In-network Contention Resolution for Disaggregated Memory

Stewart Grant and Alex C. Snoeren
UC San Diego

Abstract

Passive remote memory remains the holy grail of disaggregation. Most existing systems for disaggregated memory either use remote memory simply as a backing store, or design special-purpose data structures that require some amount of processing co-resident with the remote memory to manage and apply updates. The few proposals for truly passive remote memory perform well only with read-mostly workloads, rapidly deteriorating in the face of even low levels of write contention. We propose to leverage in-network devices (specifically, a programmable top-of-rack switch) to serialize remote memory accesses and resolve any write conflicts in flight. Our prototype is able to completely avoid write contention in the recently published Clover disaggregated key/value store, delivering a performance boost of almost 50% on our testbed under a mixed read/write workload.

1 Introduction

Major industrial players are increasingly embracing resource disaggregation, where memory and storage are physically separated from computation. This segregation provides numerous opportunities for increased scalability, power efficiency, and cost savings, but also raises fundamental performance bottlenecks [9]. Disaggregating primary storage (i.e, byte-addressable main memory) remains an unsolved challenge given the dramatic (e.g., 20×) increase in access latency when moving from on-die cache (≈ 50 ns) to network-attached options (often on the order of 1 us). As a result, many previous systems use application-transparent remote memory only as a form of backing store for infrequently accessed data [4, 10, 16, 21].

Alternatively, others have designed high-performance, special-purpose distributed data structures for use over RDMA [7, 11, 12, 14, 17, 18, 19, 20]. These non-transparent systems trade generality for performance; handling millions of operations per second. They all require a CPU be co-resident with remote memory to act as an RDMA coordinator. This design paradigm of CPU/remote memory collocation is at odds with the goals of disaggregation, demanding new approaches for remote-resource coordination. Researchers have considered a variety of trade-offs along this spectrum [2, 5, 6, 8, 9]. Ideally, systems could leverage what is sometimes called passive remote [22] or far [2] memory, which does not have local computation attached. Rather, all accesses are remote, using hardware primitives like one-sided RDMA.

Relatively few systems have been proposed to leverage passive remote memory, and those that have [3, 22] are designed to support read-heavy workloads on shared resources. Their reason for avoiding writes is fundamental: ensuring consistency in the face of concurrent writes is expensive. Lock acquisition and revocation requires multiple round trips which would devastate throughput. Conversely, optimistic schemes break down in the face of contention and typically fall back to even more expensive recovery operations.

One recent system, Clover [22], attempts to decrease the cost of conflicts by separating (potentially contented) metadata operations from the memory updates themselves. While effective for read-mostly workloads, Clover simply delays the inevitable, and metadata contention quickly becomes the bottleneck at even modest update rates. Indeed, published results show that Clover's throughput drops by more than a factor of four when moving from a read-only to 50%-write workload [22, Fig. 7]. The key issue with designs like Clover is the requirement that individual clients resolve conflicts themselves, resulting in extremely expensive operations in the worst case. The authors report that Clover requires five round trips to remote memory in the 99th-percentile case with just a 5%-write workload [22, Table 2].

The obvious alternative to completely distributed contention resolution is to have a central coordinator that manages remote memory accesses. Prior work has suggested the potential benefits of such a distributed memory controller [5], but to our knowledge none have yet been built or proposed with existing hardware. The authors of Clover evaluate a commodity server in such a role (which they refer to as pDPM-central) and discard it as a scal-

ability bottleneck that also increases the number of network hops between a node and its remote memory [22]. While an end host is indeed ill-suited for this purpose, an in-network device, on the other hand, could be an ideal location to provide such a service: it is, after all, on-path, and, if suitably located, may be in a position to observe—and even avoid—conflicting access requests before they reach remote memory.

