SwordBox: Sharing Disaggregated Memory without Skewering Performance

NSDI '23 Submission #1063

Abstract

Effectively sharing passive remote memory remains an open problem. While emerging standards like CXL promise cache-coherent memory pooling, spec-compliant hardware is not yet commercially available and its feasibility at scale remains unproven. Existing RDMA-based systems employ optimistic concurrency approaches that defer serialization to the remote memory server and rely upon heavyweight RDMA atomics to ensure consistency. Unfortunately, atomic operations scale poorly causing these approaches to degrade rapidly under contention.

We present SwordBox, an approach to leveraging the *de facto* serialization point in rack-scale disaggregated systems—the top-of-rack switch—to transparently resolve data races in flight. In cases where SwordBox has sufficient resources to interpose on all requests, it can remove heavyweight atomic operations and replace them with simple verbs, avoiding the hardware performance bottleneck. More generally, however, it can safely operate on only a subset of the requests, dynamically adjusting contending requests to avoid expensive client-based resolution in those cases. Under a YCSB-A workload, our P4-based prototype dramatically improves the performance of Clover, a state-of-the-art disaggregated key-value store: 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 [2, 5, 11, 15, 18, 25, 27, 29, 32, 34, 37, 39, 40]. By physically separating compute from storage across a network, it is possible to dynamically adjust hardware resource allocations to suit changing workloads. Considerable headway has been made at higher levels of the storage hierarchy; published and even production systems support remoting spinning disks, SSDs, and modern non-volatile memory technologies [31]. Remote primary storage—a.k.a. memory pooling—remains a fundamental challenge, however, due to the orders-of-magnitude disparity between mainboard access latency and even intra-rack round trips.

The hardware community has made great strides in closing the latency gap via novel technologies like silicon photonics and new rack-scale interconnects, but commercially available options remain significantly slower than on-board alternatives. Concretely, while industrial consortia have proposed cache-coherent memory technologies [8, 16] that would dramatically lower access latencies, currently available interconnects based on RDMA [19] remain on the order of 20× slower than a local access (e.g., 50 ns local versus 1 μs remote). As a result, despite the fact that current-generation memory transport technologies provide the ability to directly execute requests like read, write, and compare-and swap-on remote host memory through the use of RDMA-capable NICs [35], SoCs [6], FPGA SmartNICs [13, 26], or DPUs [14], most existing systems coordinate with a remote CPU on the socket at which the DMA is being performed to assist with serialization [22, 23, 33, 34, 41].

In the absence of a general-purpose CPU located alongside remote memory, it falls to each individual client to ensure that its reads and writes are serialized, usually by leveraging expensive hardware-provided atomic operations at the server like compare-and-swap (CAS) [24] as the latencies involved in client-side coordination are prohibitive. As a result, most existing systems simply partition memory completely and forgo sharing [1, 2, 39]. The few published systems that do support shared access mediate requests to specialized data structures [42, 44].

For example, Sherman, a write-optimized B+Tree [44] places its locks in NIC memory at the server to avoid crossing the remote PCIe bus. Clover [42] implements a hash table that supports lock-less reads; concurrent writes are supported through a client-driven optimistic concurrency protocol. Despite their clever designs, however, both approaches simply delay the inevitable: Sherman's NIC-based locks are subject to significant hardware limits imposed by the CAS operation required to enforce serialization. Similarly, Clover's client-based recovery scheme quickly becomes cost prohibitive when faced with non-trivial levels of write contention.

We argue that these shortcomings are not unique to the particular systems, but rather fundamental to any approach that implements distributed conflict resolution. The obvious alternative is to deploy a centralized memory controller (e.g., a CXL 2.0 switch) that can serve as a serialization point and ensure all races are resolved before accessing memory, but effective realization of such a design has proven elusive. While many proposals exist, none of them have yet been implemented in commercially available hardware. More to the point, such designs are inherently unscalable as they require all accesses to be managed by the controller, rather than forwarded directly between the client and relevant server.

In this work we make the observation that such a serialization point already exists in today's rack-scale disaggregated deployments: the top-of-rack switch. We propose to leverage the capabilities of modern programmable switches to cache sufficient information about in-flight requests to transparently detect and resolve conflicts before they occur. Unlike a centralized memory controller, however, our serializer does not need to operate on—or even maintain state for—all remote memory requests. First, it only needs to address actual conflicts and can avoid the unnecessary costs of enforcing ordering among unrelated requests. Second, in deployment scenarios where it may lack the resources to track all requests, our serializer can serve as a performance-enhancing proxy: when deployed alongside client-based conflict resolution techniques, it can allow even conflicting requests to pass through unmodified without jeopardizing safety while decreasing the frequency of conflict resolution.

We present SwordBox, an on-path serializer that dramatically improves the performance of remote memory systems that support write sharing. Like all ToRs, SwordBox imposes a globally observable total order on memory requests (i.e., packets) within a rack. In scenarios where SwordBox has the resources to explicitly manage RDMA connections, it can enforce per-server ordering at the ToR and remove the expensive CAS operations from all in-flight packets, avoiding their associated performance bottlenecks entirely. More generally, however, SwordBox can be deployed alongside an underlying optimistic concurrency scheme: remote memory operations remain guarded to ensure that clients can detect and recover from conflicts of which SwordBox may not be aware. Instead, because SwordBox understands the disaggregated memory protocol, it can keep a cache of recent operations to adjust subsequent requests whose guards it knows are doomed to fail. In such cases, it modifies requests in flight to account for the preceding operations and decrease the likelihood the guard will trip.

