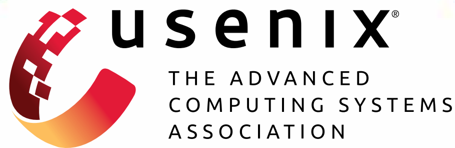

Patronus: High-Performance and Protective Remote Memory

Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/fast23/presentation/yan

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

Patronus: High-Performance and Protective Remote Memory

Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University

Abstract

RDMA-enabled remote memory (RM) systems are gaining popularity with improved memory utilization and elasticity. However, since it is commonly believed that fine-grained RDMA permission management is impractical, existing RM systems forgo memory protection, an indispensable property in a real-world deployment. In this paper, we propose PATRONUS, an RM system that can simultaneously offer protection and high performance. PATRONUS introduces a fast permission management mechanism by exploiting advanced RDMA hardware features with a set of elaborate software techniques. Moreover, to retain the high performance under exception scenarios (e.g., client failures, illegal access), PATRONUS attaches microsecond-scaled leases to permission and reserves spare RDMA resources for fast recovery. We evaluate PATRONUS over two one-sided data structures and two function-as-a-service (FaaS) applications. The experiment shows that the protection only brings 2.4 % to 27.7 % overhead among all the workloads and our system performs at most ×5.2 than the best competitor.

1 Introduction

Remote memory (RM) architecture, which decouples CPU and memory into two independent resource pools (i.e., compute nodes and memory nodes), is changing the landscape of modern data centers by providing many benefits, such as high memory utilization and efficient memory sharing [2, 12, 44]. This trend is sparked by the widely-deployed RDMA network, which allows compute nodes to access remote memory (at memory nodes) in a one-sided and low-latency manner. There are myriad efforts to make RM systems practical on multiple fronts, such as proposing easy-to-use programmable models [1, 43, 46], designing efficient remote indexes [50, 59], and deploying popular applications [38].

However, there is still an obstacle to cross on the way to practical RM systems: *remote memory protection*. Existing RM systems expose all RM resources or coarse-grained memory regions to compute nodes without carefully considering protection [2, 12, 15, 25, 31, 32, 36, 39, 41]. This inevitably induces several anomalies. First, buggy or malicious code in clients[1] can generate illegal one-sided access to the RM, introducing data corruption or privacy breaches. Second, even if the clients are well-behaved, concurrent memory reallocations can turn the in-flight one-sided access illegal (§3.1).

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

[1]Clients are processes in compute nodes accessing RM.

It is non-trivial to simultaneously achieve protection and high performance in RM systems. First, considering the high throughput of RDMA networks (e.g., ~70Mops/s in 100Gbps ConnectX-5 RDMA NIC), clients will frequently acquire/revoke permission upon memory allocation/deallocation. But the common RDMA protection mechanism, i.e., (re)-registering memory region (MR) to targeted memory areas, suffers high latency due to the overhead from OS kernel and RNIC (~1 ms for 256 MB; see Figure 1). Even worse, RM systems typically only have weak computing power at memory nodes [48, 55, 59], which limits the rate of acquiring/revoking permission, thus bottlenecking the system performance. Second, on the exception path of RM systems, i.e., clients fail or access illegal RM addresses, retaining high performance with a protection guarantee is challenging. Specifically, when a client fails, it may hold exclusive access permission to some memory areas. If the failed client's permission cannot be revoked rapidly, the progress of the whole RM system will be negatively impacted. When a client accesses illegal RM addresses, RDMA NICs (RNICs) at memory nodes will turn the associated queue pair (QP) into an error state, disabling subsequent RM access. Recovering the faulted QP needs a millisecond-scaled process and thus produces latency spikes for RM applications.

In this paper, we propose PATRONUS, a *protective* RM system that can provide high performance. In the control path, memory nodes perform memory (de)-allocation and byte-wise memory protection for clients using weak computing power (i.e., ≤ 4 CPU cores). In the data path, clients at compute nodes access RM with permission via one-sided RDMA verbs. PATRONUS attains efficiency on both normal and exception paths. This is achieved by combining advanced RDMA hardware features and careful software design.

To enable fast permission management with weak computing power on memory side, PATRONUS first exploits *memory window (MW)* [42], an advanced RDMA hardware feature allowing RNICs to regulate the access (thus supporting one-sided RDMA) while minimizing the overhead of interaction with RNICs. Different from MR, an MW operation communicates with RNIC asynchronously and in userspace. With permission bits modified by hardware, it enjoys low latency (1.1 µs; see Figure 1). However, simply using MW cannot meet the performance requirements at peak load. Thus, we introduce a set of software techniques (e.g., MW handover and delayed unbinding; §5.4) to reduce the number of MW operations, saving the computing cycles of memory nodes.

To react fast to client failures, we equip MWs with *microsecond-scaled* leases, so that the permission will be automatically reclaimed by memory nodes on timeout. However, the fine-grained leases introduce the overhead of frequent extension to memory nodes. We reduce the extension overhead by delegating the management of lease metadata to the client with one-sided verbs while retaining the protection guarantee.

To mitigate the negative effect of illegal access (i.e., QP faults), instead of recovering the faulted QP in the foreground, we switch to another intact QP as a substitution. To avoid QP creation in the critical path, PATRONUS prepares a small number of spare QPs.

We evaluate PATRONUS thoroughly over microbenchmarks and two sets of realistic applications, i.e., the remote one-sided data structures (ODS) and the function-as-a-service (FaaS) platform. Among all the workloads evaluated, the protection only brings 2.4 % to 27.7 % overhead, and PATRONUS performs up to ×5.2 better than all the competitors. On the exception path, we reduce the interruption from faulted QP by 92 %. The lease semantics ensures the progress of the system under client crashes, evaluated under the case of ODSs.

