 1-sided RDMA operations. Sherman improves performance under contention in two ways. First, it places locks for each node in the B+Tree in a special region of NIC memory exposed by ConnectX-5 NICs. This small chunk of memory exposes the same RDMA interface as host memory, but allows locking operations to execute with approximately $3\times$ the throughput. Second, Sherman's clients employ a hierarchical locking scheme to reduce the contention for server-hosted locks. This client-local optimization significantly improves performance in cases where clients are collocated; SwordBox seeks to achieve similar efficiencies at the rack scale.

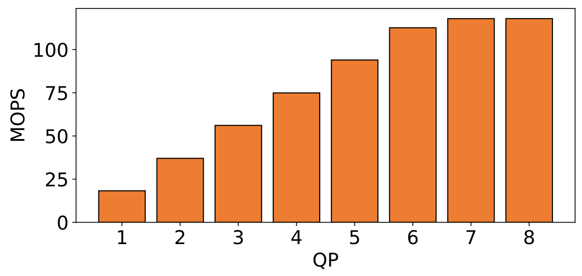


Figure 1: Max throughput as a function of the number of RoCEv2 RC connections. Each queue pair is managed by a separate core and issues in-lined writes.

Optimistic concurrency. Two recent systems support concurrent updates to remote key-value stores through optimistic use of 1-sided RDMA atomic operations and client-driven resolution protocols. In Clover, reads and writes for a given key are made to an append-only linked list stored in (persistent) remote memory [47]; clients race to update the tail of the list. In FUSEE, persistence is implemented through client-driven replication, so clients race to update a majority of replicas in an instance of distributed consensus [45]. In both cases, writes are guarded by CAS operations so clients can independently determine the outcome of the race. Because we are interested in the fundamental costs of contention—as opposed the additional challenge of replication—we focus specifically on Clover in this paper, but SwordBox could equally well apply to a (degenerate) non-replicated instantiation of FUSEE. Indeed, we provide a performance comparision in the evaluation (Section 6).

In Clover, all RDMA requests are targetted at the (presumed) tail of the list and writes are guarded by CAS operations. A client may not know the location of the tail as other writers concurrently push it forward. When an operation fails to land at the tail of the list Clover traverses the structure until the tail is found. While this provides no liveness guarantees, in the common read-heavy case concurrent clients eventually reach the end of the list. To speed up operations clients keep caches of the end of each key's linked list to avoid traversals. By implementing a shared cache at the ToR, SwordBox decreases the likelihood of stale accesses.

3 Serialization

The fundamental challenge faced by passive remote memory systems is ensuring consistency [36] by ordering ac-

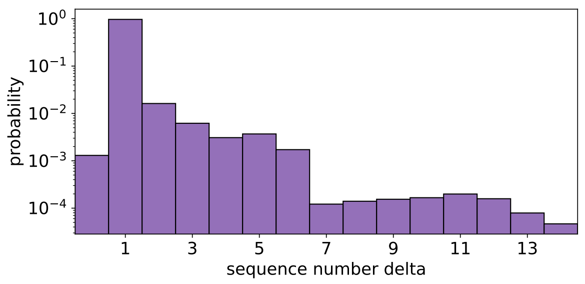


Figure 2: PDF of request reorderings. Retransmitted requests lead to reordering values of zero. 97% of requests retain their order (delta=1), however reorderings of up to 13 requests can occur. (Note logarithmic y axis.)

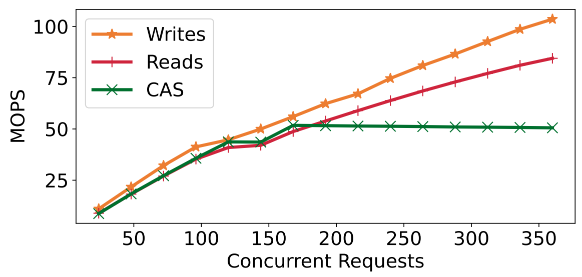


Figure 3: Achieved throughput of RDMA verbs across twenty queue pairs on data-independent addresses as a function of request concurrency. With atomic requests, ConnectX-5 NICs can support approximately 2.7 MOPS per queue pair, and 55 MOPS in aggregate.

cesses to any given location. RDMA reliable connections provide per-connection ordering, enabling clients to issue multiple outstanding requests; the NIC ensures in-order delivery despite packet reordering and drops with sequence numbers and go-back-n retransmission. When clients are collocated, queue pairs can be shared by multiple cores through techniques such as flat combining [35, 49]. Unfortunately Figure 1 shows that the performance of individual queue pairs fall far short of line rate on our NICs; we do not observe full performance until at least seven queue pairs are used simultaneously. Moreover, as traditionally conceived, a QP is intended to be established between a single client and server—limiting its utility in a disaggregated setting. In the following subsections we experimentally illustrate the challenges to ordering across such clients.

3.1 Switch-enforced ordering

Packets processed by a programmable switch pipeline are sequenced in order: updates to switch registers are atomic as each state of a pipeline is occupied by exactly one packet at a time. Moreover, all packets destined to a given port must traverse the same egress pipeline. As a result, the ToR places packets from all flows destined to the same (single-homed) destination in a total order—not only with respect to their own flow, but others as well. In the context of RDMA, however, switch-enforced packet ordering is insufficient. Even if packets (from various reliable connections) arrive at a server NIC in a given order, they may (appear to) be processed in arbitrary order due to contention at the NIC or PCIe bus [37].

Figure 2 shows that NIC and PCIe reordering is not merely an academic concern, but occurs with some frequency. In this example, we issue RDMA read, write and CAS requests at a rate of one million requests per second to 1,024 different memory locations according to a Zipf distribution and spread these requests across 32 different reliable connections. Each request is routed through a programmable switch that keeps a global request counter for each RDMA request (i.e., ground truth regarding request ordering). We track the order of responses relative to the order the corresponding request was issued from the middlebox. The plot shows the distribution of sequence-number gaps between responses. As expected, the vast majority differ by one (i.e., the same order they were dispatched from the switch), but a non-trivial number are out of order by one to five requests, and some by up to 15. Moreover, this experiment neglects the reality that some fames may be corrupted and/or lost by the link, necessitating retransmission and further cross-flow reordering.