We prototype SwordBox in two scenarios using a rack of servers equipped with ConnectX-5 RoCE-enabled NICs. First, we use a DPDK-based implementation to replace CAS requests in flight with standard RDMA verbs, allowing systems like Sherman to overcome the hardware limit on atomic requests per queue pair. Second, we implement a lightweight version of SwordBox on a P4 programmable switch to accelerate Clover's optimistic concurrency protocol. Our evaluation shows that SwordBox dramatically increases the performance of Clover in the presence of write contention: 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 and related work

The literature on disaggregated memory, efficient use of RDMA verbs, and programmable middleboxes are each too vast to detail here. Rather, we briefly highlight relevant core themes from each and provide a short overview of Sherman and Clover, the two example systems we use to demonstrate SwordBox's optimizations.

2.1 Resource disaggregation

Within the disaggregation community there is a divide between software and hardware-based approaches to remote memory management. The use of dedicated hardware to access and control memory over the network is often referred to as memory disaggregation or pooling, in which custom hardware is used to interpose on either the caching or paging system [4, 5]. Alternatively, approaches that rely entirely on commodity hardware to manage remote memory are generally referred to as far memory [1, 2, 39, 42, 43]. We concentrate on the latter as they are far more likely to see deployment in the short term; our techniques presume only standards-compliant RDMA NICs and are designed to leverage the scalability of today's programmable switching hardware.

Choosing how to expose remote memory to applications remains an open problem. Full transparency allows existing applications to use remote memory without modification. Many systems leverage virtual memory and use remote memory as a swap device, fetching and evicting pages to and from remote memory [2, 12, 17, 29, 36]. Transparency comes at a cost, however, and systems that make remote accesses explicit typically have higher performance for similar operations [37]. Most existing transparent remote memory systems do not support sharing, as unmanaged contention for resources such as shared locks can have disastrous performance implications.

2.2 RDMA-optimized data structures

Given the challenges inherent in full-blown remote memory, many have focused on more constrained, but broadly applicable storage abstractions like key-value storage. RDMA has been used in a litany of work to build fast in-memory key-value stores [9, 10, 23, 30, 33, 34]. In their quest for high performance, these works provide deep insights into the trade-offs between different RDMA verbs and the spectrum of remote-memory data structures. In most, however, it is assumed that a remote CPU is co-resident with memory to provide some degree of serialization for writes and metadata manipulation. Yet, in the context of truly passive far memory no such co-resident CPU exists; in its absence clients must enforce serialization for potentially conflicting operations through RDMA atomic requests. These hardware operations are expensive and known to (dramatically) underperform fully asynchronous verbs, both in terms of throughput and scalability [24]. We consider accelerating two different systems that employ such operations.

Sherman. Sherman [44] is a write-optimized B+Tree for passive remote memory. The tree is augmented using entirely one-sided RDMA operations. Sherman improves performance under contention in two key 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× 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.

Clover. Clover [42] implements a shared remote key-value store using one-sided RDMA requests: reads, writes, and CAS. To improve performance, Clover moves metadata storage off of the data path. On the data path, reads and writes for a given key are made exclusively to an append-only linked list stored in remote memory. All RDMA requests are made to 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 a read or a write fails to land at the tail of the list Clover iteratively traverses the structure until the tail is found. While this provides no liveness guarantees, in the common read-heavy case concurrent clients all eventually reach the end of the list. To speed up operations clients keep caches (i.e., hints) of the end of each key's linked list to avoid traversals. When writes are heavy and/or when particular keys are hot, Clover's performance degrades substantially. By implementing a similar, shared cache at the ToR, SwordBox decreases the likelihood of accesses to stale tail locations.

2.3 Programmable middleboxes

Most current proposals for fully disaggregated compute architectures are scoped to rack scale [3, 11, 20, 25]. The machinery for arbitrating memory access varies between proposals but usually relies on a centralized controller. Some are as simple as a scaled-out PCIe root complex, while others imagine a programmable middlebox with an API exposed to applications [3]. We see the latter as a promising opportunity as it allows developers to highly optimize their programs for remote memory. In this work we envision rack-scale computing with either a programmable switch, or some other programmable hardware (e.g., FPGAs) being used as a centralized switching fabric for memory operations [40]. We assume that, like programmable switches, these middleboxes are highly resource-constrained with a small amount of SRAM intended for forwarding packets and limited functionality for executing programs in the data path.

3 Serialization overheads

Passive memory systems suffer from two fundamental sources of overhead: the additional expense incurred by leveraging any hardware-enforced consistency semantics at the memory servers, and the costs of resolving conflicts from afar. Before introducing our approach to avoiding both, we characterize the potential benefits by quantifying each using microbenchmarks in the context of Sherman and Clover on our testbed RDMA hardware, NVIDIA Mellanox ConnectX-5 NICs.

3.1 RDMA semantics

Any client-based contention management scheme requires a certain level of serialization support at the memory server. There is a trade-off, however, between the strength of the consistency guarantees and the cost of implementing them. RDMA provides two different classes of guarantees; we summarize both below.

