Scaling Up Memory Disaggregated Applications with Smart

Feng Ren[†], Mingxing Zhang[†], Kang Chen[†], Huaxia Xia[‡], Zuoning Chen[♣], Yongwei Wu[†]
[†]Tsinghua University; [‡]Meituan; [♣]Chinese Academy of Engineering

Abstract

Recent developments in RDMA networks are leading to the trend of memory disaggregation. However, the performance of each compute node is still limited by the network, especially when it needs to perform a large number of concurrent fine-grained remote accesses. According to our evaluations, existing IOPS-bound disaggregated applications do not scale well beyond 32 cores, and therefore do not take full advantage of today's many-core machines.

After an in-depth analysis of the internal architecture of RNIC, we found three major scale-up bottlenecks that limit the throughput of today's disaggregated applications: (1) implicit contention of doorbell registers, (2) cache trashing caused by excessive outstanding work requests, and (3) wasted IOPS from unsuccessful CAS retries. However, the solutions to these problems involve many low-level details that are not familiar to application developers. To ease the burden on developers, we propose Smart, an RDMA programming framework that hides the above details by providing an interface similar to one-sided RDMA verbs.

We take 44 and 16 lines of code to refactor the state-of-the-art disaggregated hash table (RACE) and persistent transaction processing system (FORD) with Smart, improving their throughput by up to 132.4× and 5.2×, respectively. We have also refactored Sherman (a recent disaggregated B$^+$Tree) with Smart and an additional speculative lookup optimization (48 lines of code changed), which changes its memory access pattern from bandwidth-bound to IOPS-bound and leads to a speedup of 2.0×. Smart is publicly available at https://github.com/madsys-dev/smart.

CCS Concepts: **• Computer systems organization → Distributed architectures**; **Multicore architectures**; **• Networks → Network servers**; *Network performance analysis.*

Keywords: disaggrgated memory, one-sided RDMA, scale-up

ACM Reference Format:
Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with Smart. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3617232.3624857

1 Introduction

Due to the development of high-speed network technologies (e.g., RDMA), memory disaggregation has recently gained extensive interest [1, 6, 7, 9, 10, 18, 20, 44, 46, 54, 55, 65, 66]. It is considered to be beneficial for memory utilization, system elasticity, and failure isolation [46, 54]. However, there is still a significant gap in IOPS between local and remote memory accesses. Thus, the performance of many important disaggregated applications, such as key-value stores [33, 37, 47, 52, 57, 68, 69] and transaction processing systems [58, 64], are still limited by the network, because they need to perform a large number of concurrent fine-grained remote accesses.

According to our evaluation, the throughput of these disaggregated systems typically peaks and begins to decline with more than 32 cores. This lack of scale-up capability can be a common problem for IOPS-bound memory disaggregated applications that the cost of processing RDMA requests dominates the total execution time, which includes many important workloads such as disaggregated cache servers [29, 33], online transaction processing (OLTP) databases [58], and parameter servers [59]. Although these systems can still improve the overall performance by scaling out the cluster, they cannot take full advantage of the many-core machines now common in data centers, which can contain dozens or even hundreds of cores.

To address this issue, we performed a detailed analysis of the architecture of the RNIC (RDMA Network Interface Card) and found that several internal structures are resource contention points at high concurrency (Figure 1). As a result, simply increasing the parallelism (i.e., more threads) or concurrency depth (i.e., more concurrent RDMA operations per thread) does not necessarily improve performance. On the contrary, it can lead to worse performance due to implicit contention of certain internal RNIC resources. Furthermore, we observe that the maximum throughput of RDMA operations can be largely improved by a better paradigm and configuration of the use of the low-level APIs of the RNIC.

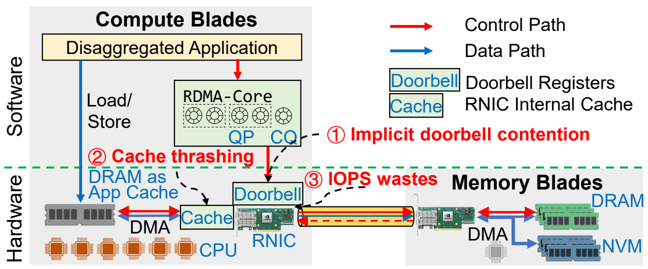


Figure 1. Major resource contention points in the memory disaggregation architecture.

Previous studies [26, 30] have already shown that low-level details are surprisingly important for the efficiency of RDMA systems. However, due to the complexity of RNIC, these details are not familiar to application developers, especially when the application scenario is constantly changing. For example, scaling to more than 32 threads is only considered important for newer RNICs, whose maximum IOPS are greatly increased and therefore can only be fully exploited by increasing parallelism. Therefore, we argue that these details should **be hidden from** application developers with good abstractions.

Contributions. In this paper, we analyze the impact of three major resource contention points, which become scale-up bottlenecks and have a broad impact on the performance of IOPS-bound disaggregated applications. The analysis uses both micro-benchmarks and RNIC performance counters (§3).

In particular, existing works [5, 16, 27, 41] have observed that the throughput of RDMA operations, especially one-sided operations that rely on *reliable connection* (RC), do not scale up well with the increasing of parallelism (i.e., the number of threads performing RDMA operations in parallel). Researchers speculate that this scalability problem may be due to cache contention within the RNIC. Recent RNIC devices improve the performance of metadata object access through on-chip SRAM caches. There is a huge penalty for cache misses because the RNIC must fetch data from DRAM through PCIe DMA reads that take several microseconds [41].

To mitigate this problem, a commonly used optimization called *connection multiplexing* [16, 52, 53] is proposed to reduce the total number of queue pairs (QPs), because QP is needed to establish RC connections and therefore should be cached for good performance. Instead of assigning a separate QP to each thread, each QP is shared by multiple threads using connection multiplexing, which trades off parallelism to reduce resource contention. However, existing works [16, 27] have shown that the sharing of QP leads to suboptimal performance because access to the QP is serialized by locks. It is also confirmed by our experiment (Figure 3).

In contrast, our investigation shows that the internal data structures of RNIC are much more complex than its external abstraction of QPs. Thus, a large number of QPs leads to poor performance, which opens up the possibility of achieving higher parallelism. In fact, we found that ① *the contention in an internal data structure called the* **doorbell register***, instead of the cache, is the real contention point when a large number of QPs are created.* The creation of a doorbell register is not directly exposed to application developers. However, the default configuration and the connection mapping between threads, QPs, and doorbell registers are not optimized for high parallelism. This is the key reason for performance degradation and can be addressed by a general thread-aware RDMA resource (e.g., doorbell registers) allocation mechanism.

After removing the limitation of parallelism, cache contention can indeed become a major scale-up bottleneck for achieving the maximum throughput. But, rather than the number of QPs, ② *the excessive number of* **outstanding work requests** *(OWRs, work requests that have been posted by applications but still have not yet been completed by the RNIC) is actually the main cause of the cache thrashing problem.* As a result, the occurrence of cache thrashing is not only related to parallelism, but also to the depth of concurrency (i.e., concurrent operations per thread executed through asynchronous APIs). To address this issue, we propose a credit-based throttling policy where the depth threshold is automatically determined according to the current workload.

In addition, due to application-level concurrency control, ③ *simply increasing the throughput of RDMA operations does not necessarily improve the throughput of upper-layer applications.* According to our investigation, many disaggregated applications propose lock-based or lock-free solutions to avoid data races caused by conflict updates from different clients. This conflict results in wasted network IOPS, which is exacerbated in high-concurrency environments. Hierarchical on-chip lock (HOPL) [57] attempts to address this problem with a lock-based solution. However, other solutions are also needed in general cases, such as lock-free data structures [52, 68, 69] and optimistic locking [61]. To this end, we propose an adaptive backoff technique that further throttles the concurrency of unsuccessful CAS (compare-and-swap) operations.

The above three major scale-up bottlenecks are summarized in Figure 1. To help developers address these issues, we propose SMART, an RDMA programming framework that hides the technical details of corresponding solutions under an easy-to-use interface (§4). SMART provides a set of coroutine-based asynchronous APIs that are similar to the original RDMA verbs, so the cost of refactoring is low.

Our optimization is also **general** for scaling up the application throughput using RC verbs. Some of these bottlenecks are also found in some non-disaggregated applications, and we expect SMART's techniques to be effective. For example, many distributed transaction systems [16, 60, 61] issue one-sided RDMA requests at high frequency with many threads, so thread-aware resource allocation will be helpful to prevent internal doorbell contention.

To demonstrate the benefits of SMART, we refactor two typical and important IOPS-bound state-of-the-art disaggregated applications: 1) a hash table (SMART-HT) over RACE [68, 69] (with 44 lines of code changed), and 2) a distributed transaction processing system (SMART-DTX) over FORD [64] (with 16 lines of code changed). We have also built a disaggregated B$^+$Tree (SMART-BT) over Sherman [57] with SMART (with 48 lines of code changed) and an additional speculative lookup optimization that further reduces the read amplification and transforms the workload from bandwidth-bound to IOPS-bound. Evaluation results show that for many realistic workloads, applications using SMART outperform their state-of-the-art counterparts. SMART-HT outperforms RACE by up to 132.4× in write-heavy workloads, SMART-DTX has up to 5.2× higher throughput than FORD in SmallBank, and the throughput of SMART-BT is up to 2.0× better than Sherman in read-only workloads. In addition, SMART is also very helpful in reducing latency. For example, the median latency of SMART-DTX is only 28.9% of FORD in SmallBank workloads.

2 Background

2.1 Memory Disaggregation

Traditional datacenters consist of monolithic servers that have both compute and memory resources. However, as the ratio of the resources is fixed, such an architecture results in low memory utilization [46, 51]. To solve this problem, the *memory disaggregation* architecture [3, 9, 52, 54, 64] is proposed. Unlike traditional monolithic servers, the compute and memory resources are physically separated into *compute pool* and *memory pool* respectively. The compute pool consists of *compute blades*, each of which has only a small DRAM buffer but can contain many CPU cores. The memory pool, on the other hand, is a collection of *memory blades*, each of which has a large amount of memory but *weak* computational power (e.g., 1 ~ 2 CPU cores), so they are unable to handle extensive computation [64, 68]. Both blades are interconnected by high-speed networks, such as RDMA, Gen-Z [7], and CXL [6]. As RDMA has been deployed in many datacenters, in this paper, we consider compute blades accessing memory blades using one-sided RDMA verbs.