3.2 Atomic RDMA operations

While the RDMA protocol does not have a mechanism to enforce cross-flow ordering, it provides atomic verbs that allow applications to implement their own mechanisms to detect races and/or enforce ordering. The 1-sided *Fetch-and-Add* and *Compare-and-Swap* (CAS) operations target 64-bit words in remote memory. Like their traditional CPU counterparts, atomic requests appear to execute at a single point in time, ensuring a total ordering on all data-dependent instructions and a partial ordering between all other instructions—i.e., all other instructions appear to occur strictly before or after the atomic, regardless of whether they are themselves atomic. Remote memory systems can use these atomic operations in two basic ways to enforce consistency: implement shared locks or optimistic concurrency control.

Remote locks In a remote-locking scheme, clients use an atomic RDMA operation to attempt to acquire a lock: because the operations are totally ordered at most one

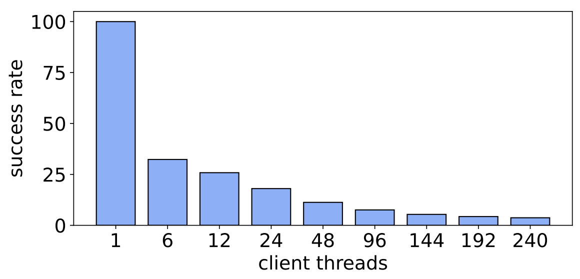


Figure 4: Percentage of successful operations in a 50:50 read-write workload spread across 1,024 keys according to a Zipf(0.99) distribution as a function of thread count. At 240 threads less than 4% of operations succeed.

client will succeed at a time. Unfortunately, atomics are famously expensive [22], fundamentally because they require mutual exclusion across all RDMA queue pairs—concurrent read and write operations with a data dependency on the atomic address must stall until the atomic completes. Figure 3 considers the best-case scenario where clients attempt to access unique locks (i.e., each instruction is issued to an isolated cache line) in remote memory using an atomic operation in comparison to reads and writes. We confirm that the findings of prior studies [22, Fig. 14] with older hardware (i.e., ConnectX-3) remain true on our ConnectX-5 NICs, namely that atomic requests scale with non-atomics only to a point.[1] CAS operations have a hard performance ceiling, while standard verbs (e.g., read and write) continue to scale with increased request concurrency. The situation is even worse when operations target the same address (i.e., lock contention; not shown).

Optimistic concurrency. One way to avoid the overhead of remote lock acquisition in low-load situations is to attempt to directly modify the data (using an atomic RDMA operation) and recover if the operation fails due to a race; such schemes are known as optimistic concurrency control. While far more performant than lock-based approaches in the un-contended case, optimistic approaches can be prohibitively expensive when contention is common. As a concrete example we consider the chances of success in Clover. Figure 4 shows the percentage of requests which succeed in a 50:50 read-write workload as a function of the number of concurrent client threads. Success rate drops dramatically with concurrency.

At present RDMA has no support for addressing failed operations at the server, such as pointer chasing or operation retryi—although some have proposed such extensions [46, 31, 10]. Rather, clients resolve failures themselves at significant cost. In some systems, the retry is a heavyweight, pessimistic operation, leading to a substantial—but fixed—overhead. In others, like Clover and FUSEE, subsequent attempts remain optimistic, resulting in a linear (per-retry) increase in costs. In the latter case, high rates of contention lead to congestion collapse, where retries are essentially doomed to fail, dramatically decreasing goodput.

Concretely, our measurements show that under contention the average bandwidth cost of Clover read and write operations can inflate by 16× (Figure 11) when compared to an optimal scenario in which all operations succeed on their first try. Perhaps even more significant than the overheads associated with the expected number of retries is the cost at the tail—namely the latency associated with those particularly "unlucky" requests that fail repeatedly. Note that these operations are precisely those for hot memory locations, so likely to be ones that matter. Under contention Clover's p99 tail latency increases by over 300× (Figure 12).

3.3 Implications

Systems that leverage RDMA atomics have hard performance limits because the aforementioned constraints. Locks located at a single address which use traditional lock, unlock operations are limited to around 500k accesses per second. This assumes perfectly coordinated requests, under contention requests which fail to acquire or release a lock still consume operation bandwidth. Under contention RDMA has poor support for traditional locking. In contrast optimistic data structures with locks scattered throughout, such as a linked list, are not rate limited by this single address restriction. However, they are fundamentally limited by the fact that any atomics have half the throughput of reads and writes. More critically, under contention optimistic data structures have no liveness guarantees.

4 SwordBox

SwordBox is our general-purpose approach to accelerating RDMA-based applications like shared disaggregated memory that require operation ordering across clients. In this section we explain the functionality SwordBox provides and then apply it to two separate remote memory systems. At a high level, SwordBox is capable of 1) tracking on-going reliable connections, 2) parsing and caching their contents, and 3) modifying operations in flight. Because it defines a total order on outgoing RDMA requests,

[1]Experiments with a ConnectX-6 exhibit similar behavior.

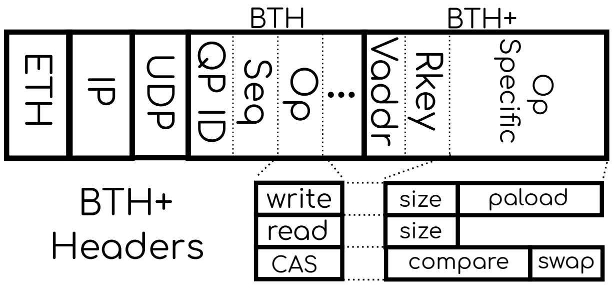
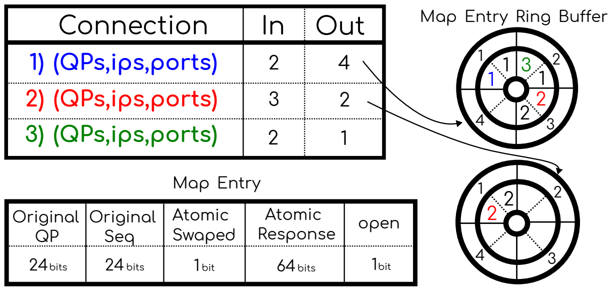


Figure 5: RoCEv2 packets consist of an Ethernet, IP, and UDP header. The RoCE BTH header stores QP data, sequence numbers, flags, and operations. The BTH+ header contains operation specific data: virtual addresses, DMA size, and atomic payloads.



Connection	In	Out
1) (QPs,ips,ports)	2	4
2) (QPs,ips,ports)	3	2
3) (QPs,ips,ports)	2	1

Map Entry

Original QP	Original Seq	Atomic Swaped	Atomic Response	open
24 bits	24 bits	1 bit	64 bits	1 bit

Figure 6: RC multiplexing in SwordBox. Per connection in and out sequence numbers are tracked to decouple sending and receiving QP. Map entries are stored in their outgoing connection's ring buffer; the ring diagram shows only the original QP row and sequence number. In this example three packets are mapped to connection 1, one from each connection. A packet that arrived on connection 2 was not remapped.