In this paper we explore the potential for a programmable top-of-rack (TOR) switch to mediate remote memory accesses. We observe that if all remote reads and writes are performed within a single rack—which is often the case in existing RDMA deployments—the TOR is in a unique position to observe—and serialize—remote memory operations. Moreover, if the TOR is able to parse and modify requests in flight, it can even resolve conflicts before they reach the remote memory, effectively recovering from optimistic write failures "for free." Finally, if clients retain the ability to resolve conflicts themselves (albeit at a significant cost to performance) the TOR can serve simply as a form of soft-state performance-enhancing proxy; conflicts that it fails to avoid result in decreased performance as opposed to incorrect semantics.

As a proof of concept we implement a prototype software-based TOR that interposes on the Clover protocol and resolves write conflicts in flight. Our conflict-detection algorithm uses knowledge of Clover's metadata structures to detect write conflicts using a small amount of cached state (only a few bytes per key/value pair). Conflict resolution is performed directly on the in-flight RDMA packets by steering their destination virtual addresses to the most up-to-date location in Clover's key/value store. This technique is complete and safe: it resolves all read and write conflicts for keys that it caches, and reverts to native Clover performance for those it does not. As in-network SRAM is expensive we show that our technique can trade completeness for memory savings while still achieving performance gains. Our preliminary evaluation with a 50/50 read/write workload shows throughput gains of $1.42\times$ when caching all keys, and gains of $1.34\times$ when using only 128 bytes of in-network memory to correct conflicts for the 8-most-popular keys in a Zipf request distribution.

2 Background

While our insight and approach is general, in order to demonstrate its practicality we prototype it in the context of Clover [22], a disaggregated key/value store. Hence, we begin with a brief overview of the relevant aspects of Clover's design for the uninitiated.

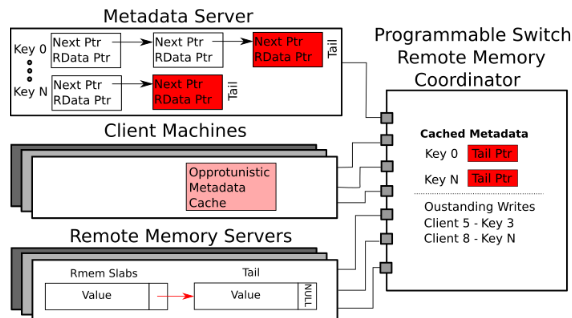

Figure 1: System overview, Metadata, client, and Remote Memory servers are Clover components. Our remote memory coordinator is located on a centralized TOR interconnecting the clover components.

2.1 Append-only updates

Clover maximizes performance by storing data in a versioned list, dubbed a *chain* by the authors, which allows for fully asynchronous non-blocking reads and $O(1)$ writes which succeed opportunistically. Writes are issued by clients as atomic append operations to the end of the chain, the structure of which is illustrated in Figure 1. Each entry in the chain contains a pointer to the next entry with the pointer at the tail of the chain set to `NULL`. Clients keep cached pointers to the current tail of the chain for each key allowing them to issue lock-free RDMA reads directly to passive remote memory servers using the address they believe to be the current tail. Clients can independently confirm their reads are fresh by ensuring the value returned has a `NULL` next-update pointer.

Write (i.e., append-to-tail) operations, on the other hand, require two steps in the common case. First a writer issues a lock-free RDMA write to store the new value update to an uncontended portion of remote memory. The second operation optimistically attempts to make the update globally visible by atomically committing it to the end of the chain. Specifically, the writer issues an RDMA compare-and-swap (`c&s`) operation to the old tail to replace its (presumably still `NULL`) next-update pointer with the address of the new value. If successful, it then updates the metadata sever with the new end-of-chain address.

The Clover authors compare their opportunistic approach with a variety of different system architectures including one which centralizes metadata on the datapath (pDPM-central). They find that their opportunistic approach achieves extremely high throughput on read-heavy workloads, performing similarly to raw RDMA reads. In contrast pDPM-central bottlenecks at far lower throughput. In the presence of writes Clover's through-

put decreases due to contention while the performance of pDPM-central remains the same as it resolves all concurrent write conflicts in the data path.