2.2 RDMA Network

RDMA is a key technology for memory disaggregation. Recent RNICs such as Mellanox ConnectX-6 have achieved up to 200 Gbps bandwidth and sub-600 ns latency, which is sufficient for many disaggregated applications [14, 17]. In addition, RDMA supports one-sided verbs, i.e., READ, WRITE, CAS (compare-and-swap), and FAA (fetch-and-add), which operate directly on remote memory without involving weak CPUs in memory blades.

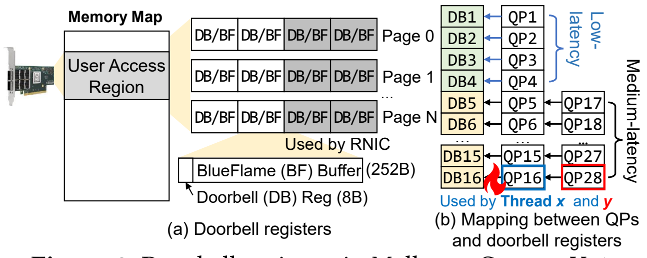


Figure 2. Doorbell registers in Mellanox ConnectX-6.

The programming interface of RDMA is centralized around the abstraction of queue pairs (QPs). Each QP contains a pair of send queue and receive queue and is associated with a completion queue (CQ). For each RDMA operation, the library posts a work request (WR) to the QP. Then, after a WR is acknowledged by the receiver's RNIC, the sender's RNIC performs a PCIe DMA write that appends a CQ entry, and the application is notified by continuously polling the CQ. However, underneath this simple abstraction, the internal structure of RNIC is very complex and unfamiliar to application developers. According to our investigation, some of the most important mechanisms are described below.

Doorbell Register (DB) is the mechanism used by the RNIC to receive notifications of newly enqueued work requests. Figure 2 illustrates the doorbell registers in ConnectX-6 [12]. By default, up to 16 DBs are allocated per RDMA device context, including 4 *low-latency* DBs and 12 *medium-latency* DBs. Each low-latency DB is associated with only one QP, but multiple QPs can be associated with the same medium-latency DB. These doorbell registers are mapped in the User Access Region (UAR), and the driver library performs memory-mapped I/O (MMIO) writes to update them. For drivers used by Connect-IB to ConnectX-7[1], updates to the same DB are all protected by a spinlock[2]. As a result, when a DB is shared by multiple QPs used by different threads, an implicit thread contention occurs and becomes an implicit bottleneck that limits the parallelism of disaggregated applications. For example, as shown in Figure 2(b), thread x using QP16 may be contended by thread y using QP28 because both QPs are associated with DB16. Obviously, such a default mapping is not optimized for massively parallel applications.

Memory Translation and Protection Table. Many existing works worry that too many QPs will exhaust the small on-chip SRAM cache in the RNIC and lead to cache thrashing problems [5, 16, 25, 41]. But, the caches inside RNICs are actually used to cache many other important objects whose sizes are not proportional to the number of QPs [26]. For example, once the RNIC is notified by the doorbell ringing,

[1]Also include ConnectX-2 (CX-2), CX-3 (Pro), CX-4, CX-5, CX-6.

[2]Enabling the `MLX5_SINGLE_THREADED` environment variable removes such spinlocks, but leads potential races between two CPUs due to write-combining buffers in multi-threaded applications [28, 38]. With BlueFlame enabled, the RNIC may even receive corrupted work queue elements (WQEs).

it must perform virtual-to-physical address translation and security checking by looking up the memory translation table (MTT) and the memory protection table (MPT). These mapping tables are cached in MTT/MPT caches to achieve better performance, whose size does not depend on the number of QPs but to the number of device contexts, memory regions (MRs), and memory windows (MWs).

According to internal performance counters collected by Mellanox Neo-Host [13], in a typical environment where all threads of the application share the same device context and thus the same MTT/MPT, the cache hit ratio of the MTT/MPT cache is higher than 95% in most cases. In contrast, the cache hit ratio of the MTT/MPT cache can drop to less than 70% when multiple device contexts are created because each device context allocates memory regions separately, even though the size of each memory region is only a few MB. Therefore, sharing the device context to reduce redundancy in global data structures is not only good for management but also for performance. Similar phenomena are also confirmed by other works [5, 30].

Outstanding Work Requests (OWRs) are enqueued work requests whose completion entries have not yet been polled from the CQ. After receiving messages from remote QPs, the RNIC must retrieve metadata of the corresponding work request, which may reside in the on-chip WQE cache. A cache miss results in a PCIe DMA read, which is much more expensive [5, 41]. As a result, cache thrashing occurs when the number of outstanding work requests exceeds a certain threshold. However, fewer OWRs can also lead to performance degradation due to a lack of parallelism. As we will see later in §3.2, the proper configuration depends on the current overall parallelism of the system and should therefore be dynamically adjusted.

3 Scalability Bottlenecks

In this section, we use the micro-benchmark and the PCIe/RNIC performance counters to analyze the factors that affect the scalability of disaggregated applications. They also motivate the design of SMART. The setup of all evaluations in this section is consistent with §6.

3.1 Implicit Doorbell Contention

After our investigation, we speculate that the implicit contention in doorbell registers is the main limitation of parallelism in disaggregated applications. To validate this assumption, we compare three different types of allocation mechanisms that have been used by existing works and a novel method that is aware of the structure of doorbell registers:

1. Shared QP [20]: all threads share a single QP.
2. Multiplexed QP [16, 52, 53]: each QP is shared by q threads in a NUMA-aware manner.
3. Per-thread QP [1, 54, 57, 64]: each thread owns and uses a dedicated QP.

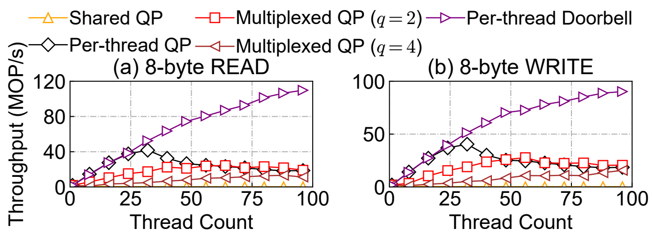


Figure 3. Throughput using different QP allocation policies.

4. Per-thread Doorbell (§4.1): each thread is associated with not only a separate QP, but also a separate doorbell register.

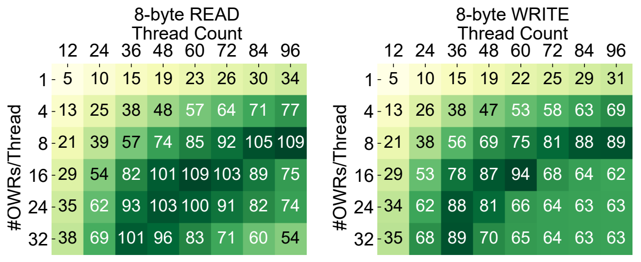
We build a bench tool to measure the throughput of READ/WRITE operations at any thread count and any concurrency depth k (i.e., OWRs per thread). Each thread repeatedly posts k work requests, rings the doorbell, and then waits for acknowledgments. The remote address of each work request is chosen uniformly randomly within a 1 GB memory region. Figure 3 shows the result of 8-byte READ/WRITEs with different thread counts and the same concurrency depth 8. The number of QPs is the main bottleneck when there are fewer than 32 threads. Per-thread QP and per-thread doorbell outperform other connection multiplexing policies by 2.4× ~ 130.1× in these cases. However, when the thread count exceeds 32, the throughput of the per-thread QP policy drops sharply. For example, the throughput of per-thread QP is cut in half after the number of threads is increased to 96.

In contrast, per-thread doorbell is able to unleash the potential of higher parallelism. For 8-byte READ requests, the throughput of per-thread doorbell can be up to 5.6×/3.2× higher than per-thread QP. The maximum throughput of per-thread doorbell can reach 110 MOPS. We have analyzed the execution overhead of the per-thread QP policy (in Figure 3, with a thread count of 96) using the Linux `perf` tool and Intel VTune Profiler [11]. Up to 74.05% of the total execution time is consumed by `pthread_spin_lock()` from doorbell-associated spin-locks. This suggests that doorbell-associated spinlocks incur significant overhead in a high parallelism environment for per-thread QP. Readers may also notice the throughput drops of per-thread doorbell at the tail of the curve, which is related to the cache trashing problem that will be discussed in the next section.

More details about the specific implementation of the per-thread doorbell are explained in §4.1. We also ran the same benchmark with a larger payload size (up to 64 bytes), and the same observation still holds. A workload with the payload size greater than 64 bytes is bandwidth-bound, not IOPS-bound.

3.2 Cache Thrashing

After removing implicit doorbell contentions, we still observe that throughput drops as the number of outstanding



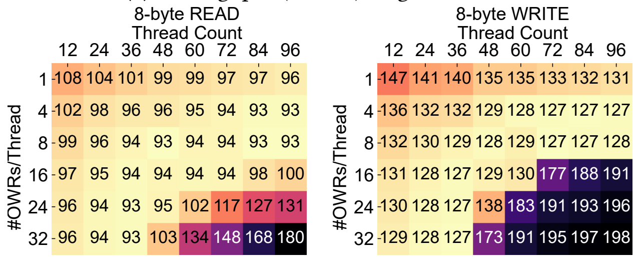
8-byte READ

#OWRs/Thread \ Thread Count	12	24	36	48	60	72	84	96
1	5	10	15	19	23	26	30	34
4	13	25	38	48	57	64	71	77
8	21	39	57	74	85	92	105	109
16	29	54	82	101	109	103	89	75
24	35	62	93	103	100	91	82	74
32	38	69	101	96	83	71	60	54

8-byte WRITE

#OWRs/Thread \ Thread Count	12	24	36	48	60	72	84	96
1	5	10	15	19	22	25	29	31
4	13	26	38	47	53	58	63	69
8	21	38	56	69	75	81	88	89
16	29	53	78	87	94	68	64	62
24	34	62	88	81	66	64	63	63
32	35	68	89	70	65	64	63	63

(a) Throughput (MOP/s). Higher is better.