SwordBox can safely remap them between different connections as well as transform atomics into lightweight verbs. As we show in the context of Clover, by tracking a small bit of application-specific state, SwordBox can also use its knowledge of operation order to modify the target address or value of conflicting operations to resolve write/write conflicts before they occur.

4.1 Connection multiplexing

RoCEv2 tunnels the original Infiniband-based RDMA protocol on top of UDP, using destination port 4791. RoCEv2 packets have two headers, BTH,and BTH+, shown in Figure 5, both of which SwordBox needs to parse. The BTH header indicates the operation, while the BTH+ headers contains the target virtual address and the operation payload. In cases where SwordBox wishes to enforce ordering across operations from different clients, it multiplexes them onto the same reliable connection. SwordBox does not establish or terminate connections itself—setup and teardown are handled end-to-end as usual by the RDMA NICs. Rather, SwordBox simply moves operations between existing connections.

Connection tracking To facilitate connection multiplexing, SwordBox maintains a table of reliable connections transiting the switch, shown in Figure 6. Connections are uniquely identified by their source and destination IP address, source UDP port (the destination port is fixed for all RoCEv2 traffic), and queue pair ID. Entries are added to the table upon queue pair establishment and removed at teardown (or after a timeout). Each row in the table is associated with a ring buffer of *map entries* that track outstanding operations. As RDMA packets arrive and are placed into a total order at the switch, the row corresponding to the packet's incoming queue pair is updated with the current sequence number. Retransmissions (i.e., packets whose sequence numbers are no greater than the value already in the table) do not update the table. The table also records the highest sequence number used by an outgoing packet on each connection, which may not be the same as the incoming sequence number due to remapping.

Remapping In general, RDMA packets will be forwarded using the same connection on which they arrived. In application-specific cases, however, SwordBox may wish to move them to a different queue pair, i.e., to multiplex them onto a shared connection. In that case, the packet needs to be rewritten to use the new connection's source and destination addresses (both IP and Ethernet), UDP port, and an appropriate sequence number—which is computed to be one higher than the last packet transmitted on the outgoing connection. (Incoming retransmissions—detected due to their non-advancing sequence number—are always mapped to the same outgoing connection and sequence number as the original.) To facilitate ACK and retransmission handling, every packet creates a map entry in the ring buffer of its outgoing connection.

For efficiency, our DPDK implementation maintains the ring buffers as fixed-size arrays and use the packet's outgoing sequence number (modulo the buffer size) as an index. Each entry contains a reference to the packet's incoming queue pair, its original sequence number, and, in the case of atomics, space to record whether the operation was replaced by a write (Section 4.3) and, if so, the prior value of the target address—which is tracked using application-specific logic discussed below. If the packet was remapped, both the invariant CRC (ICRC) and the IP checksum must be updated.

ACK coalescing Demultiplexing ACKs for remapped operations is non-trivial due to optimizations in the RDMA protocol. Specifically, an RDMA NIC may coalesce ACKs to reduce the number of packets transmitted and save bandwidth: like TCP, ACKs are cumulative. Coalescing presents a challenge when operations from multiple incoming connections are multiplexed onto another, as an ACK may correspond to operations issued by more than one client. Forwarding the ACK back to only the client who issued the (last) operation referenced in the ACK will cause the other clients whose operations were implicitly acknowledged by the server to timeout and retransmit. Conversely, forwarding ACKs to clients without outstanding operations could lead to unspecified behavior. Upon receipt of an ACK, SwordBox consults the map entries in the ring buffer for the relevant connection. SwordBox generates a separate ACK for each incoming connection with outstanding packets acknowledged by this ACK, setting the sequence number (and address and queue pair information) according to their map entries.

4.2 State caching

In addition to connection information, SwordBox can also parse RDMA operations to track the current state of memory locations of interest. The particular addresses are obviously application specific, but the mechanism is generic: SwordBox simply needs to apply the operations to its local cache in the same order it transmits the operations to the destination. We find that despite the large amounts of data transferred by passive memory systems, contention is typically localized to a few key addresses, such as those that are used to store locks, indexing datastructures, and other metadata. Moreover, these locations are typically accessed using atomic operations, limiting the data size to eight bytes a piece.

4.3 Atomic replacement

When SwordBox tracks the state of address locations of interest, it necessarily determines outcome of atomic operations. Hence, SwordBox can be configured to multiplex all operations targeting specific addresses to the same connection and replace atomic operations with writes that simply store the outcome of the atomic, whether it be a compare-and-swap or fetch-and-add. Here we describe how SwordBox handles the former without loss of generality.

Replacing a CAS operation with a write is straightforward as the RoCEv2 headers differ by only a few fields (Figure 5). SwordBox transforms CAS requests to writes by swapping the BTH OP code, setting the BTH+ size field of the write to eight (recall all CAS operations are 64-bits long), and copying the appropriate value—either the "swap" value from the CAS operation on success or the current (cached) value on failure—into the payload. SwordBox indicates the operation has been transformed in its map entry (Section 4.1) as well as recording the prior value. Because a write's size field is only four-bytes long (as compared to the second 64-bit compare field in a CAS operation), the length of the packet shrinks by four bytes; SwordBox updates the IP length field accordingly.

After processing the write operation the destination NIC will respond with a regular, write ACK. As part of its ACK processing, SwordBox applies an inverse transformation to convert write ACKs to Atomic ACKs when necessary. RDMA Atomic ACK headers are very similar to regular ACKs with the only difference being that the atomic ACK contains the value that was overwritten, which SwordBox retrieves from the corresponding map entry.

4.4 Disaggregated memory

We now describe how we use SwordBox's techniques to accelerate the contention management techniques used by Sherman and Clover. In the case of Sherman, SwordBox multiplexes all operations (encoded as CAS operations) for a given lock on a single connection, caches lock state at the switch, and replaces acquisition attempts with writes. SwordBox's acceleration of Clover is more lightweight—in fact, entirely soft-state—but even more impactful. By caching a small amount of server state that clients manage through CAS operations, SwordBox is able to adjust these requests to ensure they succeed at the server, thereby avoiding expensive application-level retries for concurrent updates.