2.2 Conflict handling

If multiple clients attempt to update the value concurrently, each will succeed in their first write to their private region of remote memory but then issue conflicting RDMA `c&s` operations to the end of the shared chain. The race is resolved at the remote memory location of the previous update in the chain: only the first `c&s` operation will succeed; all subsequent operations will fail because rather than finding a value with a `NULL` next-update pointer, they find the now (at best) penultimate value in the chain which points to the update issued by the client that won the race. During this two-RTT operation any concurrent write to the same key will cause a conflict.

If a client's write—or read—fails because it used a stale tail pointer the client concurrently requests the new tail address from the metadata server and iteratively traverses the chain themselves to find the current end in a processes known as *chain walking*. (It is insufficient to just consult the metadata server because it is updated lazily.) Failed writers must then issue a new `c&s` operation at the updated end of the chain to append their pending update. Of course, the reissued `c&s` is subject to the same race condition as many writers may be executing concurrently. This pointer-chasing reconciliation algorithm must be run independently each time a conflict occurs. While concurrent read operations will not prevent a write from succeeding, a reader that loses the race with a `c&s` operation will need to issue another RDMA read to the new address to obtain the updated value. On write-heavy workloads these race conditions happen frequently leading to large and unpredictable tail latencies [22, Table 2].

3 In-network conflict resolution

We propose a middle ground between a fully distributed and a centralized approach. Our insight is that by using data-structure-specific knowledge and caching metadata in the network, conflicting writes can be resolved at line rate on the data path using only a small amount of state. We use Clover as a platform to prove our concept and design a middlebox algorithm which intercepts Clover's RDMA read, write, and `c&s` requests, caches a small amount (64 bytes per key) of structural metadata, and resolves write conflicts by adjusting the virtual destination address of contending `c&s` guards.

3.1 Resolving concurrent requests

Our key observation is that a top-of-rack (TOR) switch necessarily serializes all RDMA operations destined to any particular remote memory location, since physical memory is hosted on a single server, and the switch is in charge of ordering packets destined to each of its output ports. (We leave the engineering details involved in addressing multi-homed servers, corrupt packets, node failures, and the like to future work.) Given an ordered stream of RDMA operations, it is straightforward for an in-network device to detect stale `c&s` operations. What's more, if the TOR maintains just a modest amount of state, it can resolve conflicts in flight. Specifically, it suffices to redirect any `c&s` operations that "lost their races" to the address of the current (or soon-to-be) end of the chain.

In the particular case of Clover, we cache the current end of the chain for each key in Clover's key/value store. This requires $O(n)$ state where n is the number of keys. For a key/value store consisting of ten-thousand keys we need to store a key mapping and value for each (both 64 bits) resulting in a total in-network storage of 80 KB. Note that this represents only a small portion of the metadata Clover maintains at its metadata servers which contains all versions on the chain. Only the end of the chain is required to resolve write conflicts.

3.2 Modifying RDMA in flight

Interposing on the RDMA protocol, however, is non-trivial. One approach (evaluated in Clover as pDPM-central [22]) employs an RDMA-enabled middlebox that sets up connections between itself and clients and another set of connections to the memory servers. The middlebox can then reorder, rewrite, and relay RDMA requests from clients to the appropriate chain locations. This solution is impractical for a TOR-based solution both because existing switches lack the ability to establish RDMA connections, and, more importantly, because the authors of Clover demonstrate it to be a performance bottleneck. We avoid both shortcomings by transparently intercepting RDMA connections established directly between the clients and remote memory servers without explicitly participating in the RDMA protocol.

In our approach, the TOR records the virtual addresses used in any RDMA writes made by Clover clients. These writes are marked as *outstanding*; i.e., they have been issued to remote memory, but have not yet been made visible in Clover because they are not yet connected to Clover's per-key chain. As described above, following the completion of an outstanding write, clients issue an RDMA `c&s` to the end of the chain to make their up-

date visible atomically in Clover, and we also track the last c&s issued for each key. In a successful update, the c&s operation will adjust the next-update pointer to the location of the most recent (outstanding) write, thereby committing the write.