3.1.1 Atomic requests

The strongest guarantee available in the RDMA specification is atomicity, through the use of special atomic

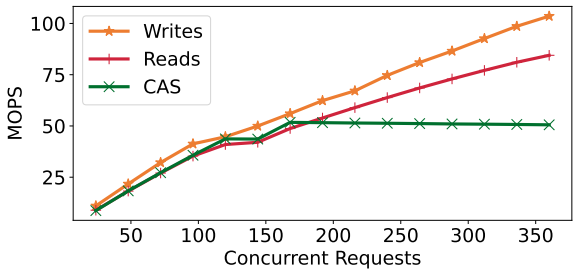

Figure 1: Achieved throughput of RDMA verbs across twenty queue pairs on data-independent addresses as a function of request concurrency. When using atomic requests, ConnectX-5 NICs can support approximately 2.7 MOPS per queue pair, up to about 55 MOPS in aggregate.

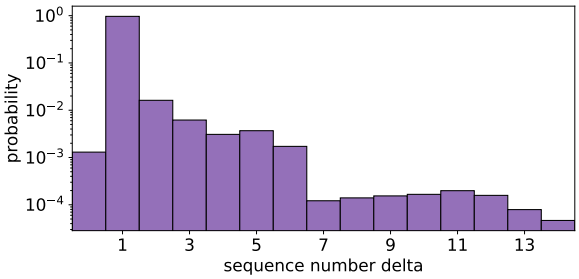

Figure 2: PDF of request reorderings. Retransmitted requests lead to reordering values of zero. 97% of requests retain their order (delta=1), however reorderings of up to 15 requests can occur. (Note logarithmic *y* axis.)

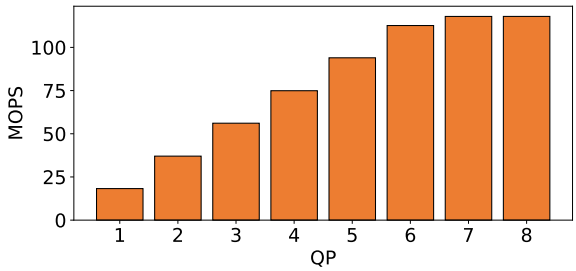

Figure 3: Max throughput as a function of the number of RoCEv2 RC connections. Each QP gets a single core, and issues in-lined writes. A single connection only offers a fraction of the total NIC throughput.

requests like fetch-and-add (FAA) and CAS. 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.

Atomic primitives enable higher-level serialization structures like locks and optimistic concurrency schemes. Unfortunately, atomics are famously expensive [24], 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 1 illustrates the scalability of data-independent (each instruction is issued to an isolated cache line) atomic operations in comparison to standard reads and writes. We confirm that the findings of prior studies [24, 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.

3.1.2 Queue-pair ordering

Hence, a scalable far memory system should avoid the use of atomics to the extent possible, instead relying upon lightweight RDMA verbs. Even then, existing hardware does provide some guarantees. For example, all modern hardware memory management units (MMUs) ensure coherent memory accesses and frequently ensure a total ordering on local memory requests (i.e., sequential consistency). Unfortunately, due to the complexities of today's NUMA architectures and PCIe bus arbitration, it is possible that memory requests may not be serviced in the order they are dispatched by the server's NIC. Similarly, individual RDMA requests may require multiple memory accesses, potentially leading to write corruption and read tearing.

Figure 2 shows that this potential reordering is not just an academic concern, but instead arises 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 queue pairs. Each request is routed through a middlebox 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), but a non-trivial number are out of order by one to five requests, and some by up to 15.

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

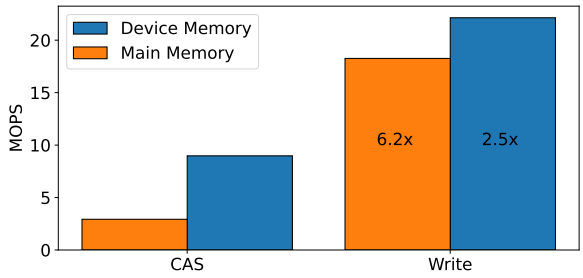

Figure 4: Throughput comparison of serialized RDMA operations in NIC-mapped device and main memory. Writes obtain 6.2× higher throughput than CAS in host memory and 2.5× higher in NIC memory despite being restricted to a single queue pair.

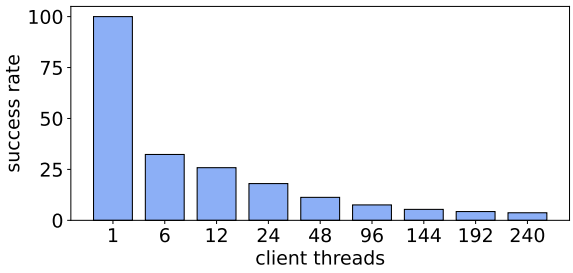

Figure 5: Percentage of successful operations in a 50:50 read-write workload spread across 1,024 keys according to a Zipf(0.99) distribution as more client threads are added. At 240 threads less than 4% of operations succeed.

Happily, the RDMA specification provides the ability to provide partial ordering amongst a subset of requests [19]. Specifically, RDMA provides three different types of queue pairs (QP) with differing ordering—and reliability—guarantees. Unreliable connections and datagrams (UC & UD) provide no ordering guarantees while reliable connections (RC) provide TCP-like in-order processing of requests on that QP. As a result, requests on an individual RC are linearizable with respect to each other, but shared memory accesses across multiple RC are not.