4.4.1 Shared locks

Sherman uses CAS operations to implement its node locks. Sherman's locks are simply specific (NIC-hosted) memory locations that store either a one (locked) or zero (free). Hence, lock requests are expressed as `CAS(0,1)`, which fail if the lock is unavailable (i.e., the stored value is currently not 0) or atomically set the value to 1—acquiring the lock—if successful. Unlock operations are the inverse. Presuming communication between the ToR and the memory server is reliable and in-order (as provided by an RDMA reliable connection), it is conceptually straightforward for SwordBox to cache the current value of the lock at the ToR.

In our design, SwordBox multiplexes all operations for a given address (i.e., lock) over the same connection to maintain ordering between the ToR and destination server. It can then use its local cache to determine whether an arriving CAS operation will succeed or fail. (If it does not have the current value at the target address cached, it allows the atomic to pass through unmodified and populates its cache with the response.) Knowing the outcome, SwordBox is free to replace the CAS operation with a lightweight write in flight. When a CAS operation arrives for a lock address, SwordBox replaces it with a write for the specified value. (Releases will always succeed, setting the value to 0, while lock acquisition attempts always leave the value as 1; their success or failure is dictated by the prior state.) When the ACK comes back, SwordBox converts the ACK to an Atomic ACK before forwarding it back over the original connection to the client. Because lock values are always zero or one, it suffices to store a single bit—as opposed to eight bytes—to record the prior value in a lock operation's SwordBox map entry.

Even in the case when the lock is already held (and the acquire attempt is doomed to fail), SwordBox still forwards a write request to the memory server to ensure the client and server agree regarding the total number of RDMA verbs communicated between them. The ACK is replaced with a CAS "failure" so that the sender knows the lock acquisition failed. This is in keeping with SwordBox's performance-enhancing-proxy philosophy: it accelerates, but does not replace, the application's end-to-end semantics. Indeed, one could implement the lock server at the ToR itself [53], but that would require a redesign of the underlying system; our goal is to support selective deployment where SwordBox may not be on-path for all servers, dictating that we do not make any changes to the existing system. Moreover, our approach does not require terminating RDMA connections at the switch, which would require extensive buffering.

In general, the determination of which operations share state is application specific and requires inspecting each packet to extract the relevant pieces of metadata. In the case of Sherman, lock locations can be identified by inspecting CAS requests. Each CAS virtual address corresponds to a node lock in the Sherman B+Tree. While SwordBox must interpose on the full set of queue pairs terminated by a given (set of) server(s), this seems reasonable as the ToR is usually on-path for all servers in disaggregated rack settings.

SwordBox is designed for closed-loop clients. Connection remapping would require large amounts of buffering if clients had many in-flight requests spread across multiple QP. Out-of-order requests would need to be buffered prior to delivering them as out-of-order packet delivery triggers RoCE's go-back-n retransmission protocol. Our aim is to enable rack-scale disaggregation where the total number of cores (clients) is less than $O(1k)$ where the few MB of available switch memory is more than sufficient.

4.4.2 Steering

Unlike Sherman, Clover does not implement locks. Instead, Clover attempts to append to a per-key linked list using atomic operations. Clover detects concurrent updates by breaking writes (i.e., list appends) into two RDMA operations: one write to create a new node, and a CAS operation to update the next pointer of the node at the tail of the list—the latter fails when another node was added concurrently. Concretely, it uses CAS operations to attempt to replace a `NULL` pointer (indicating the end of the list) with a pointer to a new element. To prevent such stale CAS requests from failing, SwordBox maintains a cache of the location of the (next pointer of the) node at the tail of each key's linked list. If a CAS request arrives at SwordBox destined for a stale virtual address (i.e., an address other than the one currently cached for that key), SwordBox *steers* the CAS operation by replacing its target address in the BTH+ header with the cached address. While SwordBox could multiplex these operations on a shared connection to enforce ordering (and replace them with writes), our evaluation shows the probability of reordering with a contending operation on a separate connection after departing the ToR is sufficiently low that the remaining cost of (clients) resolving such failures is minimal.

We implement steering by maintaining a cache of Clover's linked-list datastructures for popular keys. When a Clover packet arrives at the switch, it is parsed and passed to application-specific cache management code that extracts the salient information from the payload. Unfortunately, Clover RDMA CAS requests do not explicitly specify the write operation to which they correspond; SwordBox infers the operation by checking the size of the RDMA request and then extracts the Clover key from the appropriate the location in the packet. The key is used as an index into a lookup table to find the virtual address of the current tail node for that key. Our strategy requires 64 bytes of data per key—the size of an RDMA virtual address.

While write steering suffices to avoid write/write conflicts, concurrent reads face a similar dilemma: Clover reads seek to access the current tail of the linked list, but the address may be stale if they "lose" a race with a concurrent write. To improve performance, SwordBox similarly steers reads to the correct tail address. Unfortunately, unlike writes (which are easy to identify by their use of the CAS operation), Clover reads are simply RDMA read operations for a virtual address and a length. As reads can be for arbitrarily old virtual addresses a naive solution that stored the lineage of each key would effectively require caching the entire contents of Clover's metadata server. Instead, SwordBox hashes the address of each write into an array somewhat larger than the size of the key space and stores the key along with the address. Collisions are resolved by replacing the old entry; keys with higher update rates maintain longer histories.

When reads arrive SwordBox looks up their destination address in the table; if the address has a hit the associated key is used to look up the current tail in the write cache and the RDMA read is steered to the cached location. Should a miss occur—either because the hash bucket was overwritten by another key, or because the tail address is not cached—the read is left unmodified. If it fails to arrive at the current tail, Clover's end-to-end recovery mechanism kicks in.

4.5 Failure handling

SwordBox collocates functionality—and therefore shares fate—with the top-of-rack switch: if SwordBox fails, connectivity was already disrupted (i.e., the ToR is down). Hence, the fact that a SwordBox failure will reset all remapped RDMA queue pairs to the attached servers seems of little additional consequence. In the case of steering, however, we note that SwordBox does not maintain any hard state: failure simply results in a performance hiccup if packet-level connectivity can be maintained. The upshot is that a complete SwordBox failure does not introduce safety concerns in any event.

However, there are other failure scenarios to consider. In particular, we presume that the ToR sees the exact stream of packets that will be received—and processed—by attached servers. Unfortunately, this may not be true due to packet loss (e.g., due to CRC failures or queue overflow) or even bugs on the server. Of course, these failure cases exist even without SwordBox, and Sherman and Clover both provide their own error handling. The key distinction, however, is that SwordBox maintains a cache that may become inconsistent with an attached server, which was previously the single authority of both application and connection state.