Contributions. The main contributions are:

- An analysis of the deficiency of existing protection mechanisms and the performance goals for a protective RM system (§3).
- The design and implementation of PATRONUS, a protective RM system that retains high performance on both normal and exception path (§5).
- The thorough evaluations over microbenchmarks and realistic workloads to demonstrate the high performance of PATRONUS (§7).

2 Background

2.1 RDMA and Access Protection

RDMA is a high bandwidth (e.g., 200 Gbps) and low latency (~2 µs) networking technology widely adopted in today's data centers [12, 13, 20]. RDMA provides two types of verbs to the application, namely *one-sided verbs* and *two-sided verbs*. The one-sided verbs offer a remote memory abstraction; it allows *direct access* to the remote memory while bypassing remote CPUs. The two-sided verbs offer a *message-passing* interface similar to the well-known Linux socket. The two types of verbs make different trade-offs: the one-sided verbs are efficient for saving computation resources, but they risk data corruption for the lack of remote CPU regulation; the two-sided verbs are vice versa. The one-sided verbs are more prevalent due to their higher efficiency (i.e., ×1.7 throughput) [52].

Access protection. RDMA provides basic mechanisms for regulating RDMA verbs, e.g., *queue pair (QP)* and *memory region (MR)*. QP is the communication endpoint on which the client posts RDMA requests (via `ibv_post_send`); it offers channel-wise restrictions on the access type (i.e., readable

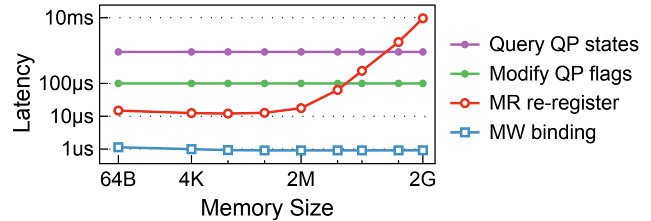

Figure 1: Median latency of protection-related operations in RDMA.

or writable). MR represents a memory area registered to the RNIC for remote access; it restricts both the access type and the accessible range of memory.

The MR/QP operations, in the RDMA control path, have orders of magnitude higher latency than the microsecond-scaled RDMA data path (Figure 1, [53]). Specifically, modifying QP flags includes a transition of QP states in the RNIC, taking ~100 µs per operation. The MR registration is synchronous and requires kernel involvement (e.g., context switches, page pre-faulting, and page pinning); it yields a non-scalable performance [3] with ~1 ms latency for a 256 MB area. Due to the inferior performance, most RM systems only involve QP constructions and MR registrations on bootstrap [12, 34, 53].

2.2 The Remote Memory Architecture

RDMA is the key enabler to the remote memory (RM) architecture for its ultra-low latency in interconnecting. RM is getting prevalent in the decade because it addresses the problem of memory usage imbalance in traditional data centers [6, 14]. With RM, the CPU and memory are assembled into two separate components, i.e., the compute nodes (CN) and the memory nodes (MN). The compute nodes gather a mass of CPU cores (10s - 100s), while the memory nodes typically have weak and limited computing power [1, 59]. The scarce computation power on MNs catalyzes a range of *RM-native applications* that mainly leverage one-sided verbs, such as KV stores [25, 36, 48], transactional systems [12, 52, 55], and data structures [4, 35, 50, 51, 58, 59]. These various workloads coexist in the cluster and share the remote memory.

3 Motivation

3.1 The Call for Stray Protection in RM

The efficiency of RDMA one-sided access comes at a price: its direct nature saves the overhead of remote CPU but in turn escapes the *protection* against *stray access*. The stray access is the illegal one towards an area of memory unowned by the process. Next, we show two causes for the stray access.

Causes of stray access. (*i*) Buggy and malicious codes. A careless bug or a piece of malicious code can generate stray access to the RM by setting an overflowed address in the RDMA request. This is a *space* anomaly that can occur when RM exposes a larger range of memory space than allowed. (*ii*) Race to memory management. The memory deallocation and reallocation make all the unaware one-sided access to-

wards the address stray. This is a *time* anomaly that can occur when RM exposes a longer duration of permission than the application logically allowed.

In response to the causes, a protective system needs to expose only the range of memory that is allowed to access, and invalidate the permission timely after access is finished.

Cases for protection and requirements. We observe two trends making in-RM protection more urgent, posing requirements for a protective system. (*i*) The RM architecture catalyzes a wide range of remote *one-sided data structures (ODS)* [4,35,50,51,58,59], which involve frequent memory (de)-allocations and floods of concurrent access, bringing a high risk of memory management race at runtime. Moreover, the access to the ODS is typically shared and fine-grained (e.g., at a granularity of buckets in the hash table), which requires fine-grained protection for proper access isolation. (*ii*) In the function-as-a-service (FaaS) platform, functions submitted from different users leverage the shared RM for performant (intermediate) data storage [26,53], which asks for access isolation to avoid data tampering or leaks. Functions have a short lifetime (~µs), scale out quickly (to ~millions), and access RM on demand [9, 14, 27, 33], which requires a low-latency and high-throughput protection management to meet performance needs in the critical path.

To conclude, a protective system needs to offer high-performance protection management in fine granularity.

3.2 Goals for the Protective RM System

Considering that the RM system is an infrastructure to determine the overall performance of workloads, it should remain efficient in a variety of situations. Besides offering fast permission management on the normal path, it should be able to retain performance even under client failures or illegal access. Next, we elaborate on the performance goals.

Goal#1: manage protection fast. Existing workloads can introduce a mass of permission requests to the system in the peak case, demanding high throughput in permission management. For example, the bulk load to a remote hash table [59] brings a flood of concurrent permission acquisitions. Querying to the hash table involves multiple access to disjoint memory areas (e.g., the bucket and the KV block), introducing multiple permission acquisitions from one query. Therefore, we expect a protective system to offer *high-throughput* protection management to avoid introducing bottlenecks.