In general, this can provide sequential consistency if all requests relating to a given memory location arrive on the same RC queue pair. Unsurprisingly, however, RC's ordering guarantee comes with its own scalability limitations. Figure 3 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.

3.2 NIC-based memory

Regardless of how operations are serialized, accesses to main memory on a remote server require multiple PCIe round trips to complete. Sherman exploits the fact that newer Mellanox NICs like the ConnectX-5 provide a small region of NIC memory that can be mapped into the address space of RDMA applications, removing the remote PCIe overhead for frequently accessed data like its B+Tree locks. Figure 4 compares the performance of serialized CAS and write operations to addresses in main vs. NIC-hosted device memory. CAS operations are issued across many queue pairs to achieve maximum throughput while the write operations are issued on a single queue pair to enforce serialization. While the use of NIC-hosted memory boosts CAS throughput from approximately 3 to around 9 MOPS, write operations remain dramatically more efficient in either case.

3.3 Optimistic conflict resolution

Optimistic approaches to concurrency management, while far more performant than pessimistic ones in the uncontended case, can frequently be prohibitively expensive when contention is common. In general, the conflicting operation(s) must be retried incurring additional latency and bandwidth. In some systems, the retry is a heavyweight, pessimistic operation, leading to a substantial—but fixed—overhead. In others, like Clover, subsequent attempts remain optimistic, resulting in a linear (per-retry) increase in costs. In the latter case, however, very high rates of contention can lead to congestion collapse, where a retry is essentially doomed to failure, dramatically decreasing system throughput.

As a concrete example we consider the costs of contended operations in Clover. Recall Clover maintains a linked list for each key and attempts to read and write to the tail of the list for each operation. The location of the tail of the list is cached at each client and used as a hint for subsequent operations. If the hint is stale the client obtains an updated tail pointer and retries the RDMA request. Figure 5 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 as client threads increase.

Bandwidth inflation. One of the most direct impacts of failed requests is the bandwidth overhead of the retries. Because a Clover operation actually consists of multiple

RDMA requests, the total bandwidth is not strictly proportional to the failure rate, but rather slightly sub-linear. Nevertheless, the expected cost of each operation rises steadily with contention, which is a function of both the workload and level of concurrency. Our measurements show that under contention the average bandwidth cost of Clover read and write operations can inflate by 16× (Figure 9) when compared to an optimal scenario in which all operations succeed on their first try.

Tail latency. 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 99th-percentile tail latency increases by over 300× (Figure 10) in our experiments.

4 On-path serialization

In this section, we show an on-path serializer can address both the the performance bottlenecks associated with RDMA hardware and the overhead of remote conflict resolution. First, we describe how to accelerate locking as used in Sherman by leveraging the ability to rewrite RDMA requests after their ordering is determined to replace expensive CAS requests with simple write verbs and rely upon the semantics of RDMA RC queue pairs to provide serializability. Second we show how even imperfect knowledge of the instruction ordering can be used to avoid most—but not necessarily all—conflicts by modifying operations in flight, thereby avoiding Clover's expensive conflict resolution machinery in many cases.

4.1 Enforcing ordering

If SwordBox is able to ensure that requests will not be reordered between when it operates on them and their arrival at a memory server—because, for example, it is directly connected over a single link—we can leverage the ordering semantics provided by RDMA queue pairs to completely eliminate data races. In that case, there is no need to incur the expense of RDMA atomic requests; SwordBox can simply replace them with lightweight verbs to dramatically increase the scalability of a given memory server. Importantly, this optimization can be applied selectively to only the set of servers (or memory ranges, i.e., locks) for which SwordBox is suitably located and sufficiently provisioned to manage; atomic operations destined for other servers and/or memory addresses can be left unmodified.

4.1.1 Shared locks

One common way to limit the impact of expensive RDMA atomics is to use them to guard only a subset of the remote data. For example, 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 QP), it is conceptually straightforward for SwordBox to cache the current value of the lock at the ToR. Indeed, one could implement the lock server at the ToR itself [45], 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 our design, SwordBox uses 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.) Moreover, SwordBox forwards all operations for a given address over the same QP to maintain ordering between the ToR and destination server, allowing it to replace the CAS operation with a lightweight write in flight. When a lock (CAS) operation arrives, the switch checks its cache for the state of the lock. If it is unlocked, the CAS operation is replaced with a write and forwarded over a specific QP to the remote server. When the (successful) response comes back, SwordBox updates its cache and converts the write response to a CAS success response before forwarding it back over the original QP to the client. In the case when the lock is already held, the switch 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—but the response will be replaced with a CAS failure so that the sender knows the lock acquisition failed.

4.1.2 Connection remapping

Atomic replacement only works when all operations—across all client connections—that share server state are vectored to the same queue pair. 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. The challenge, of course, is that the RDMA specification stipulates that each client establish its own queue pair with a given server, so operations for a given lock from different clients will arrive at the ToR on distinct queue pairs. SwordBox, then, must interpose on the full set of queue pairs terminated by a given (set of) server(s) and vector operations to queue pairs accordingly. This seems reasonable, as the ToR is usually on-path for all servers in a rack in traditional topologies.