With respect to connection mapping, if a packet is dropped between SwordBox and a server and SwordBox maps a subsequent request from a different client onto the same QP, the server will generate a go-back-n response and any other in-flight requests on that QP will become invalidated. Hence, when SwordBox sees a go-back-n ACK, it triggers the same mechanism used for ACK coalescing but in reverse: it broadcasts a go-back-n ACK to all clients with outstanding messages. While this approach amplifies the performance impact of a lost packet, we expect such scenarios to be unlikely in practice. Indeed, no packet drops ever occurred between SwordBox and a server during our experiments because our clients issue only closed-loop operations.

While we do not employ connection mapping in Clover, SwordBox must still manage potential inconsistency between its cache and server state. Concretely, it is possible for a linked list to become "broken". If SwordBox sees a client issue a `CAS(A,B)` request (attempting to append node B to the list at A) before another issues `CAS(A,C)` (appending node C to the same—stale—tail), SwordBox will steer `CAS(A,C)` to `CAS(B,C)`. If the `CAS(A,B)` operation is lost between SwordBox and the server, `CAS(B,C)` will still succeed, causing a broken chain: the pointer $A \rightarrow B$ does not exist but $B \rightarrow C$ does, and SwordBox believes C to be the tail.

In the normal case, the client will timeout and retransmit `CAS(A,B)`, which SwordBox will identify as a retransmission and *not* steer to the "new" tail, thereby repairing the list. (In the mean time, the missing link is immaterial because subsequent requests are being steered by SwordBox.) If, however, the client were to fail prior to retransmitting the CAS the chain will remain broken. Here we use an out-of-band mechanism to repair the chain: on occasion our control plane queries the switch to check for outstanding CAS requests and simply retransmits them (spurious retransmissions are handled gracefully by the server). The trickiest case is if SwordBox itself also fails in the mean time: we defer protecting against this double-failure scenario to future work.

5 Implementation

We implement SwordBox in DPDK and P4. Our DPDK SwordBox implementation (shown in Figure 7) consists of 3,392 lines of C and includes all of the features described in the previous section—including ICRC recalculation—but is limited by single-core CPU performance. Our P4 prototype has more limited functionality, but operates at 100-Gbps line rate.

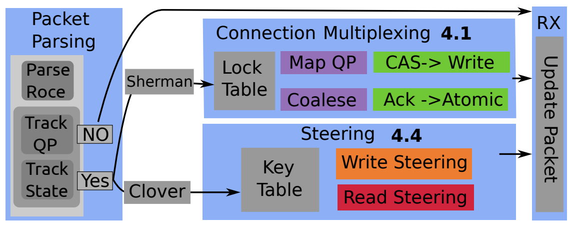


Figure 7: SwordBox's DPDK processing pipeline.

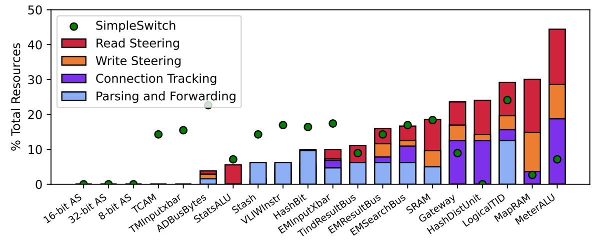


Figure 8: Breakdown of switch resource utilization by SwordBox component.

5.1 Connection steering

Our P4 prototype implements connection steering (§4.4.2) by using registers to store connection state, virtual addresses, and outstanding requests. Switch registers are constrained to 32, 16 and 8-bit blocks, and are bound to specific switch pipeline stages [19]. Packets visit each stage exactly once, so register reads and writes must be pipelined correctly so that the same stage which stores a virtual address on a write, is the same that produces the address for CAS and read. Because registers are fixed width, some lookups take multiple stages. We use two-stage lookups for (64-bit) virtual addresses with two 32-bit registers, and a single stage for queue pairs, sequence numbers, and connection IDs. Prior work has demonstrated that RDMA ICRC's can be implemented in a P4 switch, but are redundant with the Ethernet CRC [6, 51]. Hence, like previous authors [42], we disable ICRC checks at sender and receiver NICs and do not update them at the switch.

Figure 8 provides a breakdown of the resource consumption of our P4 SwordBox implementation as reported by the Barefoot SDE version 9.7.0. Each percentage is the average value across the total 16 switch pipeline stages. SwordBox fits into 8 stages, and is run entirely on the ingress pipeline. We use the header parser from the P4 simple switch to parse up to the UDP header and create our own header parser for RoCEv2 and Clover headers. SwordBox uses RoCEv2 header, write, and CAS payload information to identify traffic for steering. When new Clover traffic is identified the connection is added to the connection tracker.

5.2 Connection multiplexing

While straightforward to implement in DPDK, our P4 prototype currently does not support connection multiplexing (and, hence, atomic replacement) due to the challenge of supporting ACK coalescing. In order to determine to which clients to return an ACK, a variable number—up to the number of clients—of entries must be matched against every packet. Yet, each stage of a P4 pipeline holds unique data and supports only a single lookup. Replicating entries across stages would allow for multiple lookups per packet but a server can coalesce an arbitrary number of ACKs so no fixed number of duplications suffice (and we frequently observe coalescing of 10 or more requests). Recirculation is another alternative, but inflates bandwidth usage in the common case and causes responses to be delivered in reverse order. One obvious alternative is to disabling ACK coalescing at the server NIC, but we are unaware of a way to do so on Mellanox NICs.

6 Evaluation

We use our DPDK implementation to perform a micro-benchmark where we explicitly manage RDMA connections to remove the atomic operations used by Sherman's locking mechanism. We use the P4 implementation installed on a programmable switch to show the impact of in-flight conflict resolution at rack scale in Clover.

6.1 Testbed

Our testbed consists of a rack of nine identical machines equipped with two Intel Xeon E5-2640 CPUs and 256 GB of main memory evenly spread across the NUMA domains. Each server is equipped with an NVIDIA Mellanox ConnectX-5 100-Gbps NIC installed in a 16x PCIe slot and connected to a 100-Gbps ToR. Our DPDK-based micro-benchmarks use only three machines: a load generator, a memory server, and a machine hosting our DPDK implementation of SwordBox. The load generator is configured with default routing settings—it sends traffic directly to the memory server. We install OpenFlow rules on a Mellanox Onyx switch to redirect the traffic to the DPDK box. For the P4-based Clover experiments, we replace the Onyx switch with an Edgecore Wedge-100 programmable switch running SwordBox. We configure one server as a Clover memory server, one as a metadata server, and the remaining seven as Clover clients.