8-byte READ

#OWRs/Thread \ Thread Count	12	24	36	48	60	72	84	96
1	108	104	101	99	99	97	97	96
4	102	98	96	96	95	94	93	93
8	99	96	94	93	94	94	93	93
16	97	95	94	94	94	94	98	100
24	96	94	93	95	102	117	127	131
32	96	94	93	103	134	148	168	180

8-byte WRITE

#OWRs/Thread \ Thread Count	12	24	36	48	60	72	84	96
1	147	141	140	135	135	133	132	131
4	136	132	132	129	128	127	127	127
8	132	130	129	128	129	127	127	128
16	131	128	127	129	130	177	188	191
24	130	128	127	138	183	191	193	196
32	129	128	127	173	191	195	197	198

(b) Average DRAM access traffic per work request (bytes). Lower is better.

Figure 4. Performance metrics with different thread counts and outstanding work requests (OWRs).

work requests increases. It is highly suspicious that this phenomenon is related to the cache thrashing problem discussed above. Since both the cache size and the replacement policy are confidential to the product vendors, we use the bench tool in §3.1 to characterize this phenomenon. Specifically, we use the per-thread doorbell policy to avoid doorbell contentions and then adjust both the number of threads (i.e., parallelism) and the number of OWRs (i.e., concurrency depth). We then measure the throughput of random 8-byte READ and WRITE requests for each setting. The specific depth of concurrency per thread is achieved by polling the completion queue after every k work requests are posted, and the results are shown in Figure 4a.

As we can see from the figure, a larger number of OWRs can lead to better throughput, because they can hide the network roundtrip latency. The maximum read IOPS is achieved with 96 threads × 8 OWRs per thread = 768 concurrent work requests. In contrast, less parallelism requires more concurrency to achieve similar throughput. For example, 32 outstanding work requests are required to achieve 101 MOPS for 36 threads, which requires 32 × 36 = 1152 concurrent work requests (i.e., 50% more concurrency, but still 5% less throughput). However, as the concurrency depth increases, the throughput decreases. With 96 reader threads, the throughput with 32 OWRs per thread is only 49.5% of 8 OWRs per thread.

With our experiments based on control variables, we show that different levels of parallelism require a correspondingly appropriate concurrency depth to achieve the best throughput. We confirmed the increase in cache misses in the RNIC by measuring the *PCIe inbound bandwidth*, i.e., the traffic of the RNIC's access to the host DRAM. Figure 4b shows the average DRAM access traffic per work request for different thread counts and outstanding work requests. With 96 reader threads and 32 OWRs per thread, it is 1.9× higher than with 8 OWRs per thread (180 bytes vs. 93 bytes). These extra PCIe reads are mainly caused by WQE cache misses. Comparing Figures 4a and 4b, we find that throughput decreases as the DRAM access traffic per work request increases. Other characterization studies have also observed a similar phenomenon [24, 26], but existing solutions typically assume that a fixed size of concurrency depth is sufficient to avoid this problem. However, even a small number of concurrency depth can become a problem in our scale-up scenario, which may have more than a hundred threads.

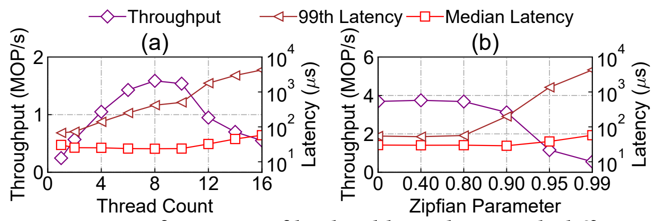


Figure 5. Performance of hash table updates with different (a) thread counts (concurrency depth = 8, Zipfian parameter = 0.99) and (b) Zipfian distribution parameters (16 threads).

3.3 Unsuccessful Retries

In addition to the sender-side contention with the observations above, receiver-side contention can also lead to scalability bottlenecks in disaggregated applications. According to our investigation, update contention, which leads to unsuccessful RDMA CAS retries, is the most common and important scalability bottleneck in receiver-side contention.

Specifically, without server-side coordination, disaggregated memory applications have to implement application-side coordination with one-sided RDMA operations, which is the main source of receiver-side contentions. For example, Sherman [57] points out that basic spinlocks using RDMA CAS suffer from the scalability problem: if the client thread fails to acquire a lock, it immediately retries, which leads to extra remote network accesses. This wastes the limited IOPS of RDMA networks. Sherman solves this problem by using a hierarchical on-chip lock (HOPL) technique.

However, in addition to lock-based methods, this scalability problem caused by unsuccessful retries exists in a wider range of applications, such as lock-free data structures [52, 68, 69] and optimistic locking [61]. Figure 5 illustrates the throughput of RACE [68, 69], a disaggregated lock-free hash table, with different parallelism and different Zipfian distributions [19]. As we can see from the figure, although the parallelism of RDMA operations can scale up to a much larger number of threads, the peak throughput of RACE is reached at only 8 threads. The throughput decreases and the 99% latency increases by up to 17.1× for more threads. We also investigate how data skewness affects the scalability of application operations. As the Zipfian parameter θ increases from 0 to 0.99, the median latency increases by 1.9× and the 99% latency increases by 78.4×.

According to our scale-up analysis, a major cause of such a scalability problem is the massive number of unsuccessful retries caused by high data contention. In RACE, after a failed RDMA CAS that attempts to update a bucket slot, a retry is triggered that initiates three more RDMA requests (i.e., re-read this bucket, write the key/value entry, and try to atomically update this slot again). This retry wastes the RNIC's limited IOPS resources. In contrast, the original RACE papers [68, 69] only evaluate up to 128 concurrent tasks distributed across four machines, and thus the above scalability problem is not significant. As we will discuss later in §6.3, 67.9% of update operations result in one or more unsuccessful retries. The notification mechanism is not applicable in our environment because it requires the involvement of remote CPUs or specialized hardware such as SmartNICs [57].

4 Design

Motivated by the analysis of scale-up bottlenecks, we design Smart, a library for building scalable disaggregated applications. We introduce the following techniques used in Smart:

1. *Thread-aware resource allocation* (§4.1). Smart avoids implicit inter-thread contention from both queue pairs and doorbell register sharing.
2. *Adaptive work request throttling* (§4.2). Smart dynamically controls the concurrency depth to avoid RNIC cache thrashing. The threshold is set dynamically on the fly.
3. *Conflict avoidance* (§4.3). Smart uses truncated exponential backoff to reduce the rate of failed retries. It further controls the concurrency depth according to the retry rate.

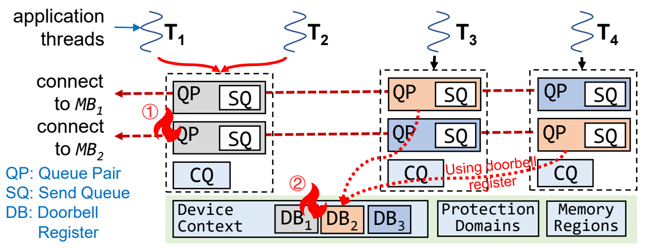
More importantly, the details of these techniques are hidden from the easy-to-use interface of Smart, which provides a set of coroutine-based asynchronous APIs that are similar to the original one-sided RDMA verbs.

4.1 Thread-aware Resource Allocation

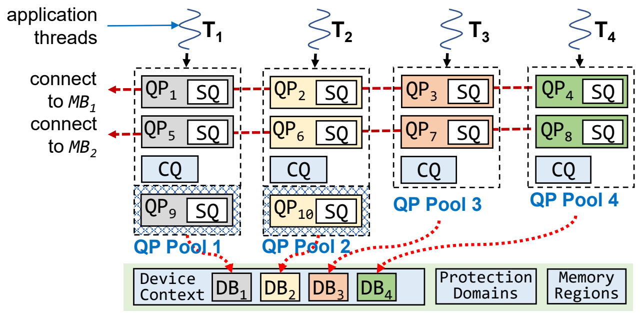
To avoid thread contention due to resource sharing (as shown in Figure 6a), besides allocating QPs on a per-thread basis [1, 54, 57, 64], Smart proposes thread-aware resource allocation. In Smart, multiple QPs used by the same thread are associated with a shared doorbell register[3].

A simple approach is to create separate RDMA device contexts per thread [36, 57]. This essentially avoids implicit thread contention because different contexts use different sets of doorbell registers. However, as discussed in §2.2, sharing device contexts is not only good for management but also helpful for performance. When using multiple device

[3]The number of threads does not exceed the number of CPU cores.



(a) Two kinds of potential RDMA resource contentions: ① multiple threads share the same QP (T_1 and T_2), ② multiple threads use different QPs, but internally share the same doorbell register (T_3 and T_4).



(b) Allocate RDMA resources in a thread-aware manner to avoid implicit contentions.

Figure 6. Thread-aware resource allocation. There are four threads (T_1 to T_4) in a compute blade that connect to memory blades (MB_1 and MB_2).

contexts, the local memory must be registered as MRs and MWs per context. The number of MRs/MWs is tens or even hundreds of times higher, which increases the size of MPT/MTT and causes performance degradation due to MPT/MTT cache thrashing [41]. In addition, the sharing semantics of `ibv_open_device()` depend on the specific implementation of device vendors. It may not return a different context for each call [2].

In contrast, Smart proposes the thread-aware allocation mechanism that shares the same device context, which is the common practice of application developers. As shown in Figure 6b, Smart allocates QPs, CQs, and DBs in a per-thread manner, and these resources are not shared by multiple threads. Other resources, including protection domains (PDs) and memory regions (MRs), are shared among all threads, thus the local memory is registered only once. A QP is created for each pair of a local thread and a remote memory blade. Thus each thread contains multiple QPs (for different remote blades), but they are associated with the same CQ and DB. Smart uses a single `ibv_poll_cq()` call to determine if there are any completed work requests posted by the current thread. QPs are also associated with the DB that belongs to the same thread, so thread contention from sharing DBs is avoided. Coroutines running in the same thread share the same QPs and CQ.