Goal#2: react fast to client failure. A client can affect the progress of the whole system if it crashes with exclusive permission held. This is common because clients are deemed error-prone in the distributed system, and access to the metadata should be exclusive in many workloads. Therefore, we expect that a protective system can react fast to client failures.

Goal#3: retain performance under illegal access. The illegal access turns the QP into an error state, in which the QP rejects any incoming RDMA requests, causing a serious interruption in application running. The interruption will further affect other innocent clients on the same QP (sharing QPs is very common under QP virtualization and is widely adopted for mitigating the scalability problem [12,49,53]). Therefore, we expect a protective system to retain performance in the appearance of illegal access.

Name	Goal#1	Goal#2	Goal#3	For RM
Two-sided	-	✓	✓	✗
MR	✗	✗	✗	✓
QP	✗	✗	✗	✓
MW	✓	✗	✗	✓
MW + ***SW*** (PATRONUS)	✓✓ (§5.4)	✓ (§5.3)	✓ (§5.5)	✓

Table 1: Deficiency of existing solutions. Goals are elaborated in §3.2. Only MW with *software (SW)* co-design meets all the goals (PATRONUS).

3.3 Deficiency of Existing Solutions

Existing solutions are all deficient for a protective RM system (Table 1). The two-sided verbs do not work well in the RM architecture where computation power is scarce on the memory node. Other existing solutions that regulate one-sided access (i.e., MR and QP, §2.1) can not simultaneously meet the three goals. In this section, we revisit these mechanisms and examine their applicability.

For protection management (G#1), the QP-based solution is coarse-grained and the MR-based solution is slow. The QP-based solution is channel-wise; it is unable to offer byte-wise protection as workloads require. Therefore, its use is very limited [3]. The MR-based solution has a high latency (~20 µs per 2 MB[2], Figure 1) and does not scale; due to the performance issue of MR, existing systems do not utilize its protection at runtime. For example, Octopus [34] registers MRs on bootstrap and never manipulates them later; FaRM [12] uses a large memory region of 2 GB, which essentially leaves the whole region unprotected.

For detecting client failures (G#2), MR is not aware of any failures on the remote side. Although QP does reveal remote failures (by issuing RDMA operations as heartbeats), it does not detect failures that leave QPs intact (e.g., clients using virtualized QPs as communication channels and clients getting hanged).

Finally, in the appearance of illegal access (G#3), whether violating the permission restricted by QP or MR, the QP will run into an error state, requiring an expensive bootstrap procedure to recover the QP (~1 ms, §7.2.5).

We conclude that *pure* hardware solutions are insufficient to achieve all the goals.

[2]We use 2 MB huge pages, a widely-adopted solution to reduce RNIC's page translation cache misses [12,52].

4 Approach Overview

4.1 Opportunity: Memory Window (MW)

The *memory window (MW)* [42] is an advanced RDMA feature widely supported in commodity RNICs. It acts as a supplementary layer over MR to deliver flexible protection management at runtime.

Interface. The MW needs to be allocated before use. It supports two types of operations, i.e., *bind* and *unbind*[3]. Binding an MW over a memory area exposes the access permission while unbinding the MW invalidates it. Note that we can bind an MW multiple times after it is allocated; each time the previous permission will be invalidated and the new permission will be granted (also called *rebind* in this paper).

Binding (rebinding) an MW takes the address and size of the memory area and the access type (read or write) as parameters. The MW exposes the memory area by generating an *rkey* (a 32 bit integer) as the permission token to the client, like in the case of MR. MW is *byte-granularity* in that it works for unaligned memory areas of any size. Binding/unbinding the MW uses the same `ibv_post_send` interface as RDMA verbs, which communicates with the RNIC in userspace asynchronously and allows requests batching.

Latency: MW vs. MR. MW binding contrasts with MR registration in two ways. First, MW binding has a *constant* latency with memory areas of any size, unlike MR registration taking proportional overhead to the size. Second, MW binding has much lower latency, i.e., 1.1 µs in median (Figure 1).

The reason for the performance difference is that MW binding communicates to the RNIC asynchronously in userspace, while MR registration is synchronous and requires kernel involvement[4], introducing the additional overhead of context switches, page pre-faulting, and page pinning.

4.2 Solutions

Although MW accelerates permission modifications, it does not introduce new features beyond MR. Therefore, MW also has limited functionality as MR. We observe that to achieve the goals, direct adoption of MW is not enough, and *software co-design* must be involved. Next, we show how software techniques are developed to fill the gap.

Can software further contribute to the overall performance? (G#1) Although MW already acts as a low-latency mechanism to manage permission, the overhead of MWs can still burden the memory nodes where CPUs are limited. We observe that software co-design can exploit the true potential of hardware for efficient permission management.

Solution: save MW operations without sacrificing protection semantics. Instead of improving performance by lowering the protection guarantee, we reduce overhead in a way the protection assurance is not sacrificed. This is possible by leveraging the characteristics we find in management. For example, by noticing that permission requests come in a batch, we can leverage the pairs of *opposite* operations to reduce the number of MW operations effectively by half (called *MW handover*). By noticing that some memory areas will not be re-used immediately after being freed, we delay the unbinding of the MWs to save operations. Finally, by exploiting the potential of address contiguity, we can combine multiple MWs into one. They are elaborated in §5.4.

How to react fast to client failure? (G#2) Like MR, MW itself is not aware of any failures from the CN side. Extra techniques must be developed to detect and handle the failure.

Solution: borrow the idea of leases. MW only offers space-wise protection. We introduce the *lease semantics* (i.e., expire on timeout, [41]) to MW from the software to enable time-wise protection. In doing so, the system can resume progress by expiring any exclusive permission on timeout, no matter whether the permission is held by a crashed or a slow client.