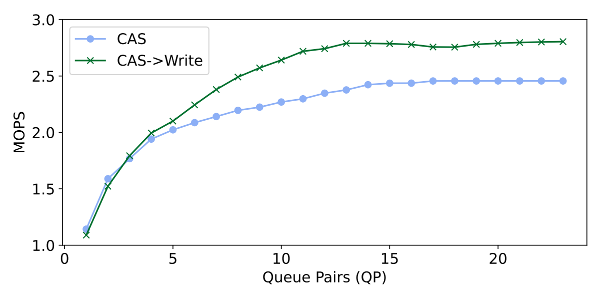


Figure 9: Throughput of conflicting CAS and rewritten CAS requests as a function of client threads/QPs.

6.2 Atomic replacement

We show that SwordBox is able to overcome the NIC hardware bottleneck by replacing CAS operations with writes serialized on a given RC by running a micro-benchmark that focuses exclusively on CAS performance. Specifically, we extract the CAS request from Sherman's lock operation and repeatedly generate it from one client to a single memory server (while routing it through SwordBox using OpenFlow rules). Each client thread is bound to its own queue pair, and all client threads issue CAS requests to the same shared virtual address. We set the number of cores on the SwordBox middlebox to 24 so that in our maximal test case each client thread flows through exactly one middlebox core for the lowest degree of interference between QP.

Figure 9 shows the results when all requests are directed at the same address in the remote server's main memory. In the default case (labeled CAS in blue), SwordBox lets CAS requests flow through without modification, each on their own queue pair. In the CAS$\rightarrow$Write (green) configuration SwordBox maps all client requests to the same QP at the server to ensure serialization and replaces the CAS operation with a simple write. We see a significant increase in performance when SwordBox converts CAS-guarded requests to QP-serialized writes. Each configuration hits a distinct hardware limit: CAS requests bottleneck at the server NIC due to being applied to a single key (c.f. Figure 3). When converting CAS to serialized write operations, the bottleneck moves to the DPDK middlebox. Specifically, DPDK requires all TX for a destination QP to be done by the same core; hence, all requests must flow through a single core, capping the performance of our DPDK implementation to the maximum per-core throughput of our middlebox server: 2.8 MOPS.

6.3 Steering in Clover

While atomic replacement is feasible, it requires SwordBox to explicitly manage and remap all the RDMA connections to a given (set of) server(s)—a resource-intensive task. Here, we consider the more general and lightweight case where SwordBox serves as a performance-enhancing proxy and attempts to avoid failed operations by steering requests in flight. We use workloads from the YCSB benchmark [9] to access 1,024 128-byte objects stored in Clover. (Results for a range of sizes are presented in Appendix A.1.)

6.3.1 Throughput

Figure 10 shows the impact of SwordBox's techniques at various levels of contention. A read-only workload exhibits no contention, so SwordBox simply passes through all operations unmodified achieving a maximum throughput of approximately 40 million operations per second in our testbed. As a point of comparison, we also plot (in green) the performance of a non-replicated instance of FUSEE, in which case their SNAPSHOT consensus algorithm degenerates to a lock-based approach. While FUSEE's absolute read throughput on our testbed is considerably higher than reported by the original authors on their own hardware, it is less than half that of Clover on this workload. While Clover clients can safely cache the linked-list location for popular keys (because any updates will cause the next pointer of the returned element to be non-NULL), FUSEE clients must always issue two seperate, dependant RDMA reads: one to obtain the current location for the desired key, and then one to read the value.[2]

Clover (shown in blue) performance decreases markedly with even 5% writes, nearly matching FUSEE; write steering alone (orange) provides minimal performance improvement as the vast majority of writes succeed on their first try—it is the reads that are failing. Steering both reads and writes (red) restores performance, although to a slightly lower overall throughput as even successful Clover writes require two RDMA operations instead of one. At 50% writes, over half of all write requests fail so applying write steering almost doubles performance. The steered writes, however, then out-pace reads causing the majority of reads to fail unless SwordBox also applies read steering. (The impact on tail latency is clearly shown in Figure 12.) Of course, in a

[2]While the results in the FUSEE paper suggest it outperforms Clover [45, Figs. 13–15], Clover's client cache is disabled in those experiments, forcing all reads to go through the metadata server. Moreover, in our experiments, FUSEE fails to scale beyond 256 clients—published results only go to 128 [45].

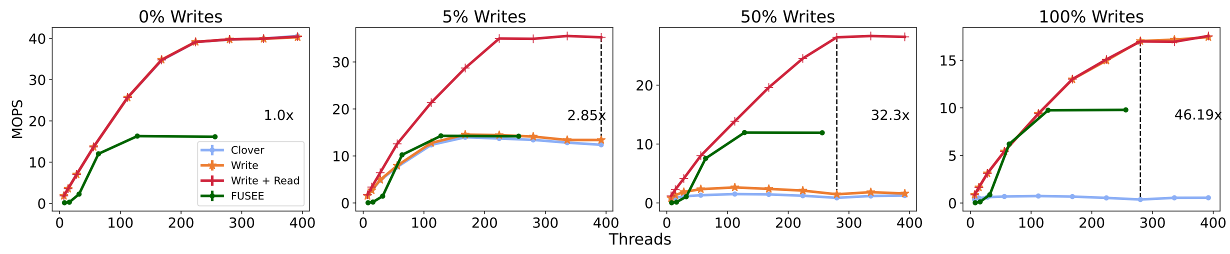


Figure 10: Steering applied to Clover with 128-byte objects across 4 YCSB benchmarks. The percentage of workload writes increases from left to right. SwordBox throughput relative to Clover is 1.0, 2.8, 32, and 46× respectively.

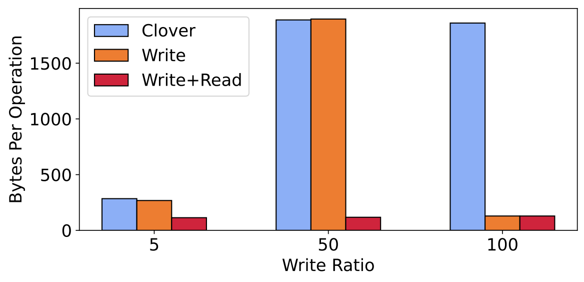


Figure 11: Average number of bytes required per Clover operation on 128-byte objects using each of the three techniques at various write intensities.