More specifically, Smart maintains a *QP pool* for each thread, where all the QPs in the same pool are all associated with the same CQ and DB. Some QPs are active (e.g., QP_1 and QP_5 in QP pool 1), while others are idle (e.g., QP_9). Each thread allocates QPs only from its own QP pool and releases them to its own QP pool after use.

To construct enough QP pools during application startup, the number of **medium-latency** doorbell registers available should not be fewer than the number of threads, but it is only 12 by default. The RNIC driver supports tuning the number of doorbell registers per RDMA context, e.g., the environment variable `MLX5_TOTAL_UUARS` for Mellanox products. However, the driver also limits the number of medium-latency doorbell registers, so a minor modification to the driver code is required. The maximum valid number of DBs is 512 in the Mellanox ConnectX-6 RNIC [12], 256 in the Intel E810 RNIC [22], which are typically larger than the number of CPU cores per machine[4]. This limitation also suggests that a one-to-one mapping between DBs and QPs is not practical, since the number of QPs can be much larger than 512.

On the other hand, since the structure of doorbell registers is not exposed to the developers, there is no external API that can be used to bind a QP with a specific doorbell register. However, after digging into the source code of the RNIC user-mode driver, we found that each newly created QP is associated with doorbells in a round-robin fashion (Figure 2b as an example). Therefore, before creating a QP, we can know which doorbell register it will be associated with. It can be used to determine the associated QP pool. Different providers may have different DB allocation policies, but our mechanism is possible as long as it is deterministic. We also advocate that the provider provide **explicit DB binding APIs**, as they are critical to performance.

Resource Allocation in Memory Blades. Because an RC connection requires QP-to-QP communication, a memory blade needs to establish as many QPs as the total thread count of compute blades. We confirm that the outbound throughput does not drop as the QP count increases [5]. Thus, there is no need to allocate RDMA objects on a per-thread basis as long as memory blades never post RDMA requests or poll CQ entries.

4.2 Adaptive Work Request Throttling

To prevent performance degradation due to RNIC cache thrashing, Smart uses *credit-based throttling* [41] to limit the number of outstanding work requests of a QP. We lack information about cache replacement policies used by specific RNICs, so Smart proposes a heuristic mechanism. As discussed in §3.2, once the number of threads is determined, performance degradation can be avoided by limiting the *upper bound* of OWRs. We also assume that the application workload remains stable over a short period of time (about 500 ms), so that the optimal threshold can be updated periodically.

[4]If DBs are insufficient, QPs used by multiple CPUs can associate with the shared doorbell register.

Algorithm 1 Work Request Throttling

```
1: thread_local $C_{\max}$, $credit \leftarrow C_{\max}$
2: procedure SmartPostSend($qp, wr, bad\_wr$)
3:     $size$ = Length($wr$)                       ▷ length of the wr linked list
4:     $wr[size-1].wr\_id \leftarrow \langle size, \ldots \rangle$      ▷ fill metadata into wr_id
5:     while $credit - size < 0$ do
6:         Wait                                  ▷ defer posting unless credit is enough
7:     $credit \leftarrow credit - size$                 ▷ deduct the credit
8:     $ibv\_post\_send(qp, wr, bad\_wr)$           ▷ bad_wr returns failing WR
9: procedure SmartPollCQ($cq, num\_entries, wc$)
10:    $ibv\_poll\_cq(cq, num\_entries, wc)$
11:    for each $e$ in $wc$ do
12:        $\langle size, \ldots \rangle = e.wr\_id$              ▷ extract metadata from wr_id
13:        $credit \leftarrow credit + size$                ▷ replenish the credit
14: procedure UpdateCMax($target$)
15:    $credit \leftarrow credit + (target - C_{\max}), C_{\max} \leftarrow target$
16: procedure Update
17:    $target\_list = [4, 6, 8, 10, 12]$            ▷ candidate values of $C_{\max}$
18:    $P_{opt} = -1, target_{opt} = -1$
19:    for each $target$ in $target\_list$ do
20:        UpdateCMax($target$)
21:        $P$ = number of completed RDMA work requests for $\Delta$ milliseconds
22:        if $P > P_{opt}$ then
23:            $P_{opt} \leftarrow P, target_{opt} \leftarrow target$
24:    UpdateCMax($target_{opt}$)
```

Throttling. Based on the above observations, we design an adaptive throttling algorithm (Algorithm 1 Lines 1–13). The thread-local variable credit indicates the number of available credits. It is initially set to $C_{\max}$, which is the maximum credit. We use the work request id field (`wr_id`) to indicate the number of work requests to submit (Line 4). Before issuing `ibv_post_send()`, the credit is subtracted from size (Line 7). However, if the credits are depleted, posting work requests are throttled until the credits are replenished (Line 6). If the CQ poll is successful, the credit is increased by the number of completed work requests (Line 13).

Update Available Credits. According to our observations (§3), the optimal maximum credit $C_{\max}$ depends on both the thread count and the workload. Smart updates $C_{\max}$ periodically during application execution. We follow the epoch-based model, i.e., each epoch consists of an *update phase* and a *stable phase*. The application runs normally whatever phase it is in. To increase or decrease the available credits, Smart uses the UpdateCMax function, where target is the new value of $C_{\max}$ (Line 15). During the update phase, Smart calls Update (Lines 14–24 in Algorithm 1) to find the optimal value of $C_{\max}$. For each candidate value of $C_{\max}$, it changes the available credits (Line 20) and then counts the number of completed RDMA operations for the next $\Delta = 8$ milliseconds. Such an interval allows the application to generate a sufficient number of RDMA requests to accurately estimate the throughput of RDMA requests. At the end of the update phase, Smart determines the optimal target that achieves the highest throughput, and updates the available credits (Line 24). $C_{\max}$ does not change during the stable phase. In

SMART, the stable phase lasts longer, i.e., $T = 60 \times \Delta = 480$ milliseconds.

4.3 Conflict Avoidance

After dealing with the scale-up bottlenecks at the sender side, contention at the receiver side, especially the wasted IOPS caused by unsuccessful CAS operations, leads to degraded application performance. Our solution to this problem is based on exponential backoff [39] with an adaptive limit.

Truncated Exponential Backoff. The basic idea of exponential backoff is simple: once an unsuccessful retry is detected (e.g., an RDMA CAS fails), we delay the next retry by t CPU cycles. Thus, each time an unsuccessful retry is detected, the delay time is multiplied by two, i.e., $t = t_0 \times 2^i$ in the i-th retry. To avoid extremely long latencies, which make the performance even worse [32], SMART uses the *truncated* variant of exponential backoff by limiting the maximum backoff cycle $t_{\max}$. To avoid collisions, the backoff cycles are also randomized. In summary, for the i-th attempt,

$$t = \min\{t_0 \times 2^i, t_{\max}\} + Rand(t_0), \tag{1}$$

where $Rand(x)$ is a random value between 0 and x. In SMART, the backoff unit is $t_0 = 4096$ cycles, which is close to the time of an RDMA roundtrip. $t_{\max}$ is determined dynamically using the algorithm below.

Dynamic Backoff Limit. The maximum backoff cycle, i.e., $t_{\max}$, has a significant impact on performance. A smaller $t_{\max}$ increases the probability of collisions, which degrades scalability. However, a larger one also leads to lower performance, especially if some of the operations suffer from longer latency. The optimal $t_{\max}$ depends on the number of concurrent operations. We use an algorithm to automatically find the optimal $t_{\max}$. Every millisecond, SMART collects and statistics the percentage of retries for all operations. We call it the *retry rate*, or γ. If γ is greater than the high-water mark γ_H, $t_{\max}$ is multiplied by two. If γ is less than the low-water mark γ_L, $t_{\max}$ is divided by two. We make sure that $t_{\max}$ is between t_0 (the backoff unit) and t_M (the longest backoff cycles, $t_M = 2^{10} \times t_0$ by default). In our prototype, we choose $\gamma_H = 0.5$ and $\gamma_L = 0.1$, which help $t_{\max}$ converge to different values depending on the workload skew. For example, $t_{\max} = t_M = 1.6ms$ for skewed (high contention) updates, while $t_{\max} = t_0$ for uniform updates (i.e., retries are rare).

Concurrency Depth Throttling. To further reduce conflicts from concurrent operations performed by the same thread, SMART reduces the number of concurrent operations through *credit-based* coroutine throttling. This essentially controls the number of concurrent tasks because SMART is implemented as a coroutine-based asynchronous framework. For example, under uniform workloads, a coroutine suspends its execution (*yield*) when it waits for RDMA ACKs. However, under high contention workloads, a coroutine does not suspend until the current operation has been completed. To achieve this, the scheduler must keep at most $c_{\max}$ coroutines that can be executed concurrently for each thread. Other coroutines must be blocked until some running coroutines have completed their current operations and replenished their credits. Similar to $t_{\max}$, the value of $c_{\max}$ is dynamically determined according to the retry rate. We shrink or expand $c_{\max}$ when γ is above γ_H or below γ_L. Note that $t_{\max}$ is only updated if $c_{\max}$ reaches the lower or upper bound (e.g., the abort rate $\gamma > \gamma_H$ and $c_{\max} = 1$).

5 Implementation

We implement SMART in C++, with the core library consisting of about 3,000 lines of code.

5.1 Programming Interface

SMART provides the following main APIs:

1. `connect`: connect to a memory blade;
2. `read`, `write`, `faa`, `cas`: append a work request (WR) to the local buffer;
3. `post_send`: post WRs in the local buffer to the RNIC;
4. `sync`: wait for all previously posted WRs to complete;
5. `backoff_cas_sync`: perform *a* CAS operation with exponential backoff optimization.