The lease management metadata seems too crucial to be exposed. Nevertheless, we notice the *byte-granularity* property of MW, which allows us to expose only the necessary part of metadata to the client. With this help, we are able to offload part of the lease management overhead to the client without risking metadata tampering (*CN-collaborated extension*, §5.3). It saves CPU cycles for the memory nodes.

How to retain performance under illegal access? (G#3) Like MR, MW protects against data corruption but *does not* protect the QP from running into an error state, which seriously interrupts application running.

Solution: conceal the interruption rather than prevent it. We notice that illegal access is unable to prevent because the memory node invalidates the permission (i.e., unbinds the MW) without notifying the client. Therefore, we try to conceal the interruption caused by the faulted QP instead of preventing it. We prepare *spare QPs* to stand in for the faulted ones at runtime so that the recovery overhead can be concealed in the background. In doing so, we leveraged a special property of MWs: they can remain valid across all QPs[5]. Therefore, the granted permission remains valid even if the underlying QP has changed. They are elaborated in §5.5.

5 PATRONUS: The Protective RM system

Motivated by how stray access is common and necessary to be prevented (§3.1), we design a protective RM system in response, called PATRONUS, to offer complete protection with sufficient performance for existing workloads (§3.2).

Different from previous systems where the whole remote memory is exposed to the clients [12, 34], the basic idea of PATRONUS is leaving clients with *no initial permission* and demanding permission acquisition before allowing clients to issue remote access.

[3]To distinguish, we use the verb *bind* for MW and *register* for MR.

[4]Although MR supports on-demand paging (ODP), which can remove page pinning, it is notorious for causing high latency (>=10 ms) on normal RDMA access upon remote page faults [22].

[5]Precisely, MWs work across all QPs under the same *protection domain (PD)*.

Category	API	Parameters	Return	Description
Control Path	`allocate`	*size, time, ex/shr*	*Perm*	Allocate memory and acquire permission
	`acquire`	*addr, size, time, r/w, ex/shr*	*Perm*	Acquire permission over ⟨*addr*, *size*⟩
	`extend`	*Perm, time*	*Success*	Extend the permission lease
	`revoke`	*Perm*	*Success*	Revoke the permission
Data Path	`read/write/CAS/FAA`	*Perm, addr, size, buffer*	*Success*	Issue remote access (support batching)

Table 2: The PATRONUS APIs. In the parameters, *r/w* specifies read/write permission; *ex/shr* specifies exclusive/shared access mode; *time* specifies the expected lifetime of the permission lease. *Perm* is an opaque object containing the remote address, the `rkey` as the permission token, and the expiration time. *Success* denotes whether the call succeeded.

Failure model and leases. PATRONUS considers two kinds of failures for the client, i.e., fail stop and fail slow, both addressed by leases. First, the lease ensures availability on client crashes (fail stop). The *orphaned* exclusive permission held by a crashed client precludes other clients from accessing the memory, resulting in unavailability. The lease resumes system progress by expiring the permission on timeout. Second, lease accelerates memory reclamation with slow clients (fail slow). A slow client (e.g., due to network traffic) hinders *actual* memory reclamation, because the active permission it holds makes the memory area potentially accessible and thus not reclaimable. The lease forcibly invalidates permission on timeout to allow reclamation on time. We assume loosely-synchronized clocks for the lease to work, like similar work [19]. PATRONUS does not handle the failure of memory nodes, where orthogonal work (e.g., erase coding) can be applied [30, 57].

5.1 The Interface

PATRONUS provides control path APIs to acquire new permission, extend the permission lease, and revoke permission. The data path APIs accept the permission as a parameter and are translated into one-sided verbs for remote access (Table 2).

Permission starts in two cases. (*i*) Client allocates remote memory via the `allocate` call. (*ii*) Client attempts to access a known remote area for the first time, in which case the client needs to get the permission via the `acquire` call. Both calls will issue an RPC to the memory node, where the memory node starts new permission by binding MWs to the allocated/specified memory, and responds with a `Perm` object to the client. `Perm`, needed by all the data path API, contains the permission token (`rkeys` of the MWs), the expiration time, and the remote address. Note that re-access to the same memory can re-use the previous `Perm` as long as it has not expired.

In the parameters, the client specifies the access mode (read-/write), ownership (shared/exclusive), and expected lifetime for the permission lease. For acquisitions that conflict in the ownership, PATRONUS postpones granting the latter permission until the conflict resolves.

PATRONUS allows pre-allocation to amortize the overhead; i.e., clients call `allocate` at a larger granularity and back their fine-grained allocators on the blocks. Nevertheless, it does not speed up the queries or in-place updates from other clients (which is also common), because those clients still need to call `acquire` for their own fine-grained permission.

Permission extends via the `extend` call. Extending an existing permission is more efficient than re-acquiring a new one. We assume that clients only extend the *hot* permission that is re-used frequently.

Permission ends when the client explicitly revokes the permission (via `revoke`) or when the lease expires. The `revoke` will issue an RPC. The to-expire leases are detected by periodical scans from the memory node. In both cases, the memory node unbinds the MWs to invalidate the permission.

Data path. PATRONUS purely uses one-sided verbs in the data path (`read`, `write`, `CAS`, and `FAA` calls). It supports batch execution and therefore allows the familiar IO consolidation optimizations in the application [59].

5.2 Architecture Overview

PATRONUS provides a library for the client in compute nodes (*CN-lib*) and a *manager* daemon for memory nodes, as illustrated in Figure 2. The manager manages the remote memory and permission in response to clients' RPC.

Main components. PATRONUS manager takes over the whole memory on MN. Most of the memory will be exposed for the client's use; we call them *buffers* in the memory pool. The other ($\leq 0.02\,\%$) is reserved on bootstrap to use as the *header pool*.