Sequence-space stitching. Multiplexing and demultiplexing RDMA requests across established connections requires significant care. Requests on a single connection must have monotonic sequence numbers from both the sender's and receiver's perspectives. If monotonicity is broken, the server NIC will trigger retransmission or issue explicit congestion notifications. To ensure monotonic sequence numbers on shared connections SwordBox tracks the outstanding sequence number per QP and injects the appropriate sequence number into the packet after the mapping decision has been made. (Ironically, there is no need to ensure any particular ordering among requests from separate clients, so any total ordering suffices.)

When an RDMA request arrives at SwordBox from a client, it is mapped to the appropriate queue pair for the relevant lock and a *stub* is stored to aid in mapping the request back, much like a network address translator (NAT). The stub keeps track of the original request's sequence number, IP address, MAC address, and queue pair. Stubs are stored in an array of size n indexed by their sequence number $\pmod n$ to ensure $O(1)$ lookup when demultiplexing the response. In addition to sequence numbers, a *message* sequence number is used as an RDMA optimization by the memory server NIC. This value is transmitted as part of the RoCE BTH+ header in the response packet and corresponds to the highest request number the server has processed. If this value is wrong in the response packet from the perspective of the sender (i.e., is less than another message sequence number previously received by that client), the entire request is retransmitted. SwordBox maintains the message sequence number each client expects to see by keeping track of the number of requests a client has issued and adding it to the value of the original message sequence number for that connection.

Request coalescing. As an additional complication, SwordBox must deal with the fact that RoCE coalesces some ACKs as an optimization. In particular multiple RDMA requests can be acknowledged by a single acknowledgment, in the form of either an ACK, ATOMIC ACK, or Read Response. This occurs when multiple RDMA requests are processed concurrently. Because SwordBox maps requests across connections, some coalesced acknowledgments may need to be disaggregated from the clients' perspectives. SwordBox can detect such conditions by comparing the acknowledged sequence number to that stored in a client's stub; when a request is coalesced a gap in the sequence number is observable. In this case we generate the needed acknowledgment at SwordBox and insert it into the queue pair.

4.1.3 Atomic replacement

From a RoCEv2 perspective the per-packet transformation from CAS to write can be applied easily in the data path; RoCEv2 CAS and write headers only differ by a few fields. CAS can be thought of as a special case of a write, where the write is conditional and the length is preset to 8 bytes. SwordBox transforms CAS requests to writes by modifying their headers: it switches the RDMA OP field of the CAS to write, copies the CAS value to the write payload location, sets the DMA length of the write to 8, and shrinks the IPv4 length value by 4. Post modification the RoCEv2 and IPv4 checksums are recalculated. The transformation from CAS to write is deterministic and only requires a few cycles to transform the header.

On the memory server, the NIC processes the write and responds with a regular, write ACK. Once the ACK reaches SwordBox we apply the inverse transformation from the ACK to an Atomic ACK which the sender expects to see. RDMA Atomic ACK headers are very similar to regular ACKs with the only difference being that the atomic contains the original data from the memory location of the CAS. This original data is missing due to the transformation so we inject a value which indicates success or failure to the sender (see Section 4.1.1).

4.2 Conflict avoidance

While atomic replacement can alleviate the NIC hardware bottleneck that limits systems like Sherman, it does not address the underlying contention for shared data structures. Systems like Clover that eschew locks in favor of optimistic concurrency therefore see little benefit.

In these cases, however, SwordBox can have an even greater impact by adjusting doomed-to-fail optimistic requests in flight, a process we term *steering*. Note that steering is strictly a performance enhancement—because the requests continue to be (guarded by atomics when necessary and) processed by the server as usual, any unexpected reordering will cause the request to fail (and be retried by Clover), just as it would have without rewriting.

4.2.1 Write steering

Recall that writes in Clover are destined to the presumed tail of a key's linked list, but the target of any individual RDMA request may be out of date due to races with concurrent updates. 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. 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 redirects the corresponding Clover write request by replacing its target with the cached address.

Unfortunately, Clover RDMA CAS requests do not explicitly specify the write operation of which they are a part. SwordBox infers the operation by checking the size of the RDMA request and then extracts the 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 latest write for that key. Our strategy requires 64 bytes of data per key—the size of the RDMA virtual address.

4.2.2 Read steering

While Clover writes contain the key to which they pertain—which allows for a table lookup—Clover's RDMA read requests only contain the target virtual address and a size. 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, allowing keys with higher update rates to maintain longer histories in the table.

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 recovery mechanism kicks in looks up the last known address at the metadata server, then repeats the processes.

4.3 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 RDMA connection state.

At an application level, Clover never acquires any locks so failures do not result in resource stranding, and Sherman periodically detects if locks were kept by a dead client. At the connection level they both rely upon RDMA to provide reliable, in-order delivery of their messages on a per-client basis and react to failed CAS operations by retrying their operations. SwordBox must therefore be able to detect and properly handle packet retransmissions; in particular SwordBox must not treat a retransmitted packet as a new operation. Hence, SwordBox tracks the most recent sequence number on each QP, the transformation it applied (remapping or steering), and whether the operation was ACK'd by the server. Upon retransmission SwordBox re-applies the previous transformation.

Enforcing ordering. Managing responses when mapping queue pairs is somewhat more complex. If a packet is dropped between SwordBox and a server and Sword-

Box 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.

Conflict avoidance. Without queue-pair mapping, there are fewer concerns at the connection level, but SwordBox must now be aware of potential inconsistency between its cache and server state. Concretely, in the case of Clover, 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. The P4 prototype supports conflict avoidance (§4.2) in Clover at 100-Gbps line rate, but does not perform the queue-pair remapping necessary to enforce ordering (§4.1) that our full-featured DPDK implementation provides.