When a user connects to a memory blade, SMART internally allocates RDMA resources in a thread-aware manner (§4.1). SMART wraps up RDMA verbs to support adaptive work request throttling (§4.2). Specifically, it maintains thread-local work request buffers. Developers use verb functions such as `read()` that add new work requests to the buffer. Then, developers use `post_send()`, which posts buffered work requests to the RNIC, which essentially calls SMARTPOSTSEND() (Algorithm 1). Finally, when `sync()` is called, the current coroutine is suspended until all CQ entries are received by SMART, which executes SMARTPOLLCQ() (Algorithm 1) internally. This suggests that SMART absorbs the backpressure by internal stalling. For conflict avoidance (§4.3), SMART provides the `backoff_cas_sync()` API. Its semantics are the same as the combination of the `cas()` and `sync()` APIs. If CAS fails, it will also delay a few CPU cycles and configure the concurrency depth before returning to the user code. This allows applications to change the *expected* value.

Coroutines are used in SMART to achieve asynchronous programming and increase the concurrency depth of each thread. Similar to FORD [64] and other disaggregated memory applications, SMART uses Boost's coroutine engine [42]. An application creates multiple threads, each of which spawns multiple coroutines. All coroutines from the same thread share the same QPs, CQ, and DB. They will be assigned to each thread accordingly (§4.1). SMART also uses a dedicated coroutine for each thread to poll CQs.

5.2 Applications

We also implement three typical and important IOPS-bound applications for disaggregated memory to demonstrate the scalability of SMART.

Lock-free Hash Table. **SMART-HT** (44 out of 1,078 lines of code changed to refactor the original application with

SMART) uses the same hashing scheme as RACE [68, 69]. Since the RACE code is not publicly available, we implement our own version of RACE from scratch.

Persistent Transaction Processing. **SMART-DTX** (16 out of 3,018 lines of code changed) is derived from FORD [64] using techniques from SMART. For each thread, SMART-DTX uses only one QP for each connection to a memory blade.

Lock-based B$^+$ Tree. **SMART-BT** (48 out of 1,210 lines of code changed) borrows the main idea of Sherman [57], but we do not use the *two-level version* mechanism, because we found that our RNIC may *not* read or write data in increasing address order. SMART-BT retrofits the per-cacheline version mechanism introduced by FaRM [16]. It's safe to update a key/value entry belonging to a single cacheline without changing the version field because data atomicity per cacheline is guaranteed by RNIC.

Our profiling also shows that the original implementation of Sherman leads to bandwidth-bound memory access patterns due to its read amplification problem. Since compute blades do not know the address of the target key/values, Sherman must fetch the *entire* leaf node (larger than 1 KB) from remote memory and then look up the requested key/-value pair. We propose *speculative lookup*, an optimization that solves this problem and thus makes SMART-BT IOPS-bound. Each compute blade stores a small cache mapping from the key to the data address in the remote. This allows a lookup operation to first take a fast path by using the cached address to fetch the data item (a small-sized READ request). If the fast path succeeds, the lookup operation returns immediately. Otherwise, either because the key is not cached or the fetched data item is not valid (by an application-level validator), the lookup falls back to the regular lookup. A detailed breakdown analysis is given in §6.2.3 to demonstrate the speedup of speculative lookup and SMART separately.

6 Evaluation

We evaluate SMART to answer the following questions:

- What performance does SMART achieve for different types of disaggregated applications, and how does each technique contribute to the overall performance? (§6.2)
- How effective is SMART in reducing resource contention? (§6.3)

6.1 Evaluation Setup

We use eight machines in our cluster, each with two Intel Xeon Gold 6240R CPUs (96 cores in total), 384 GB DRAM (32 GB×12), 1.5 TB Intel Optane DC Persistent Memory (128 GB×12)[5], and a 200 Gbps Mellanox ConnectX-6 InfiniBand RNIC. Each RNIC is connected to a 200 Gbps Mellanox InfiniBand switch. The hardware limit of the RNIC on our test platform is 110.0 MOP/s. These machines are installed with Ubuntu 20.04 LTS (Linux kernel 5.4.0) and Mellanox OpenFabrics Enterprise Distribution for Linux (`MLNX_OFED`) v5.3-1.0.0.1. For each experiment, some machines in the cluster are used to emulate memory blades. 2 MB huge pages are used to reduce page translation cache misses from RNICs. We also enable the memory interleave policy. Unless otherwise stated, the concurrency depth of each thread is 8 by using coroutines to maximize the throughput.

6.2 End-to-end Performance

6.2.1 Hash Table.

In this section, we report the performance results of both SMART-HT and RACE hashing [68, 69].

Workloads. To be consistent with the existing works of disaggregated indexes [57, 68, 69], we use YCSB [8] to evaluate the performance of different hashing indexes. Specifically, three types of read-write ratios are reported:

1. **Write-heavy**: 50% updates and 50% lookups,
2. **Read-heavy**: 5% updates and 95% lookups, and
3. **Read-only**: 100% lookups.

The queried keys follow the Zipfian distribution [19] (with the skewness parameter $\theta = 0.99$), which is more common in production environments [8]. Each record in the index consists of an 8-byte key and an 8-byte value. For each experiment, we load 100 million items into the hash table and then run the corresponding workloads. Unless otherwise noted, all hash table experiments are performed with one compute blade and two memory blades.

Scalability. Figure 7 shows the throughput of RACE and SMART-HT with different thread counts. In Figure 7(a)–(c), only one compute blade is running. Overall, SMART-HT has higher throughput than RACE because it has better scalability. For write-heavy workloads, the highest throughput of RACE is 2.8 MOP/s (with 8 threads), while SMART-HT is 5.7 MOP/s (with 48 threads). For read-heavy and read-only workloads, we find that the throughput of RACE is less than 11.4 MOP/s, while SMART-HT reaches 21.2 MOP/s and 23.7 MOP/s respectively. Each lookup operation requires three RDMA READs. With 64 or more threads, SMART-HT cannot improve the throughput any further because the bandwidth resources of RNIC/PCIe are exhausted.

We also evaluated the scale-out cases as shown in Figure 7(d)–(f), using up to 6 compute blades. Each compute blade runs 96 threads, so there are up to 6 × 96 = 576 concurrent threads in an execution. For both write-heavy and read-heavy workloads, SMART-HT outperforms RACE by up to 132.4× and 77.3× respectively, due to the higher contention ratio caused by the higher concurrency. For read-only workloads, although RACE's throughput increases with the number of compute blades, SMART-HT still has 2.0× ~ 3.8× higher throughput.

Performance Breakdown. To further understand the results, we break down performance gaps between RACE and SMART-HT by applying each technique in turn: (1) `+ThdResAlloc`

[5]Optane DC Persistent Memory is used only for evaluating SMART-DTX with FORD [64]. Both DRAM and persistent memory DIMMs are properly installed in both sockets, and the memory controllers are fully utilized.

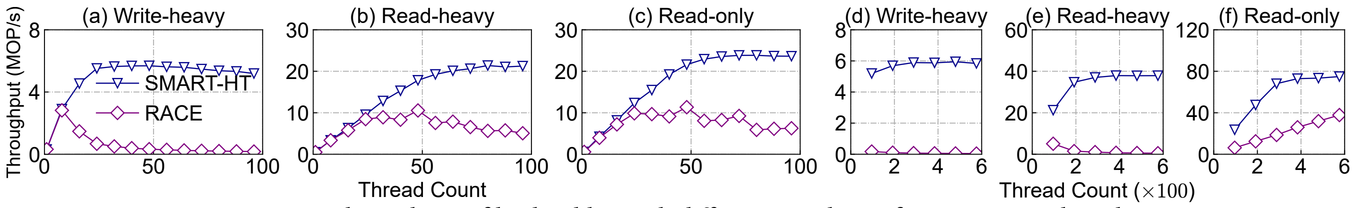


Figure 7. Throughput of hash tables with different numbers of computation threads.

(thread-aware resource allocation); (2) +WorkReqThrot (adaptive work request throttling); and (3) +ConflictAvoid (conflict avoidance). The results are shown in Figure 8.

ThdResAlloc removes implicit doorbell contentions. With a higher ratio of lookup operations, such as the read-only workloads, thread contention in doorbell registers is the main factor that degrades the application performance. Enabling thread-aware resource allocation is very effective in improving application scalability in these read-heavy scenarios. As mentioned earlier, since the maximum number of doorbell registers can be much smaller than the number of QPs, a thread-aware allocation mechanism is required, rather than simply increasing the number of doorbell registers.

WorkReqThrot throttles down the number of outstanding work requests. This technique is effective for write-heavy workloads when the thread count is between 8 and 32. Introducing work request throttling increases the IOPS of RDMA messages, as we will demonstrate later in §6.3. This also increases the throughput of index operations. However, as the thread count and the network IOPS continue to increase, an update operation is more likely to contend with others, making unsuccessful retries more common. WorkReqThrot also allows applications to create many more coroutines (higher concurrency depth) without suffering from performance degradation caused by cache thrashing.

ConflictAvoid reduces unsuccessful retries through the exponential backoff mechanism. For write-heavy workloads with skewed key distribution, unsuccessful retries play a dominant role in wasted IOPS. Even for read-heavy workloads, where only 5% of updates occur, there are still massive failed retries when the thread count exceeds 64. By enabling conflict avoidance, the hash index achieves optimal throughput and scalability in both workloads.

For Figure 7(d)–(f), multiple compute blades are involved during execution. For write-heavy and read-heavy workloads, the most important optimization to improve the scalability of RACE is still ConflictAvoid. For read-only workloads, there is still a throughput gap (up to 3.8×) between RACE (essentially disabling ThdResAlloc) and SMART-HT. RACE's throughput increases because the doorbell registers are not shared across multiple compute blades.

Throughput vs. Latency. We examine the latency-throughput correlation in both SMART-HT and RACE, with 96 threads running in each. We control the throughput by intentionally throttling execution. The latency-throughput curve is shown in Figure 9. Note that the maximum throughput of SMART-HT is less than 23 MOP/s, which is caused by the additional overhead of measuring the execution time of each operation. Compared to RACE, SMART-HT reduces the median latency (i.e., the 50th percentile latency) by 69.6%. By enabling thread-aware resource allocation, the median latency is less than 30.3μs, because this technique prevents a thread that is about to post a work request from being blocked by other threads. The minimum median latency is 11.2μs, achieved at a throughput of 12.2 MOP/s. SMART-HT also reduces the tail latency (i.e., 99th percentile latency) by up to 80.6% of RACE.