The header is a central structure that stores all necessary metadata for a permission. Each individual permission (possibly over the same memory area but owned by different clients) has an individual header. The header contains two kinds of information (Figure 3). (*i*) The resource information, i.e., the address and size of the RM buffer and the corresponding MWs. (*ii*) The lease information, such as when the permission starts and how long the permission will last. We use the address of the header as the cluster-wise permission *identifier* in the RPC between clients and the manager.

Permission management. The start of permission is triggered by the client's `allocate` or `acquire` calls. In response, the manager allocates a header for the new permission and then binds *two* MWs to expose both the buffer and the header to the client (❶ in Figure 2). The header is additionally exposed

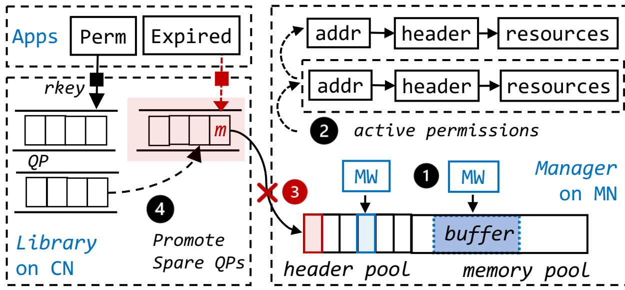

Figure 2: The architecture of PATRONUS. We assume that CNs own a mass of CPU cores (10s - 100s), while MNs use several weak cores to operate the manager.

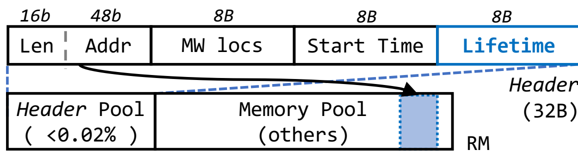

Figure 3: The format of the 32 B *header*. `MW locs` denotes the locations of the MWs. Blue indicates the area exposed to the client, i.e., the *Lifetime* variable in the header and the buffer in the RM.

so that the client can facilitate permission management, a technique called *CN-collaborated extension* (§5.3).

The remote memory is managed by slab allocators with different object sizes, similar to FaRM [12]. Permission has the same granularity as memory management: clients must start permission over the whole object, but not a part of it. The manager uses a per-slab hash table to store all the active permission, with addresses as the keys; in this way, permission can be queried efficiently. To detect any to-expire permission, the manager periodically polls the hash table to collect the timeout ones (❷ in Figure 2). The polling overhead is minor because the number of active permissions in the system is typically small.

To invalidate permission, the memory node unbinds the MWs, which in turn invalidates the permission token (`rkey`), causing RNIC to forcibly reject the RDMA requests with that `rkey` (❸ in Figure 2).

5.3 CN-collaborated Extension

Considering that unexpected permission expiration seriously interrupts application running, PATRONUS allows extending the permission on runtime. The naive approach is using an RPC to notify the manager of the extension. However, the communication brings significant overhead to the memory node where computation power is scarce. To mitigate this overhead, we propose to utilize the collaboration from the CNs for extension while handling careless and malicious clients correctly.

Collaboration from CN. The metadata in the header seems too crucial to be exposed. Nevertheless, we notice that MW is *byte-granularity*; therefore, it can be used to expose only the necessary part of metadata to the clients without risking metadata tampering.

The basic idea is to expose the *lifetime* variable in the header (Figure 3) so that the client can update it in a *one-sided* way. In turn, the manager will encounter extended permissions on polling for the timeout ones; the manager skips them.

Regulate the extension. The CN collaboration can introduce the starvation problem in the system without proper regulation. Specifically, (*i*) the client is able to set *lifetime* to a large value to own it infinitely. (*ii*) The client is also able to extend continuously, starving other clients.

In response, we propose two regulations for the extension. First, we require that any permission can not live beyond a predefined maximal lifetime (empirically set to several milliseconds). Any aged permission will be detected by the manager and be forcibly invalidated.

Second, to avoid starving other clients, we need an efficient way to notify the owner that a permission is no longer extendable. In PATRONUS, the notification is implemented by setting the lifetime to *zero* and requiring the CN-lib to update the lifetime with `RDMA_CAS` instead of `RDMA_WRITE`. In this way, the zeroed lifetime causes `RDMA_CAS` to fail and thus the clients are notified. Note that the permission is arbitrated on the memory node, which means that the manager can always reclaim the permission by forcibly invalidating the MWs if it suspects any anomalies, without negotiating with the client.

Trade-off analysis. With collaboration, the overhead of an RPC (the naive approach) is reduced to one inbound one-sided access. The collaboration benefits the performance because (*i*) one-sided verbs are more efficient than two-sided ones, and (*ii*) inbound verbs are more efficient than outbound ones [25]. The benefit will enlarge if extensions occur multiple times.

Exposing the lifetime variable introduces the overhead of using one more MW. Nevertheless, we deem the overhead minor compared to the naive approach (taking one RPC), because the additional MW operations can be batched together in the `ibv_post_send` API, communicating with the RNIC once. Furthermore, as we show in §5.4, this extra MW overhead can be reduced most time.

Polling: the alternative to lease. An alternative approach to the lease semantics is *QP polling*, where MNs track whether CNs are still alive by periodically issuing RDMA operations to each QP as heartbeats. Polling has several deficiencies compared to leases. First, it does not handle fail slow of clients. Second, polling can not distinguish clients sharing the same QP (while QP sharing is common [12, 49, 53]). In terms of overhead, polling and leases both pay one RDMA operation for keepalive; however, leases allow to save one `revoke` call by letting the permission expires itself, potentially yielding better performance.