Our algorithm detects the existence of a conflict by noticing when there are multiple outstanding writes for a given key. The current write for each client is tracked, when two clients have outstanding writes to the same key a conflict has occurred. When a c&s operation arrives, we inspect the virtual address it is trying to update. If it points to an old tail, its virtual address is modified by the TOR (without the knowledge of the issuing client) to point to the true tail of the list. The cached latest version of the key is then updated to point to the address to which the c&s was directed. Clover clients learn the updated locations using their default read algorithm.

3.3 Prototype

Our prototype is implemented in a software switch using DPDK, but is designed to have low memory and computational overhead making it ideal for network devices such as programmable switches. RDMA packets 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 c&s packets their ICRC must be recalculated or the packet will be rejected by the receiving NIC. FPGA implementations of RDMA ICRC have been built in the past [15]; the required CRC calculation is identical to Ethernet CRC, with some additional header components and field masking.

4 Evaluation

Our testbed consists of four machines, where a single server plays the role of both Clover memory server (MS) and metadata server (DN). Two machines are configured as Clover clients, and the last hosts our DPDK-based TOR. Physically, the machines are identical: each is equipped with two Intel Xeon E5-2640 CPUs and 256 GB of main memory evenly spread across the NUMA domains. Each server communicates using a Mellanox ConnectX-5 100-Gbps NIC installed in a 16x PCIe slot interconnected via a 100-Gbps Mellanox Onyx Switch. All Clover servers are configured with default routing settings: clients send directly to the metadata and data server. We install OpenFlow rules on the Onyx switch to redirect the Clover RDMA traffic to the DPDK “TOR”.

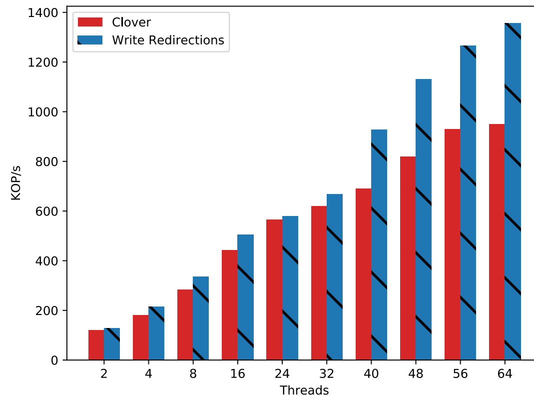

Figure 2: Default Clover throughput vs. Clover with write conflict detection and correction turned on

4.1 Conflict resolution

We test the performance gains of resolving write conflicts using our caching TOR. Clover clients are configured to run a YCSB-A benchmark, 50% read, 50% write for 1 million requests. Requests for keys are based on a Zipf distribution generated with an s value of 0.75. In each experiment the number of client threads is increased which in turn increases the load on the system. Clover requests are blocking; thus, the throughput is a factor of both the request latency and the number of client threads. Figure 2 compares the performance of native Clover (plotted in red) against our in-network conflict resolution (hatched blue).

As the number of clients increases so too does the probability that two client threads will make concurrent writes to the same key. The number of conflicts resolved in flight directly correlates to throughput improvements as each successful request reduces the multiple round trips necessary to resolve write conflicts. Our current implementation provides a $1.42\times$ throughput improvement at 64 client threads.

Throughput is limited by the scale of our experimental setup, i.e more client machines can produce higher throughputs. The number of in-flight conflicts is also impacted by the Zipf distribution. We use a Zipf of 0.75, however a Zipf of 1.0 would result in a distribution skewed towards fewer keys, which in turn results in more conflicts. Moreover, we find that Clover’s current design leads to hardware contention on the servers themselves. In particular, ConnectX-5 NIC performance degrades as the number of RDMA c&s operations to the same memory region across different queue pairs increases [13]. As our design eliminates the need for c&s operations on cached keys, future work will seek to reduce or eliminate c&s