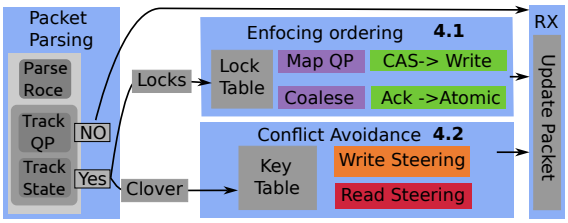

Figure 6: SwordBox's packet processing pipeline.

5.1 P4 implementation

Steering is straightforward to implement in P4. We use P4 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 [21]. 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. SwordBox's switch resource utilization is detailed in Appendix A.3.

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 and Clover header information to identify traffic for steering. When Clover traffic is identified the connection is added to a connection tracker and a connection ID is generated. RDMA invariant CRCs (ICRCs) are tricky as P4 hardware does not support generating them directly. Like some prior work [38], we bypass them in P4 by turning off the ICRC check on our CX5 NICs—the Ethernet CRC provides similar guarantees.

5.2 DPDK vs P4

While our P4 prototype has limited functionality due to hardware limitations of our programmable switch, the DPDK SwordBox implementation (shown in Figure 6) consists of 3,392 lines of C and includes all of the features described in the previous section—including ICRC recalculation (see Appendix A.4). In this section we describe why tracking outstanding requests and uncoalescing ACKs is challenging in current RMT hardware, and suggest small hardware modifications which would make a P4 implementation simpler.

Connection mapping. When a packet is mapped to a queue pair a 64-bit map entry is required to map the

response back to its sender. In the worst case a connection may have a mapped entry for every connected client. Because switch registers are allocated at compile time, statically provisioning for the worst case requires $O(n^2)$ space complexity (n entries for every connection). Supporting only 512 clients requires 2MB of SRAM (1/16th of a Tofino's total buffer space) given this constraint. Storing these 64-bit entries takes 2 pipeline stages in the best case, and 8 in the worst (8-bit registers).

Space-efficient alternatives are harder to implement. Hash tables have much better space utilization, but collisions are hard to deal with. For each potential collision, additional pipeline stages must be allocated and entries must include QP IDs to check for collisions—increasing a single hash table lookup to four stages. At millions of packets per second collisions are common (3 or more) which exceeds the pipeline length of Tofino switches.

Request coalescing. Recall that server NICs frequently summarize multiple ACKs into a single ACK. In order to perform QP mapping, these ACKs must be iteratively reconstructed if the original requests were generated by different clients. Implementing this functionality in P4 is extremely challenging. 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. Of course, the issue could be avoided entirely if it were possible to disable coalescing on server NICs.

6 Evaluation

We use our DPDK implementation to perform a microbenchmark 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.

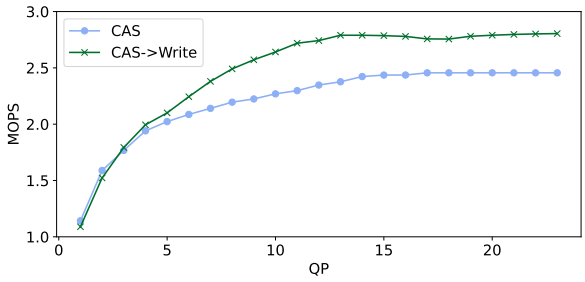

Figure 7: Throughput of conflicting CAS and rewritten CAS requests as a function of client threads/QPs.

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 QP by running a microbenchmark 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 7 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→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 sim-

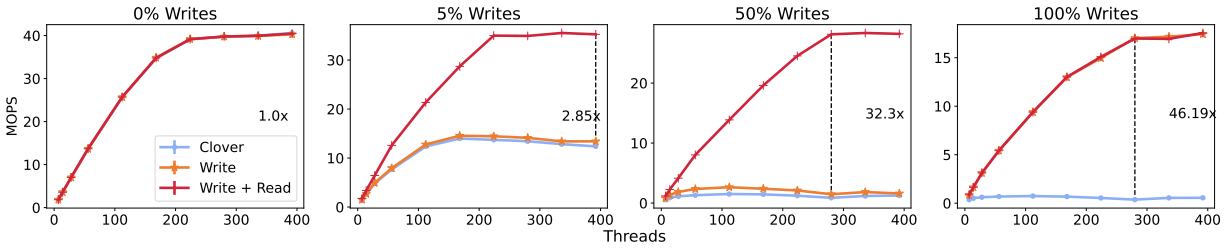

Figure 8: Steering applied to Clover with 128-byte objects across 4 YCSB benchmarks. The percentage of workload writes increases from left to right. SwordBox throughput relative to Clover is 1.0, 2.8, 32, and 46× respectively.

ple 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 1). 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 prior to being forwarded, limiting the forwarding performance of our DPDK implementation to the maximum per-core throughput of our middlebox: 2.8 MOPS.

6.3 Conflict avoidance

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 [7] to access 1,024 128-byte objects stored in Clover.

6.3.1 Throughput

Figure 8 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. Clover (shown in blue) performance decreases markedly with even 5% writes; 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 10.) Of course, in a 100% write workload write steering alone is sufficient.