5.4 Reduction of Permission Overhead

In this section, we introduce our techniques for reducing the permission overhead *without* sacrificing protection assurance.

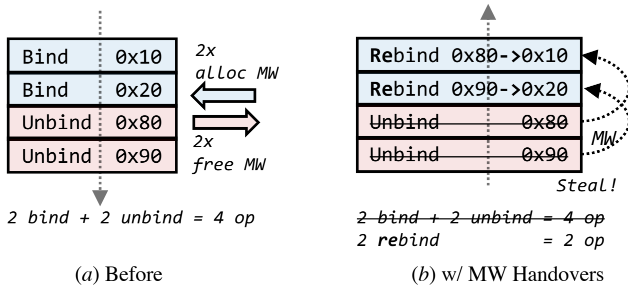

Figure 4: The *MW handover* technique saves half of MW operations by stealing (reusing) MWs from the unbinding requests to the binding requests. `0xdd` denotes the *addr*.

We elaborate on the characteristics we find in protection management and how we leverage them to reduce MW operations.

#1: Leverage pairs of opposite operations. Every active permission ends eventually. Therefore, among all the MW operations in the system, about half of them are binding while the others are unbinding. Following this fact, we can combine every pair of opposite MW operations into one *rebinding* operation, a technique we call *MW handover* (Figure 4). Rebinding an MW, which takes the new $\langle addr,\ size \rangle$ as parameters, generates new `rkey`s and invalidates the old `rkey`s (§4.1). To adopt this technique, the manager collects opposite operations with the best effort: the to-expire permission will be polled and clients' requests will be scanned before performing the handover.

Trade-off analysis. The benefit of this technique comes with no price because handover is only performed in a speculative way. First, the manager never waits for future requests; no latency is introduced. Second, client requests on the memory nodes are naturally batched by the RNIC, which is a hardware approach and does not introduce extra batching overhead. The memory node, accordingly, always handles requests in a batch, leaving room for performing handover.

#2: Delay unbinding if memory is not re-used. We observe that if a memory area is not re-used after deallocation, we can delay unbinding the MW because stray access to this area does not introduce data corruption. The header is a good candidate for doing so: we reserved more-than-enough headers in the system, so it is easy enough not to re-use the just-deallocated header in the near future. If the available memory buffers are adequate in the system, we also delay unbinding the buffer MW; however, if it is not the case, we unbind the buffer MW and reclaim (thus re-use) the buffer promptly.

Trade-off analysis. This technique wastes available headers and MWs but in a minor way. The waste of headers is negligible because we reserve adequate headers in the pool for the extreme case. We carefully encode the header so that the pool occupies no more than 0.02 % of a regular memory node (§5.6 for details). The waste of MWs is trivial because the RNIC (Mellanox CX-5 in our case) allows ~16 million MWs, far from possibly being used up.

#3: Exploit the potential of contiguity. We observe that if two addresses are contiguous and share the same protection lifetime, we can combine the two MWs into one. At first glance, the situations of this case are rare because addresses are generally *not* contiguous and memory buffers seldom share the same protection lifetime. We exploit this potential in handling the `allocate` RPC: we can allocate an extra 32 B to place the header right before the buffer; thus, the header and the buffer are contiguous (this case is not revealed in the figures for brevity). While doing so, we carefully place the to-expose variable (i.e., the *lifetime* variable in Figure 3) at the tail of the header to make the to-expose areas contiguous.

Trade-off analysis. The benefit comes with no price. At first glance, allocating an extra header introduces a lot of 32 B holes. However, these holes are not wasted, because they can be re-used as headers again when the permission over the same buffer is re-acquired. This situation is very common; for example, inserts to the remote hash table involve allocations of KV blocks. These KV blocks will be re-accessed when being read or modified. In this case, the following permission acquisitions can re-use the holes as headers again. On deallocation, the frontal 32 B will be reclaimed altogether; thus they are not leaked.

5.5 Isolation from Illegal Access

Although MW can prevent illegal access from corrupting the memory, it does not handle the consequence of it: the illegal access will turn the underlying QP into an error state. The faulted QP requires an expensive procedure for recovery (~1 ms), seriously interrupting application running.

We observe that this interruption is not preventable by the software. This is because process scheduling and network traffic can introduce a nondeterministic delay to the one-sided request. During the delay, the permission may have expired. Therefore, we propose to *conceal* the interruption rather than prevent it.

Conceal the interruption. We prepare spare QPs to conceal the interruption caused by QP failure. Specifically, each client is assigned a virtual QP number, which maps to a physical QP initially. On QP failure, we transparently promote one of the spare QP by altering the virtual-to-physical mapping (❹ in Figure 2). In this way, the client can resume its execution immediately. A special property from MWs enables continuous execution, i.e., the MWs are able to remain valid across all QPs. This wide validity allows the previous permission to remain valid in the new QP. Therefore, the client does not need to re-acquire permission when QP switches.

A small number of spare QPs are sufficient to hide the foreground interruption as the manager performs QP recovery in the background. We assume a low illegal access rate (less than 1-10s per second) compared with the speed of QP recovery (~1 Kops).

Trade-off analysis. The spare QPs introduce no overhead for the normal path. The reason is that the spare QPs, while

inactive, will not contend for the rare RNIC resources (e.g., the limited cache [37]).

The spare QPs consume host memory but in a negligible way. To adopt this technique, considering that we use the peer-to-peer RC (reliable connection) type of QP, each memory node needs to prepare $O(C)$ spare QPs for connection, where C is the number of compute nodes. It does not cost much, because even for a large cluster with one thousand CNs, preparing 3 QPs for each CN only consumes ~1 MB host memory (each QP takes ~375 B; [40]).

5.6 Implementation Details

MW pool. The allocation of MWs, unlike binding, has a much higher latency (1 µs vs. 100 µs). We maintain an MW pool to offload the allocation off the critical path.