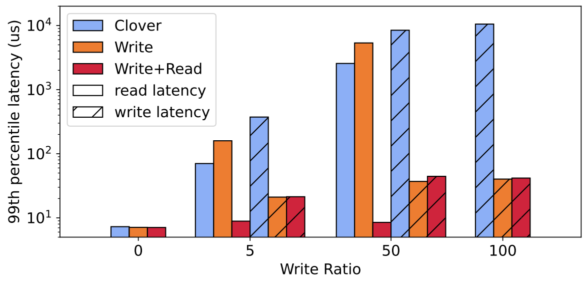


Figure 12: 99th-percentile tail latencies of read (solid) and write (striped) Clover operations at various write intensities. (Note logarithmic y axis.)

100% write workload write steering alone is sufficient. While FUSEE suffers less from increased contention, its writes require three or more RDMA operations; as a result SwordBox pushes Clover to achieve 1.9–2.5× higher throughput than FUSEE.

6.3.2 Bandwidth reduction

Under contention, Clover's remote operations can require additional packet exchanges which inflate the bandwidth necessary to service the same number of memory accesses. SwordBox's steering algorithms remove the need for requests to retry, eliminating the overhead. Figure 11 plots the average bytes per operation for each strategy across the three workloads with writes. (The read-only workload, not shown, never needs to retry.) We calculate the value for each technique by summing the total bandwidth across a run and dividing by the number of operations. Clover's bandwidth usage increases with contention, growing by 2.5× at 5% and 16× at 50% writes—all of which is recovered by applying read and write steering. Write steering alone causes significant inflation in the cost of operations at 50% writes because many read requests fail as discussed above.

6.3.3 Tail latency

Optimistic concurrency is well known to exhibit poor tail latency under contention, and Clover is no exception. SwordBox significantly reduces latency as steering ensures that nearly all requests succeed on the first try. Figure 12 shows the 99th-percentile tail latencies associated with SwordBox's read and write steering in comparison to default Clover at varying write intensities. Clover's p99 read latency (solid blue) at 5% writes is 70 μs, around 10× its baseline our our testbed. With read and write steering (solid red) the read tail latency drops to 8 μs—a 8× improvement over Clover even in this low-contention regime. At 50% writes the performance increase from steering increases dramatically: p99 read latency drops by over 300×. Writes (hashed) have slightly more than double the latency of reads as they require two round trips and atomics are slower to execute than other operations. Combined write and read steering provides a 17, 189, and 252× improvement in write tail latency, respectively, across 5, 50, and 100% write workloads. As one might expect, performing write steering alone privileges writes over reads, dropping their tail latencies slightly further—at the cost of a dramatic spike in read tail latency.

7 Conclusion

We leverage the top-of-rack switch to provide cross-client ordering semantics unavailable in today's RDMA standard, dramatically increasing performance of systems that rely on atomic operations and optimistic concurrency. Concretely, we show that SwordBox can resolve in-network contention and enable efficient sharing of remote memory on commodity RDMA NICs. for passive remote memory and show that by resolving conflicts in flight,

Our full-featured DPDK prototype demonstrates the potential of removing atomic operations and relying entirely upon queue-pair ordering guarantees. While connection multiplexing is currently gated by the single-core performance of DPDK, our P4 steering implementation shows order-of-magnitude gains in the context of one of the highest-performing systems available.

References

[1] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., NOVAKOVIĆ, S., RAMANATHAN, A., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 775–787.

[2] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[3] ANGEL, S., NANAVATI, M., AND SEN, S. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)* (July 2020), USENIX Association.

[4] ANONYMOUS. Anonymized workshop paper.

[5] BALAKRISHNAN, H., SESHAN, S., AMIR, E., AND KATZ, R. H. Improving tcp/ip performance over wireless networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 1995), MobiCom '95, Association for Computing Machinery, p. 2–11.

[6] BELTMAN, R., KNOSSEN, S., HILL, J., AND GROSSO, P. Using p4 and rdma to collect telemetry data. *2020 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)* (2020), 1–9.

[7] CALCIU, I., IMRAN, M. T., PUDDU, I., KASHYAP, S., MARUF, H. A., MUTLU, O., AND KOLLI, A. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, p. 79–92.

[8] CHEN, Y., LU, Y., AND SHU, J. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[9] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, Association for Computing Machinery, p. 143–154.

[10] DHARANIPRAGADA, S., JOYNER, S., BURKE, M., SZEKERES, A., NELSON, J., ZHANG, I., AND PORTS, D. R. K. Prism: Rethinking the rdma interface for distributed systems. In *SOSP 2021* (October 2021).

[11] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[12] FARABOSCHI, P., KEETON, K., MARSLAND, T., AND MILOJICIC, D. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, 2015), USENIX Association.

[13] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 249–264.

[14] GRINER, J., BORDER, J., KOJO, M., SHELBY, Z. D., AND MONTENEGRO, G. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[15] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 649–667.

[16] GUO, Z., SHAN, Y., LUO, X., HUANG, Y., AND ZHANG, Y. Clio: A hardware-software co-designed disaggregated memory system, 2021.

[17] INTEL. Intel rack scale architecture: Faster service delivery and lower TCO. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[18] JIN, X., LI, X., ZHANG, H., FOSTER, N., LEE, J., SOULÉ, R., KIM, C., AND STOICA, I. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 35–49.

[19] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 121–136.

[20] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 1–16.

[21] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 295–306.

[22] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 437–450.

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.

[24] KARANDIKAR, S., OU, A., AMID, A., MAO, H., KATZ, R., NIKOLIĆ, B., AND ASANOVIĆ, K. FirePerf: FPGA-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 715–731.

[25] KIM, D., LIU, Z., ZHU, Y., KIM, C., LEE, J., SEKAR, V., AND SESHAN, S. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 90–106.

[26] LEE, S.-S., YU, Y., TANG, Y., KHANDELWAL, A., ZHONG, L., AND BHATTACHARJEE, A. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 488–504.

[27] LI, J., MICHAEL, E., AND PORTS, D. R. K. Eris: Coordination-free consistent transactions using network multi-sequencing. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (October 2017).

[28] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. K. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 467–483.

[29] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., AND YU, M. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2019), SIGCOMM '19, Association for Computing Machinery, p. 44–58.

[30] LIM, K., CHANG, J., MUDGE, T., RANGANATHAN, P., REINHARDT, S. K., AND WENISCH, T. F. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, Association for Computing Machinery, p. 267–278.

[31] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RYAN, M., RUBOW, E., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles* (New York, NY, USA, 2019).

[32] MARUF, H. A., AND CHOWDHURY, M. Effectively prefetching remote memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 843–857.