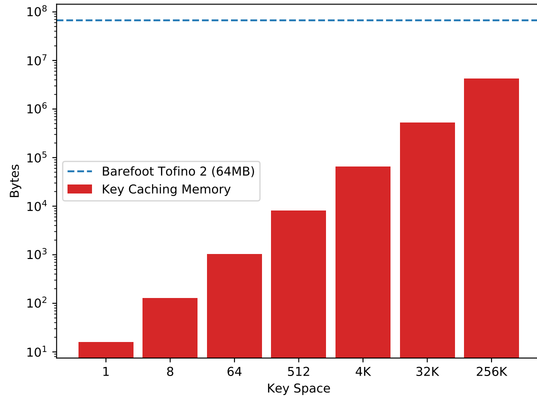

Figure 3: Cost of caching metadata in-network vs. key space size

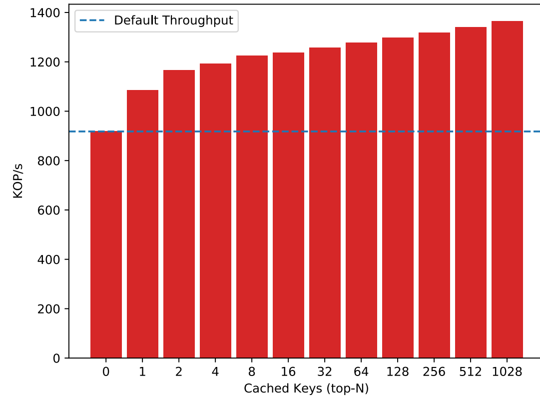

Figure 4: Performance as a function of keys cached. Caching a few of the top-N keys provides the greatest marginal throughput benefits.

operations by replacing them at the TOR with RDMA writes.

4.2 Memory consumption

Resources on networking hardware are scarce. High-end SoC SmartNICs have just a few gigabytes of RAM, and programmable switches have only megabytes of SRAM. Moreover the use of this memory is not free: using memory for any purpose other than buffering packets has a direct performance cost. The metadata we cache in network is minimal: we only cache the virtual address of the last write per key, as well as the last key written per client. Clients are not explicitly known to our middlebox and are identified at runtime by their QP. Tracking clients in this way is necessary to detect write conflicts in Clover. This overhead could be eliminated by explicitly adding key information to `c&s` requests. Figure 3 shows the memory overhead as a function of keys. Note that 100K keys can be supported using 2.5% of the available memory (64 MB) on a Barefoot Tofino 2 programmable switch [1].

Hot keys are the most likely to contribute to conflicts. We test the effect of caching only hot keys by restricting our in-network cache to track and resolve conflicts on only the top-N keys. In this experiment RDMA requests for keys which are not cached pass through our DPDK TOR without modification; conflicts are resolved using Clover's existing reconciliation protocol. Figure 4 shows the throughput for 64 client threads when caching a varying number of keys out of a total key space size of 1024 keys. The request distribution is Zipf(0.75), therefore the vast majority of conflicts occur on the top-eight keys. The in-network memory requirement is 128 bytes, which results in $1.3\times$ throughput improvement.

5 Discussion and future work

In our approach the switch updates its memory prior to the RDMA packet landing in remote memory. This operation is safe under the assumption that no packets are reordered after egress from the switch and that all operations are successful. If a `c&s` packet updates switch memory, and then is rejected by the NIC or end host a reconciliation of memory must take place. The indication to the switch that a failure has occurred is an RDMA `c&s` NACK. When this occurs the switch can dump all of its soft state and reset. This will cause the Clover protocol to revert to its default chain walk to learn new values. Our approach requires only a single successful `c&s` operation per key to rebuild its cache.

Additional performance. Despite our significant performance gain, there are several more optimizations that could be made in a Clover-specific design. As observed during our evaluation, compare-and-swap operations bottleneck quickly on existing hardware when locking is applied across queue pairs [13], limiting the maximum performance of systems that rely on it as a guard. We are exploring two potential approaches for reducing NIC-based lock contention: 1) Remap keys to QPs in flight. Cross-QP locking can be avoided if all requests to a shared remote memory address arrive on the same destination QP. 2) Compare-and-swap is not required for requests handled by our algorithm as they are serialized. These guarded write operations can be converted to regular writes by replacing a few RDMA header fields. This approach would allow full-speed operation throughput with zero locking.