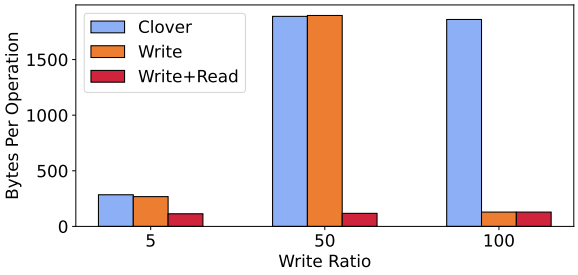

Figure 9: Average number of bytes required per Clover operation on 128-byte objects using each of the three techniques at various write intensities.

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 9 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.

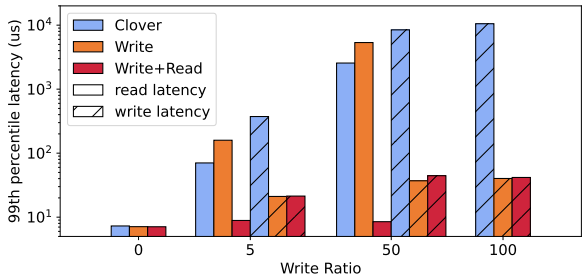

Figure 10: 99th-percentile tail latencies of read (solid) and write (striped) Clover operations at various write intensities. (Note logarithmic *y* axis.)

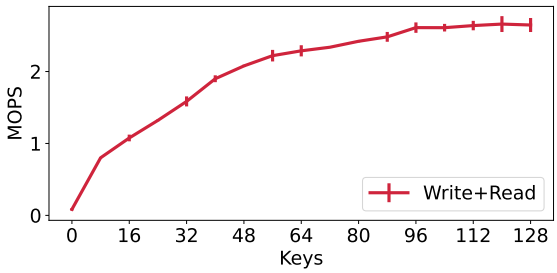

Figure 11: Per-client throughput as a function of the number of Clover keys SwordBox steers. 50:50 workload averaged across 6 hosts each running 56 threads.

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 10 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$\times$ its baseline our our testbed. With read and write steering (solid red) the read tail latency drops to 8 μs—a 8$\times$ 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$\times$. 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$\times$ 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.

6.3.4 Partial steering

One of most appealing aspects of SwordBox's steering is the fact that it need not be applied to all servers, or even memory regions (i.e., Clover keys) of a given server. Figure 11 shows per-client throughput as a function of the number of keys steered by SwordBox. To accentuate the impact, we use a Zipf parameter of 1.5—as opposed to 0.99 in prior experiments—to enhance the locality of requests. (See Appendix A.2 for a full range.) Steering requests for only the hottest-8 keys provides a 9.5$\times$ improvement while tracking the hottest 64 delivers 27$\times$.

7 Limitations

As starkly demonstrated above, the main performance limitations of our SwordBox prototypes are due to their respective implementations. Even our Clover experiments fall short of the underlying hardware limits. While our results show significant performance boosts from reducing hardware contention, there may be additional bottlenecks we have not yet uncovered. In future work we would like to extend our measurements to push the limitations of the underlying hardware. Based on our measurements so far, we expect that higher request rates will see even further benefits from reduced contention.

More generally, some practical aspects of RDMA interposition—especially under failure and overload—are left out of this work. For instance, correctly handling ECN packets is a difficult question when connections are being multiplexed as the generated ECN packet has a single destination. One option is to broadcast the ECN to all clients multiplexed on the destination connection and allow end-to-end congestion control. Another is to keep track of individual client request rates and only issue ECN to the highest requesting clients. We leave the finer points of congestion control to future work.

8 Conclusion

We leverage the top-of-rack switch to resolve in-network contention. SwordBox can improve the performance of systems that rely on atomic operations and optimistic concurrency. Our full-featured prototype demonstrates the potential of removing atomic operations and relying entirely upon queue-pair ordering guarantees. While connection mapping 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] BIELSKI, M., SYRIGOS, I., KATRINIS, K., SYRIVELIS, D., REALE, A., THEODOROPOULOS, D., ALACHIOTIS, N., PNEVMATIKATOS, D., PAP, E., ZERVAS, G., MISHRA, V., SALJOGHEI, A., RIGO, A., ZAZO, J. F., LOPEZ-BUEDO, S., TORRENTS, M., ZYULKYAROV, F., ENRICO, M., AND DE DIOS, O. G. dredbox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (2018), pp. 1093–1098.

[5] 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.

[6] CAVIUM. Liquid IO II 10/25g smart NIC family. https://www.marvell.com/documents/08icqisgkbtn6kstgzh4/, 2017.

[7] 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.

[8] CXL CONSORTIUM. CXL 3.0 specification. https://www.computeexpresslink.org/download-the-specification.

[9] 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.

[10] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 371–384.

[11] 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.

[12] FEELEY, M. J., MORGAN, W. E., PIGHIN, E. P., KARLIN, A. R., LEVY, H. M., AND THEKKATH, C. A. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, Association for Computing Machinery, p. 201–212.

[13] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An open-source 100-Gbps NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2020), pp. 38–46.

[14] FUNGIBLE. The Fungible data processing unit. https://www.fungible.com/product/dpu-platform/, 2021.

[15] 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.

[16] GENZ. Gen-Z Consortium. https://genzconsortium.org/, 2018.

[17] 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.

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

[19] INFINIBAND TRADE ASSOCIATION. Infiniband specification. https://www.afs.enea.it/asantoro/, 2007.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (October 2017), ACM, pp. 137–152.