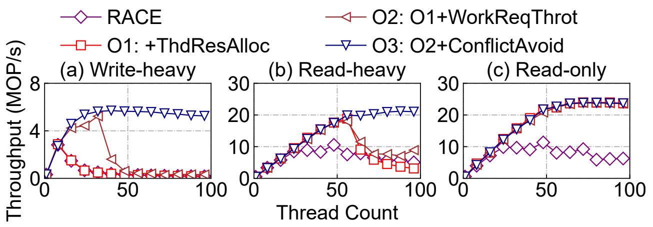


Figure 8. Performance breakdown of hash tables.

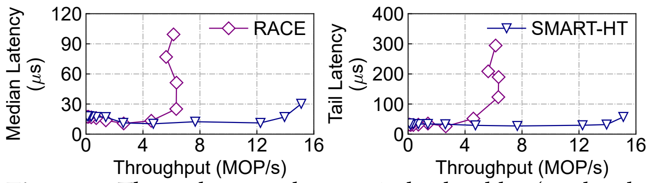


Figure 9. Throughput vs. latency in hash tables (read-only workload).

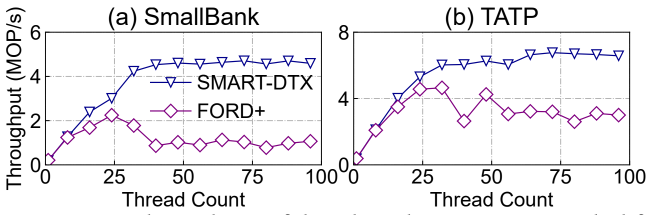


Figure 10. Throughput of distributed transactions with different numbers of executing threads.

6.2.2 Persistent Distributed Transaction.

We compare SMART-DTX with FORD [64]. For a fair comparison, we do not use dedicated QPs to perform asynchronous undo logging in FORD. The modified version is FORD+ which outperforms the original FORD in throughput.

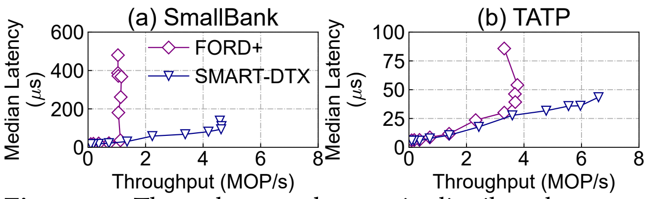


Figure 11. Throughput vs. latency in distributed transactions.

Workloads. We use two OLTP benchmarks to evaluate the end-to-end performance of distributed transactions: (1) *SmallBank* [50], which simulates bank account transactions, and 85% of transactions are read-write; and (2) *TATP* [49], which models a telecom application, consisting 80% of read-only transactions. Two memory blades are used for each test, and database records are stored in NVM as FORD. We run 6 million transactions in each benchmark and count the number of *committed* transactions per second. Latency is the time it takes to commit a transaction.

Scalability. Figure 10 shows the throughput of both FORD+ and Smart-DTX. FORD+ achieves the highest throughput with 24 and 32 threads for the SmallBank and TATP benchmarks respectively. However, FORD+ suffers from performance degradation with more threads. For example, the throughput of SmallBank is only 1.0 MOP/s with 96 threads running. This is mainly due to implicit doorbell contention. Readers may notice that the throughput of 48 threads for FORD+ is higher than both 40 and 56 threads for TATP. This outlier is also related to the default doorbell register allocation policy. With 48 working threads, there are at most 4 threads using QPs associated with the same doorbell register, but with 47 or 49 working threads, there are 8 threads that can interfere with each other. The same scenario occurs with 72 and 96 threads.

According to our breakdown analysis, the improvement of Smart-DTX is mainly due to the thread-aware allocation of RDMA resources. In contrast to disaggregated data structures, committing a distributed transaction requires more RDMA messages. As a result, resource contention during RDMA message delivery can lead to significant performance degradation. Smart-DTX outperforms FORD+ by up to 5.2× in SmallBank, and 2.6× in TATP. Other techniques, such as work request throttling and conflict avoidance, also slightly improve the performance of SmallBank. Committing a transaction requires multiple READ/WRITE operations, depending on the read and write sets. The maximum throughput of Smart-DTX is bounded by either network IOPS (consume up to 90 MOP/s in SmallBank) or bandwidth (consume up to 104 Gbps in TATP[6]).

Throughput vs. Latency. Similar to Smart-HT, we also examine the relationship between the median latency and the throughput. As shown in Figure 11, there are $96 \times 8 = 768$ concurrent tasks. In addition to improving maximum throughput, Smart-DTX dramatically reduces the median latency of SmallBank and TATP by up to 45.8% and 77.0%, respectively, thanks to thread-aware resource allocation. When the throughput of Smart-DTX and FORD+ is less than 0.8 MOP/s on the SmallBank benchmark, both have similar median latency. The same observation holds for TATP.

6.2.3 B$^+$Tree.

In this section, we compare Smart-BT with Sherman [57]. The open-source version [56] may crash with massive threads, so we fix it by using the per-cacheline version mechanism (§5). We call the modified version Sherman+.

Workloads. We use the same workloads as described in §6.2.1. To keep the setup consistent with the original paper [57], we emulate each server as both a memory blade (using 2 threads) and a compute blade (using a maximum of 94 threads).

Scalability. Figure 12 shows the throughput of Sherman+ and Smart-BT at different thread counts. In Figure 12(a)–(c), only one server is involved. For write-heavy workloads, update operations dominate the execution time, and the throughput of Smart-BT is slightly higher than that of Sherman+. For read-heavy and read-only workloads, Smart-BT outperforms Sherman+ by 1.8× and 2.0× respectively (using 94 threads). Figure 12(d)–(f) shows the throughput with multiple blades. Each compute blade is running 94 threads. With 6,016 coroutines, Smart-BT outperforms Sherman+ by 1.1× and 2.0× in read-heavy and read-only workloads respectively.

The performance improvements of Smart-BT in both read-heavy and read-only workloads are firstly contributed by *speculative lookup*. We evaluate the scalability of Sherman+ with Speculative Lookup (i.e., Sherman+ w/ SL), as shown in Figure 12(a)–(c). Sherman+ w/ SL has up to 1.6× higher throughput than Sherman+ under read-heavy workloads. By reading a single key/value entry instead of the entire leaf node, high read amplification is mitigated. This allows RNICs to achieve higher throughput when processing RDMA messages. As a result, applications get more benefits from speculative lookups with higher lookup ratios and higher lookup skewness.

However, we find that Sherman+ w/ SL (which becomes an IOPS-bound rather than bandwidth-bound application) does not scale up well with 64 and more threads. Under read-only workloads, the throughput of Sherman+ w/ SL is only 16.3 MOP/s when using 94 threads. Smart's techniques, especially thread-aware resource allocation, avoid the performance penalty of implicit doorbell sharing. The remaining techniques, such as adaptive work request throttling, have almost no impact on overall performance. For write-heavy workloads, the HOCL technique proposed by Sherman has reduced the number of RDMA messages when acquiring a spinlock.

[6]Our evaluation platform only supports PCIe 3.0, and the maximal bandwidth is 128 Gbps.

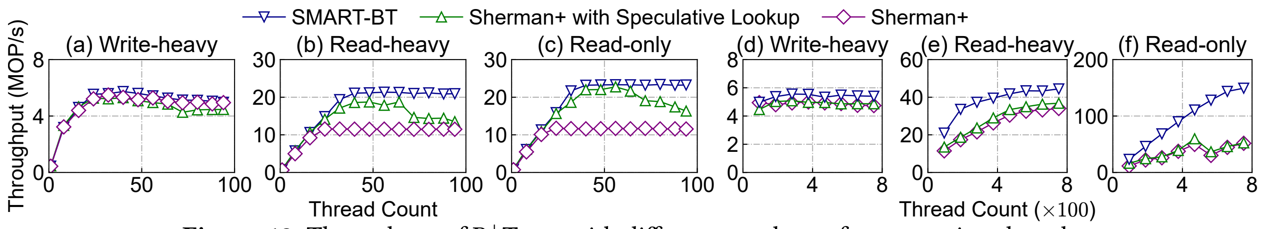


Figure 12. Throughput of B$^+$Trees with different numbers of computation threads.

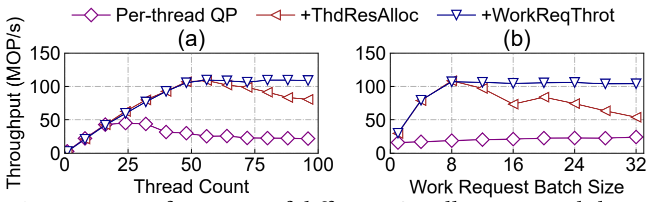


Figure 13. Performance of different QP allocation and throttling techniques: (a) different thread counts, work request batch size = 16; (b) different batch sizes, thread count = 96.

6.3 Micro-Benchmarks

In this section, we evaluate how Smart's techniques are effective in reducing resource contention.

Thread-aware Resource Allocation. To demonstrate that Smart avoids resource contention, as shown in Figure 13, we use the bench tool (§3.1) and measure the throughput of 8-byte READs using different resource allocation policies. In Figure 13(a), we keep up to 16 OWRs for each QP. Per-thread QP cannot scale to 24 or more threads (§3). By enabling thread-aware resource allocation (+ThdResAlloc), the throughput reaches up to 110.0 MOP/s, which is the hardware limit of the RNIC on our test platform. It also outperforms per-thread QP in throughput by 1.0× ~ 4.3×.