Moreover, our current implementation concentrates of fixing write contention, however there is no limitation

which prevents us from gaining a performance boost on reads. The same RDMA cache could be used to steer reads which are issued by clients with stale information.

Alternative datastructures. While our initial exploration has focused explicitly on Clover and its append-only key/value chain structure, our approach is not limited to a particular datastructure nor only associative operations. More complex structures can be supported, but the choice of structure must be made with care. For our caching approach to resolve metadata conflicts in-network, it requires enough information to enforce remote datastructure integrity invariants. Invariants such as ordering, or maintaining a balance in a tree require more metadata and computation to enforce than appending to the tail of a list. We plan to investigate data structures which have the ideal property of requiring a small amount of metadata (ideally $O(\log n)$, or $O(\log \log n)$) to maintain their structural invariants while also supporting more operations, such as range queries.

The more complicated the structural invariants are to maintain, the greater the information which must be cached; for example an *ordered* list. To illustrate the additional complexity of maintaining order consider how clients could perform inserts. First, like Clover, clients could write their entry to a private memory region. Second, two pointers must be written, one which points to the next item, and another from the prior item to the newly written one. The client could issue the writes itself, however when the insert occurs it would need to traverse part of the list to ensure that the result had been inserted to the correct location and collect a lock on both the prior and successor items. Naively enforcing the ordering invariant requires that the switch cache the entire list.

We are exploring the class of data structures which have either weak structural invariants, or those which only cost $O(1)$ to check. Additionally some data structures amortize the cost of operations which require complex invariants. For instance, rather than storing an ordered list, using a partially ordered list with fast accesses which can be periodically transformed with expensive operations to be consistent.

Deployability. Designing and running custom code on programmable switches is hard, while understanding how to resolve write conflicts is relatively easy. We would like to design a generic interface for developers to resolve write conflicts, and orchestrate in-flight RDMA operations in an application-independent fashion, perhaps as part of a larger disaggregated computing framework or operating system [21].

Indeed, such a system might further consider *where* to deploy conflict-resolution logic. The advantage of using a TOR is that all operations within a rack can be serialized. However in many cases this degree of total ordering is not required. For instance access to a single memory server can be serialized by performing ordering on a NIC connected to the end host. Our techniques could be built into SmartNICs which would allow for them to scale arbitrarily under the assumption that writes do not span multiple remote memory machines.

Acknowledgments

This work is supported in part by the Advanced Research Projects Agency-Energy (ARPA-E) and the National Science Foundation (NSF) through grant CNS-1911104. The authors thank Alex Forencich and Cindy Moore for assistance with our testbed, and Yiying Zhang and the anonymous reviewers for helpful suggestions and feedback. Additional thanks to Yizhou Shan for his guidance in setting up our own instance of Clover, and special thanks to Steve and Don Grant of Halftime Holdings Ltd. for providing workstations, Internet and coffee.

References

[1] Intel tofino 2 p4 programmability with more bandwidth. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series/tofino-2.html`, 2020.

[2] AGUILERA, M., KEETON, K., NOVAKOVIC, S., AND SINGHAL, S. Designing far memory data structures: Think outside the box. In *17th Workshop on Hot Topics in Operating Systems (HotOS)* (May 2019), ACM.

[3] 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.

[4] 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.

[5] 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.

[6] CARBONARI, A., AND BESCHASNIKH, I. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets-XVI, Association for Computing Machinery, p. 164170.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 295306.

[13] 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.

[14] 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.

[15] 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), 11381143.

[16] 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.

[17] 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.

[18] 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.

[19] 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. 318.

[20] 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).

[21] 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.

[22] Tsai, S.-Y., Shan, Y., and Zhang, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 33–48.