[27] 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.

[28] MANSOUR, W., JANVIER, N., AND FAJARDO, P. FPGA implementation of RDMA-Based data acquisition system over 100-Gb Ethernet. *IEEE Transactions on Nuclear Science 66*, 7 (Jul 2019), 1138–1143.

[29] 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.

[30] 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.

[31] NANAVATI, M., WIRES, J., AND WARFIELD, A. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 17–33.

[32] 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.

[33] NOVAKOVIC, S., DAGLIS, A., BUGNION, E., FALSAFI, B., AND GROT, B. Scale-out numa. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, Association for Computing Machinery, p. 3–18.

[34] NOVAKOVIC, S., SHAN, Y., KOLLI, A., CUI, M., ZHANG, Y., ERAN, H., LISS, L., WEI, M., TSAFRIR, D., AND AGUILERA, M. K. Storm: a fast transactional dataplane for remote data structures. *CoRR abs/1902.02411* (2019).

[35] NVIDIA. ConnectX SmartNICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[36] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (aug 2015).

[37] 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.

[38] 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.

[39] 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.

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

[41] SINGHVI, A., AKELLA, A., ANDERSON, M., CAUBLE, R., DESHMUKH, H., GIBSON, D., MARTIN, M. M. K., STROMINGER, A., WENISCH, T. F., AND VAHDAT, A. Cliquemap: Productionizing an RMA-Based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), SIGCOMM '21, Association for Computing Machinery, p. 93–105.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

A Appendix

Here we provide additional evaluation results and details regarding our implementations.

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 12 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

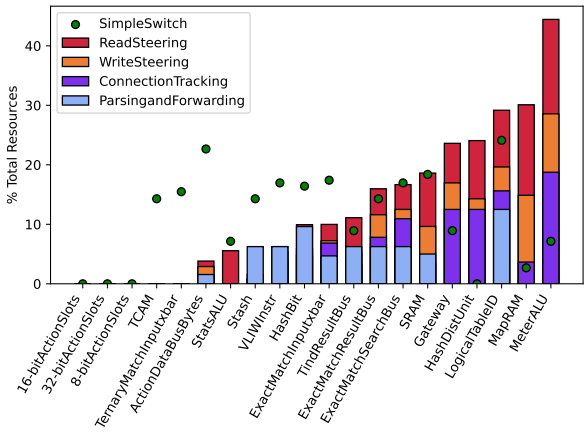

Figure 14: Breakdown of switch resource utilization by SwordBox component.

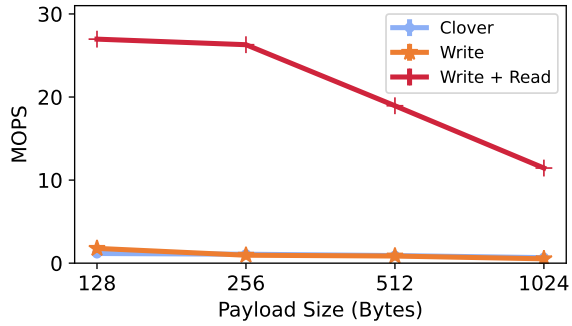

Figure 12: Performance across RDMA payload sizes using 400 client cores at a 50:50 read/write ratio.

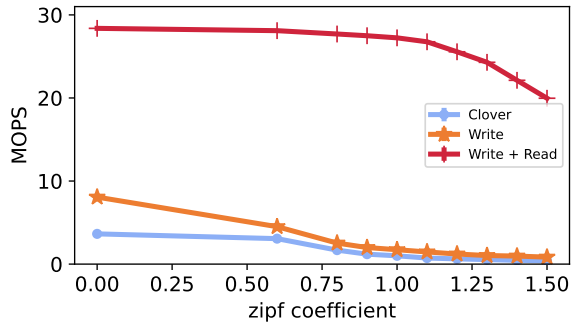

Figure 13: Performance as a function of Zipf coefficient using 400 client cores on a 50% write workload.

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 13 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.

A.3 Switch resources

Figure 14 provides a breakdown of the resource consumption of our P4 SwordBox implementation. We collected these values from our testbed using the barefoot SDE version 9.7.0. Each reported 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.

A.4 RDMA ICRC

RDMA requests are not intended to be modified in flight, and care must be taken not to corrupt them. RDMA invariant CRCs (ICRC) are calculated at the time of sending and are designed to ensure the integrity of the payload. When we modify requests their ICRC must be recalculated or the packet will be rejected by the receiving NIC. Such an error would cause an extreme performance hit as any dropped packet triggers a timeout, and go-back-n retransmission. FPGA implementations of RDMA ICRC have been built in the past [28]; the required CRC calculation is identical to Ethernet CRC, with some additional `crc32` for the calculation. This algorithm is highly optimized for general case CRC calcuation.

Recalculating the CRC for modified requests is the primary overhead of SwordBox as it must be recalculated after a squence number update, which occurs on all multiplexed and demultiplexed requests. This overhead could be reduced by either removing the need for the CRC (which is not a feature available on CX series NICs) or by allowing an alterative, lightweight checksum which could be quickly updated based on changes made to the packet while in flight. While these options are not currently available we believe that commodity switches and SmartNICs could leverage hardware offloads to reduce the cost, as the CRC calculation is identical to Ethernet except masking RoCE-specified packet felids.