Header encoding and overhead. The header only takes up 32 B after our effort on data encoding. We leverage the *tagged pointer* [59] to steal the higher 16 bit for the buffer size (*Len*), which is able to present 0-64 KB. For the case where larger buffers are common, we use a scale factor for *Len*, e.g., 64 or 4096. Since we need to locate two MWs (i.e., header and buffer) for each permission, we store *two* 4 B MW indexes to locate MWs in the MW array. *Start time* and *lifetime* are encoded in microseconds; 8 B is ample to encode any time in theory. In the extreme case where 1 million permissions are simultaneously present in the system (clients own very few active permissions in general), the headers only occupy 32 MB in total (< 0.02 % with 128 GB memory). In conclusion, the memory consumption is negligible.

Handling double invalidation. Without careful management, double invalidation of permissions may occur in the system, caused by the famous *ABA problem*. Specifically, the ABA problem comes when an obsolete RPC tries to locate the permission header that has already been re-used. To address this problem, the *start time* is additionally attached with the permission identifier in each RPC. The manager filters out any RPCs whose start time does not match.

6 The Cases for PATRONUS

In this section, we demonstrate the benefits of PATRONUS through case studies. We explain the way to adopt PATRONUS to these cases individually.

6.1 One-sided Data Structure

We mainly focus on two one-sided data structures (ODS), i.e., the start-of-the-art RACE hashing [59] and a concurrent queue [17]. Other ODSs, such as the hashing-based ones [51], the tree-based ones [50, 58], and the skip list [35], are similar.

The RACE hashing is an RDMA-conscious extendible hash table. It purely uses one-sided verbs and leverages `RDMA_CAS` for the lock-free remote concurrency control. The concurrent queue follows the design in [17]; it is implemented as a lock-free linked list of segments, with each segment containing multiple entries.

Necessity for protection. Inserting (removing) elements to (from) the data structure involves memory allocations. Since the remote data structure is shared by multiple clients, the race of memory reallocation, especially invoked from other clients, turns any concurrent one-sided requests into stray access. Specifically, RACE hashing uses copy-on-write (CoW): updating a new value involves freeing the old KV block $\langle K, V_{old} \rangle$. A seriously *delayed* client, e.g., due to network traffic or scheduling, may post one-sided access to V_{old}, the already unowned memory. Similar situations apply to any one-sided data structures involving memory management. Note that this race is hard to address from the design of data structures because the delay is nondeterministic.

Necessity for performance. The one-sided data structures are the essential building block of applications in remote memory; their efficiency determines the performance of the system. The data structures typically support millions of operations per second with microsecond-scaled latency, asking for high-performance protection at the same level.

Adoption of PATRONUS. For RACE hashing, we take insertion as a concrete example. In the vanilla implementation, insertion takes four steps in the common path. (*i*) Allocate a KV block and write the KV to the block. (*ii*) Read the bucket in the subtable. (*iii*) Link the KV block into the bucket via CAS. (*iv*) Re-read the bucket to detect duplicity. To adopt PATRONUS, we use our `allocate` API for KV block allocation (and the permission) and use one `acquire` for the permission to access the subtable. Among the four RDMA operations (one write, two reads, one CAS), two PATRONUS operations are introduced (one `allocate` and one `acquire`). Note that the subtables, as the metadata, are (re)-accessed frequently; therefore, the active permission to the subtable can be re-used several times, possibly across insertions.

For concurrent queue, it is implemented as a lock-free linked list of segments, with each segment containing multiple entries. At insertion, the client tries to fetch an index of an available entry slot from the segment via `FAA`, and fill the entry slot via `write`. If failed (i.e., the segment is full), the client allocates a new segment and links it to the back of the list via `CAS`. The concurrent queue also contains a metadata block maintaining the (possibly stale) head and tail of the linked list. With PATRONUS, each new segment introduces one `allocate` for allocation and one `acquire` for the access permission to the segment. Each client also maintains a prolonged permission to the metadata block.

6.2 Function as a Service

The FaaS is a cloud computing paradigm where the applications are developed and served at the unit of functions. Each function runs in a virtualized environment for isolation and performance fairness. We consider that the FaaS platform equips RM as an external medium for data storage.

Necessary for protection. In the FaaS system, functions submitted by different users access shared remote memory

CPU	Xeon Gold 6240M @2.6 GHz, 32 logical cores, with hyperthreading enabled
RAM	186 GB 2666 MHz DDR4
NIC	Mellanox MT27800 ConnectX-5 Family
OS	18.04.5 LTS, Linux 4.15.0-153

Table 3: Experimental cluster configuration. The evaluation was carried out on a 4-node cluster.

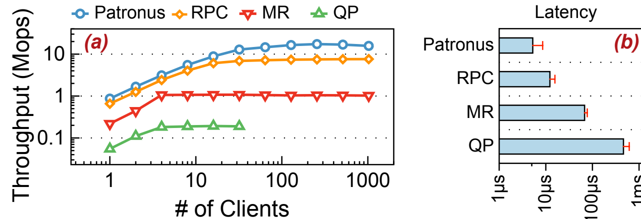

Figure 5: The throughput *(a)* and latency *(b)* of random access of RM with PATRONUS and other protection techniques. QP does not scale beyond 32 clients per machine.

simultaneously. It asks for the isolation of remote access in addition to the existing isolation of local memory and storage. With PATRONUS, future FaaS systems can offer sandboxed remote memory to functions with flexible control over which function is allowed to access which piece of remote memory.