Adaptive Work Request Throttling. By enabling adaptive work request throttling (i.e., +WorkReqThrot), as shown in Figure 13(a), the throughput remains stable at 56 threads and above, and throughput degradation is avoided. Quantitatively, the throughput is up to 5.0× and 1.9× higher than per-thread QP and per-thread context respectively. This is effective by controlling the send queue depth and avoiding cache thrashing.

To investigate how outstanding work requests affect the throughput, Figure 13(b) shows the throughput with different work request batch sizes. As the batch size is greater than 8, +WorkReqThrot achieves the highest throughput of all the configurations. This shows that Smart avoids performance degradation due to massive outstanding work requests.

To evaluate the performance under dynamic workloads, we extend the benchmark so that it changes the number of concurrent threads at regular intervals (32 ~ 2048 ms). The range of concurrent threads varies between 36 and 96, and the work request batch size is 64. Based on the analysis in Section 3.2, we expect the throughput to be close to 110.0 MOP/s after enabling adaptive work request throttling. Table 1 shows the throughput of 8-byte RDMA READ. If the changing interval is longer than the epoch duration (512 ms by default), the throughput is nearly maximized with minor performance oscillations. In other cases, throughput decreases by up to 13.0% because the workload may not be stable within an epoch, but the optimal maximum credit $C_{\max}$ does not change. However, the adaptive work request throttling technique can still provide performance gains even when the workload changes frequently.

Table 1. Throughput (MOP/s) of 8-byte RDMA read operations under dynamically changing workloads.

Changing interval (ms)	32	64	128	256	512	1024	2048
w/o WorkReqThrot	79.3	79.2	77.9	74.7	73.8	71.8	65.7
w/ WorkReqThrot	97.8	95.1	99.7	101.0	105.8	109.3	109.6

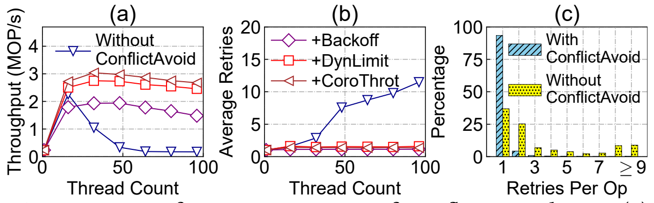


Figure 14. Performance metrics of conflict avoidance: (a) throughput; (b) average retries per operation; (c) distribution of retry counts per operation with 96 threads.

Conflict Avoidance. To reveal the impact of conflict avoidance, we measure the number of retries per operation (i.e., retry count) and report the result of Smart-HT with 100% updates. The evaluation setup is consistent with §6.2.

Figure 14(a) shows the scalability with different conflict avoidance configurations, and Figure 14(b) shows the average number of retries. Note that the queried keys follow the Zipfian distribution with $\theta = 0.99$. Disabling conflict avoidance causes significant performance overhead on massive concurrent threads, and the average number of retries also increases dramatically. This is because update operations are more likely to be interrupted by other threads. We also evaluate Smart-HT with different conflict avoidance settings. Enabling exponential backoff only (+Backoff) keeps the average retries per operation below 1.7, and the throughput does not decrease significantly as the thread count increases. Dynamical backoff limit (+DynLimit) increases the throughput by 1.6× of +Backoff. Enabling all techniques, including coroutine throttling (i.e., +CoroThrot), improves the throughput by up to 67% of +Backoff.

Figure 14(c) shows the distribution of the retry counts for an update operation. With 96 threads, the average number of retries without conflict avoidance is 11.5 per update, while Smart-HT is only 1.1. 93.3% of updates in Smart do not involve extra roundtrips or messages.

7 Related Work

Memory Disaggregation. In addition to the disaggregated data structures [35, 57, 68] and transaction processing systems [62, 64, 65] discussed in this paper, current work also includes the following categories: 1) Disaggregated memory management [1, 21, 34], which provides a transparent and efficient shared memory abstraction like local DRAM. 2) Disaggregated application runtime [37, 44], which allows developers to manipulate data placement in a fine-grained manner. 3) Disaggregated key/value store [33, 47, 52], which enables better utilization of data storage. Smart can be applied to all the above types of disaggregated memory applications.

As RNICs are commercially available in datacenters, most disaggregated applications are based on RDMA [1, 20, 52, 54, 57, 64, 68]. Our work is aimed at RDMA-based applications. Other protocols, like CXL [18, 23] and Ethernet [44], are also used in memory disaggregation.

SmartNIC. SmartNICs offload some network functions to on-chip computation powers. For example, Microsoft's Catapult [43] and StRoM [48] rely on FPGA-based SmartNICs to offload application-level kernels. RDMA is supported by most SmartNICs, both one-sided and two-sided verbs. The doorbell contention and cache thrashing issues are general because of the hardware implementation limitations. The unsuccessful retries issue can get help from SmartNIC because we can harness the application-level semantics to mitigate this problem in SmartNIC.

Scalability of RDMA Network. Improving the scalability of RDMA connections has been a long-standing challenge. Kalia et al. [26] and Collie [31] find that RC READs with large WQE batch sizes cause performance degradation over InfiniBand and RoCEv2 networks. Smart's adaptive work request throttling addresses this issue. FaRM [16] proposes QP multiplexing to address the scalability issue. X-RDMA [36] allocates RDMA context resources on a per-thread basis. Similarly, LITE [53] implements QP sharing between multiple processes. Smart's thread-aware resource allocation performs better in hardware resource utilization.

HERD [25], FaSST [27] and eRPC [24] use unreliable connection (UD) for better scalability, because one QP can associate with multiple QPs in this QP transport type. However, UD does not support one-sided RDMA verbs, which limits the scope of its use.

Two RPC communication frameworks, ScaleRPC [5] and Flock [41], throttle the number of QPs that can be used at any given time. This prevents cache thrashing from RNICs on the RPC server side (memory blades). This is different from disaggregated memory where the memory nodes have near-zero compute resources and the cache thrashing happens in the compute blades.

Finally, Dynamically Connected Transport (DCT) [15] is a patented Mellanox technology that allows a QP to dynamically connect and disconnect from any remote node. However, switching a connection from one remote node to another often results in performance degradation [27].

RDMA-based Data Structures. The hashing index is widely used for fast lookup in distributed systems. Existing hashing schemes [16, 40, 61, 68, 69] could be implemented using only one-sided RDMA verbs. RACE [68, 69] is optimized for disaggregated memory by reducing network roundtrips. Tree-based indexing (e.g., B+Tree [57, 67] and adaptive radix tree [35]) supports range queries. FG [67] uses a B-link tree structure and distributes tree nodes across different servers. It is the first distributed sorted index that completely leverages one-sided RDMA verbs. Sherman [57] improves the scalability under skewed write workloads by proposing two-tier locking. As discussed before, Smart can be used to improve the scalability of these data structures as long as they are IOPS-bound.

RDMA-based Distributed Transactions. One-sided RDMA verbs are widely used in both transaction execution [4, 16, 45, 61] and data replication [63]. Smart is effective for improving the scalability of these systems by mitigating RDMA resource contentions. Applications that use both one-sided and two-sided RDMA verbs (e.g., DrTM+H [60]) can also benefit from Smart.

8 Conclusion

This paper presents our observations on the three major scale-up bottlenecks for IOPS-bound disaggregated applications and the corresponding general solutions. We also propose Smart, an easy-to-use RDMA programming framework that hides technical details from application developers. Our evaluation results show that typically less than 50 lines of code are sufficient to refactor the state-of-the-art disaggregated systems with Smart to achieve significant improvements in both operation throughput and latency.

Acknowledgements

We thank our shepherd Vasilis Gavrielatos and all the reviewers for their valuable comments and suggestions. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216, 61877035), Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), Tsinghua University Initiative Scientific Research Program, and Beijing HaiZhi XingTu Technology Co., Ltd. Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn) and Kang Chen (chenkang@tsinghua.edu.cn) are the corresponding authors.

A Artifact Appendix

A.1 Abstract

This appendix helps readers to reproduce all experiments in Section 3 and Section 6. In Section 3, there are 3 experiments (Figures 3, 4, and 5), each of them points out one of the scalability bottlenecks. In Section 6, there are 9 experiments (Figures 7–14 and Table 1) that compare Smart-refactored applications (Smart-HT, Smart-BT, and Smart-DTX) with the state-of-the-art disaggregated systems (RACE, Sherman, and FORD) respectively. We provide scripts to launch these experiments and plot the corresponding figures automatically. With the hardware and software described in Section A.3.2 and A.3.3, this artifact should reproduce all experimental results.

A.2 Artifact check-list (meta-information)

- **Program:** RACE, Sherman and FORD. They are included in this artifact.
- **Binary:** Source code and scripts included. To support unlimited medium-latency doorbell registers, we provide a precompiled modified `libmlx5.so` in this artifact. You can also build it from scratch.
- **Data set:** Synthetic data sets (YCSB, SmallBank and TATP) are provided along with the artifact.
- **Run-time environment:** This artifact relies on Linux OS and Mellanox MLNX_OFED 5.x RNIC driver. To reproduce Figure 4b, Mellanox Neo-Host needs to be installed. Root access may be required for both reserving huge pages and profiling.
- **Hardware:** Must be a cluster of multiple machines (8 machines to reproduce all experiments). For each machine, Mellanox InfiniBand RNIC (preferably ConnectX-6) must be installed. To evaluate Smart-DTX and FORD under persistent memory, Intel Optane DC Persistent Memory (1st Gen or newer) is also required.
- **Run-time state:** The test cluster cannot perform other operations that consume CPU and/or network resources at the same time.
- **Execution:** Approximately 18 hours to evaluate all the benchmarks.
- **Metrics:** Number of operations processed per second (throughput), and elapsed time to complete an operation (latency).
- **Output:** Plots similar to the figures in the main paper.
- **Experiments:** Bash and Python scripts are provided to run the benchmarks and plot the corresponding figures respectively. Numerical variations in the results are negligible.
- **How much disk space required (approximately)?:** Approximately 143 MiB.
- **How much time is needed to complete experiments (approximately)?:** Approximately 18 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Workflow framework used?:** No
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.8376543

A.3 Description

A.3.1 How to access The artifact can be downloaded either from the GitHub link https://github.com/madsys-dev/smart or from the DOI link https://doi.org/10.5281/zenodo.8376543.