[33] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 103–114.

[34] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing CPU and network in the cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 451–464.

[35] MONGA, S. K., KASHYAP, S., AND MIN, C. Birds of a feather flock together: Scaling rdma rpcs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 212–227.

[36] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)* (Boston, MA, Dec. 2002), USENIX Association.

[37] NEUGEBAUER, R., ANTICHI, G., ZAZO, J. F., AUDZEVICH, Y., LÓPEZ-BUEDO, S., AND MOORE, A. W. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 327–341.

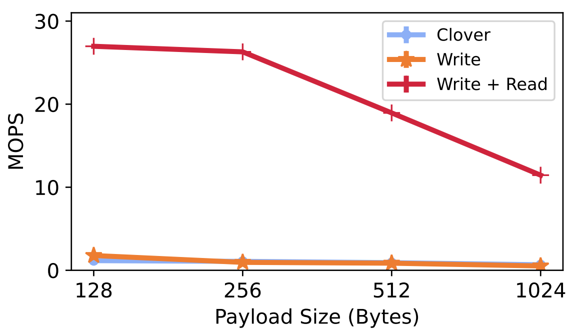


Figure 13: Performance across RDMA payload sizes using 400 client cores at a 50:50 read/write ratio.

[38] NITU, V., TEABE, B., TCHANA, A., ISCI, C., AND HAGIMONT, D. Welcome to Zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 16:1–16:12.

[39] NOVAKOVIC, S., SHAN, Y., KOLLI, A., CUI, M., ZHANG, Y., ERAN, H., PISMENNY, B., LISS, L., WEI, M., TSAFRIR, D., AND AGUILERA, M. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (New York, NY, USA, 2019), SYSTOR '19, Association for Computing Machinery, p. 97–108.

[40] PORTS, D. R. K., AND NELSON, J. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2019), HotOS '19, Association for Computing Machinery, p. 209–215.

[41] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 315–332.

[42] SAPIO, A., CANINI, M., HO, C.-Y., NELSON, J., KALNIS, P., KIM, C., KRISHNAMURTHY, A., MOSHREF, M., PORTS, D., AND RICHTARIK, P. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 785–808.

[43] SHAN, Y., HUANG, Y., CHEN, Y., AND ZHANG, Y. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, 2018), USENIX Association, pp. 69–87.

[44] SHAN, Y., LIN, W., KOSTA, R., KRISHNAMURTHY, A., AND ZHANG, Y. Disaggregating and consolidating network functionalities. arXiv, 2021.

[45] SHEN, J., ZUO, P., LUO, X., YANG, T., SU, Y., ZHOU, Y., AND LYU, M. R. FUSEE: A fully Memory-Disaggregated Key-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association, pp. 81–98.

[46] SINGHVI, A., AKELLA, A., GIBSON, D., WENISCH, T. F., WONG-CHAN, M., CLARK, S., MARTIN, M. M. K., MCLAREN, M., CHANDRA, P., CAUBLE, R., WASSEL, H. M. G., MONTAZERI, B., SABATO, S. L., SCHERPELZ, J., AND VAHDAT, A. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 708–721.

[47] TSAI, S.-Y., SHAN, Y., AND ZHANG, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX Annual Technical Conference* (July 2020), pp. 33–48.

[48] TSAI, S.-Y., AND ZHANG, Y. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, 2017), Association for Computing Machinery, p. 306–324.

[49] WANG, Q., LU, Y., AND SHU, J. Sherman: A write-optimized distributed B+Tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, 2022), Association for Computing Machinery, p. 1033–1048.

[50] WANG, X., CHEN, G., YIN, X., DAI, H., LI, B., FU, B., AND TAN, K. StaR: Breaking the scalability limit for RDMA. In *Proceedings of the 29th IEEE International Conference on Network Protocols (ICNP)* (2021).

[51] XING, J., HSU, K.-F., QIU, Y., YANG, Z., LIU, H., AND CHEN, A. Bedrock: Programmable network support for secure RDMA systems. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 2585–2600.

[52] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. FileMR: Rethinking RDMA networking for scalable persistent memory. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Feb. 2020), pp. 111–125.

[53] YU, Z., ZHANG, Y., BRAVERMAN, V., CHOWDHURY, M., AND JIN, X. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 126–138.

[54] ZHANG, W., WOOD, T., AND HWANG, J. NetKV: Scalable, self-managing, load balancing as a network function. In *2016 IEEE International Conference on Autonomic Computing (ICAC)* (2016), pp. 5–14.

[55] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale RDMA deployments. *SIGCOMM Comput. Commun. Rev. 45*, 4 (Aug. 2015), 523–536.

A Appendix

Here we provide additional evaluation results.

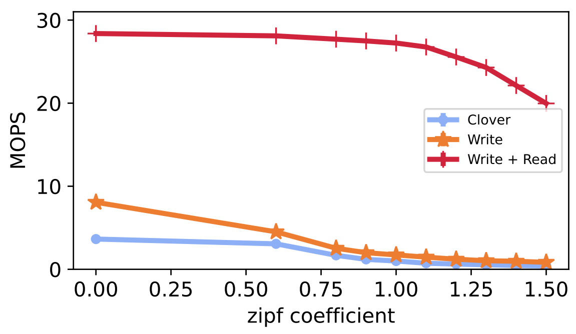


Figure 14: Performance as a function of Zipf coefficient using 400 client cores on a 50% write workload with 128-byte objects.

A.1 Packet size

SwordBox is designed to run at line rate, and is unperturbed by packet size. In the case of Clover, its retries incur a significant bandwidth overhead which causes additional slowdown when payloads are large. Figure 13 shows how SwordBox reacts to changes in packet sizes. At 128 bytes the limitation is the load applied by clients. At 256 bytes and above the 100-Gbps limit of the ConnectX-5 NICs becomes the bottleneck for read and write steering, and the throughput drops proportionally. Clover retries on larger packets are more expensive than on smaller packets because the retry consumes additional bandwidth on an already saturated link.

A.2 Contention

SwordBox is designed to operate well under even extreme contention. We generate workloads at different points across a Zipf distribution. The community standard for Zipf is 0.99; at this ratio the most frequently requested key is requested around 17% of the time. We measure further down the distribution up to Zipf of 1.5, at which point the hottest key is requested over 50% of the time, and the second hottest is around 20%. Figure 14 shows that in the face of high contention 1.0 and above write and read steering yields a 40× and above performance improvement at a 50% write workload. The decrease in read and write performance at high contention is due to Clover's block allocator which becomes a bottleneck with very hot keys.