A.3.2 Hardware dependencies A cluster formed by multiple machines (8 machines to reproduce all experiments). Each of them requires Mellanox InfiniBand RNIC (preferably ConnectX-6), ≥ 96 cores and > 128 GB memory. To evaluate Smart-DTX and FORD under persistent memory, Intel Optane DC Persistent Memory (1st Gen or newer) is also required.

More specifically, this paper is based on the following environment:

- **CPU**: Dual-socket Intel Xeon Gold 6240R CPU (96 cores in total)
- **Memory**: 384GB DDR4 DRAM (2666MHz)
- **Persistent Memory**: 1.5TB (128GB*12) Intel Optane DC Persistent Memory (1st Gen) with DEVDAX mode
- **RNIC**: 200Gbps Mellanox ConnectX-6 InfiniBand RNIC. Each RNIC is connected to a 200 Gbps Mellanox InfiniBand switch

A.3.3 Software dependencies

- **RNIC Driver**: Mellanox OpenFabrics Enterprise Distribution for Linux (MLNX_OFED) v5.x (v5.3-1.0.0.1 in this paper).
- **Build Toolchain**: GCC = 9.4.0 or newer, CMake = 3.16.3 or newer
- **Other Software**: libnuma, clustershell, Python 3 (with matplotlib, numpy, python-tk)

A.3.4 Data sets Synthetic data sets (YCSB, SmallBank and TATP) are provided along with the artifact.

A.4 Installation

All machines in the cluster need to perform the following actions separately:

1. Clone the git repository using the following command:

```
git clone https://github.com/madsys-dev/smart
```

   Note that the Smart source code should be located in the same directory on all machines.
2. Install the dependencies (libnuma, clustershell, matplotlib, numpy, python-tk) by running `deps.sh`. It requires the root privilege.
3. Build binaries (static library, micro-benchmark program and applications) by running `build.sh`.

A.4.1 Basic Test We provide a micro benchmark program `test/test_rdma` in Smart. It can be used to evaluate the throughput of RDMA READ/WRITE between two servers. Note that the optimizations of Smart are enabled by default.

1. Set the server hostname, access granularity and type in `config/test_rdma.json`. Set RDMA device and enabled optimizations in `config/smart_config.json`.
2. In the server side, run the following command:

```
cd /path/to/smart && cd build
LD_PRELOAD=libmlx5.so ./test/test_rdma
```

3. In the client side, run the following command:

```
cd /path/to/smart && cd build
LD_PRELOAD=libmlx5.so ./test/test_rdma ↙
    [nr_thread] ↙
    [outstanding_work_request_per_thread]
# Example:
# LD_PRELOAD=libmlx5.so ./test/test_rdma 96 8
```

4. When the execution completes, the client displays the following information to stdout (sample output).

```
rdma-read: #threads=96, #depth=8, ↙
    #block_size=8, BW=848.217 MB/s, ↙
    IOPS=111.177 M/s, conn establish ↙
    time=1245.924 ms
```

It also adds a line to the file specified by `dump_file_path` (see `config/test_rdma.json`):

```
rdma-read, 96, 8, 8, 848.217, 111.177, 1245.924
```

A.5 Experiment workflow

A.5.1 Application Functional Test

For other applications (i.e., Smart-HT, Smart-BT and Smart-DTX), we assume two memory blades and one compute blades to be used. Here is an example of Smart-HT to illustrate the steps.

1. Set parameters in configuration files, including `smart_config.json` (for all processes), `backend.json` (for memory blades), `datastructure.json` (for hashtable and B+Tree) and `dtx.json` (for distributed transactions).
2. For each memory blade, run the following program:

```
cd /path/to/smart && cd build
LD_PRELOAD=libmlx5.so ↙
    ./hashtable/hashtable_backend
```

3. In compute blades, run the benchmark program using the following command. It must match the backend started in the previous step.

```
cd /path/to/smart && cd build
LD_PRELOAD=libmlx5.so ↙
    ./hashtable/hashtable_bench [nr_thread] ↙
    [nr_coro]
```

4. After execution, the benchmark program displays the throughput and latency to stdout. It also adds a line to the file specified by `dump_file_path`.

A.5.2 Reproduce Tests

Configuration. It requires at least eight machines with Smart installed to to reproduce experimental results. **Passwordless login must be enabled in this cluster.**

In the scripts we provide, the test cluster consists of machines whose hostnames are `optane00` through `optane08`. You can replace both the hostnames and the root directory that appear in the following configuration files as appropriate (**see `ae/README.md` for details**):

- `ae/collect/common.sh`
- `config/*.json`
- `ae/collect/config/*.json`
- `ae/plot/common.py`

Execution. On the client machine specified by `ae/collect/common.sh`, run `run.sh` in artifact's root directory, which will execute all the experiments and generate the corresponding figures.

If only part of the test needs to be performed, you can comment out unnecessary lines in both `ae/collect/runall.sh` and `ae/plot/runall.sh`.

A.6 Evaluation and expected results

The test scripts produce numerical results in `ae/raw/*.csv`, and figures in `ae/figure/*.pdf`. We provide reference results in `ae/raw-reference/*.csv` and `ae/figure-reference/*.pdf` respectively.

A.7 Experiment customization

The parallelism for the real-world applications performance evaluation experiment can be modified by setting the `thread_set` and `depth_set` variables to appropriate number of threads and coroutines in the scripts. You should modify `ae/collect/common.sh` and `ae/plot/common.py` together and keep them consistent.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[2] Dotan Barak. ibv_open_device() - rdmamojo. https://www.rdmamojo.com/2012/06/29/ibv_open_device/, 2022.

[3] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[4] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.

[5] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[6] CXL Consortium. Compute express link. https://www.computeexpresslink.org/, 2022.

[7] Gen-Z Consortium. Gen-z. https://genzconsortium.org/, 2022.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb.

In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[9] Hewlett Packard Corporation. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2022.

[10] Intel Corporation. Intel rack scale design (intel rsd). https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, 2022.

[11] Intel Corporation. Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, 2022.

[12] NVIDIA Corporation. Mellanox adapters programmer's reference manual (prm). https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf, 2022.

[13] NVIDIA Corporation. Neo-host. https://support.mellanox.com/s/productdetails/a2v50000000N2OlAAK/mellanox-neohost, 2022.

[14] NVIDIA Corporation. Nvidia connectx infiniband adapters. https://www.nvidia.com/en-us/networking/infiniband-adapters/, 2022.

[15] Diego Crupnicoff, Michael Kagan, Ariel Shahar, Noam Bloch, and Hillel Chapman. Dynamically-connected transport service, July 3 2012. US Patent 8,213,315.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[17] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264, 2016.

[18] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with directcxl. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[19] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 243–252, 1994.

[20] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[21] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2022.

[22] Intel. Intel® ethernet controller e810. https://www.intel.com/content/www/us/en/products/details/ethernet/800-controllers/e810-controllers/docs.html, 2022.

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

[24] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[26] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[27] Anuj Kalia, Michael Kaminsky, and David G Andersen. 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)*, pages 185–201, 2016.

[28] Linux Kernel. Rdma core userspace libraries and daemons. https://github.com/linux-rdma/rdma-core/blob/master/providers/mlx5/qp.c#L787, 2022.

[29] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal De Lara. Jitserver: Disaggregated caching jit compiler for the jvm in the cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 869–884, 2022.

[30] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, and Alvin R Lebeck Danyang Zhuo. Understanding rdma microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[31] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, Renton, WA, April 2022. USENIX Association.

[32] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. Performance analysis of exponential backoff. *IEEE/ACM transactions on networking*, 13(2):343–355, 2005.

[33] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. Dinomo: An elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023 – 4037, 2022.

[34] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[35] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. Smart: A high-performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 553–571, Boston, MA, July 2023. USENIX Association.

[36] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[37] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773, 2020.

[38] Mellanox. ibv_open_device() - rdmamojo. https://dlsvr04.asus.com.cn/pub/ASUS/mb/accessory/PEM-FDR/Manual/Mellanox_OFED_Linux_User_Manual_v2_3-1_0_1.pdf, 2022.

[39] Robert M Metcalfe and David R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[40] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[41] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling rdma rpcs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 212–227, 2021.

[42] The Boost Organization. Boost c++ libraries. https://www.boost.org/, 2022.

[43] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE Computer Society, 2014.

[44] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. Aifm: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[45] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, pages 433–448, 2019.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[47] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. Fusee: A fully memory-disaggregated key-value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 81–98, 2023.

[48] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[49] TATP. Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net/, 2022.

[50] The H-Store Team. Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2022.

[51] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

[52] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48, 2020.

[53] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[54] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, 2022.

[56] Qing Wang. Sherman: A write-optimized distributed b+tree index on disaggregated memory. https://github.com/thustorage/Sherman/, 2022.

[57] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1033–1048, 2022.

[58] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. The case for distributed shared-memory databases with rdma-enabled memory disaggregation. *Proc. VLDB Endow.*, 16(1):15–22, nov 2022.

[59] Zixuan Wang, Joonseop Sim, Euicheol Lim, and Jishen Zhao. Enabling efficient large-scale deep learning training with cache coherent disaggregated memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 126–140. IEEE, 2022.

[60] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, 2018.

[61] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[62] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment*, 10(6), 2017.

[63] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. *Proceedings of the VLDB Endowment*, 12(11):1637–1650, 2019.

[64] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. Ford: Fast one-sided rdma-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, 2022.

[65] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment*, 14(10):1900–1912, 2021.

[66] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, et al. Carbink: Fault-tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, 2022.

[67] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 741–758, 2019.

[68] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29, 2021.

[69] Pengfei Zuo, Qihui Zhou, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua, James Cheng, Rongfeng He, and Huabing Yan. Race: One-sided rdma-conscious extendible hashing. *ACM Transactions on Storage (TOS)*, 18(2):1–29, 2022.

Received 2023-04-20; accepted 2023-07-29