Necessity for performance. First, functions in FaaS have a low bootstrap latency. The state-of-the-art FaaS systems [9, 14, 27, 33] enable microsecond-scaled ultra-low bootstrap latency in the case of hot starts, emphasizing the need for low-latency permission management. Second, functions scale out quickly to millions of instances [33]. Even if each function accesses remote memory once, it introduces a flood of permission acquisitions, asking for high throughput permission management to meet performance needs. Finally, functions access remote memory on demand. Functions are spawned dynamically in response to user requests, and remote access from functions can not be known in advance. It involves permission management in the critical path, precluding the optimization of pre-acquiring permission.

Adoption of PATRONUS. Functions access remote memory on bootstrap, do some calculations, and exit. With PATRONUS, for each data on RM, one `acquire` call is introduced on function bootstrap and one `revoke` is introduced on exit. Re-access to the same data shares the same permission from one `acquire`. Specifically, for the image processing and data analysis workloads we evaluated in the experiment, functions call `acquire` for permission to access the input image and in-memory database respectively. We assume cascadingly invoked functions are executed in the same container so that they can share permission (communicate) with local memory (a technique called *sequence function chain* [8, 16]).

7 Evaluation

Compared mechanisms. We adopt **PATRONUS** to enable protection in various workloads and compare it against three

Name	**(Abbr)**	**# of MW**	**# of RPC**
Baseline		2 + 2	2
Delay Unbind	(+ Delay)	2 + **1**	2
Use Contiguity	(+ Cont)	**1** + 1	2
MW Handover	(+ HO)	1 + **0** †	2
Lease Expire	(+ Expr)	1 + 0 †	**1**

Table 4: A summary of techniques for reducing the permission management overhead (§5.4). The **# of MW** column reports *binding* + *unbinding* operations. † means at probability.

mechanisms used in existing RM systems. (*i*) Re-registration of memory region (**MR**), representing the mechanism adopted by FaRM [12] and Octopus [34]. (*ii*) Modification of QP flag (**QP**), used by uPaxos [3]. (*iii*) Using **RPC** in the data path (no permission acquisitions needed), used by AIFM [43] and Redy [56]. Finally, **Unprot** stands for the vanilla implementation of workloads without any protection.

Experimental setup. We perform the evaluations on a cluster with 4 nodes. Table 3 summarizes the configuration. One node acts as the memory node with limited use of 4 CPU cores. The others are compute nodes with 32 cores. We bind each client thread to a core; for more than 32 clients, we spawn coroutines in each thread to simulate a larger deployment. On reporting latency, we disable coroutines to avoid the schedule variance. The number of clients reported is per machine.

7.1 Overall Performance

Experimental setting. We performed an experiment to reveal the overall data path performance of PATRONUS and compared mechanisms. In the experiment, each client randomly accesses 64 B within a large memory region. The client will re-access the same address three times while using the same permission, representing the common use cases with space locality [12, 52]. The effective access throughput and latency are reported in Figure 5.

Result. Among these techniques, PATRONUS performs the best and only `RPC` can keep pace with it. The performance gap will be enlarged significantly for a larger IO size because `RPC` pays extra overhead of memory copy for each access, but the MW overhead that PATRONUS pays is irrelevant to the size. The performances of `MR` and `QP` are not comparable to PATRONUS. The MR registration is expensive, because it is synchronous and incurs kernel involvement (§2.1). The latter requires modification to the QP flag, which includes the complex QP state management overhead in the RNIC. The QP-based solution also precludes sharing QPs among clients; therefore, we can not evaluate it with more clients.

7.2 Effect of Software Co-design

In this section, we evaluate the effect of the software co-design that makes PATRONUS achieve all the goals.

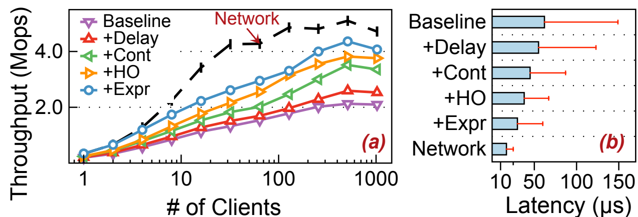

Figure 6: Throughput *(a)* and latency *(b)* of permission acquisition under different optimizations.

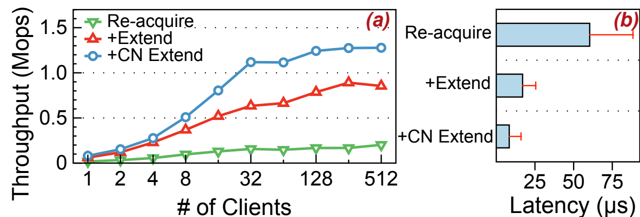

Figure 7: Comparison of not allowing extension (Re-acquire), using RPC for extension (Extend), and our CN-collaborated extension technique (CN Extend). Reporting throughput *(a)* and latency *(b)*.

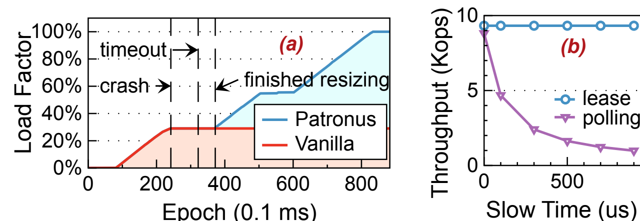

Figure 8: *(a)* The load factor of RACE hashing under client crash for vanilla implementation and PATRONUS. *(b)* The comparison of polling and leases with clients of different fail-slow degrees.

7.2.1 Performance of Permission Management (G#1)

Experimental setting. We evaluate the performance of permission management by breaking down the techniques we adopt. Table 4 summarizes the technique. Besides the three techniques described in §5.4, we additionally consider the lease semantics as the final technique, which effectively eliminates the overhead of `revoke` RPC.

To evaluate the performance, we saturate the system with enough clients to acquire (and revoke) permission over 64 B areas with random addresses. We report the throughput and latency of permission acquisitions in Figure 6.

Result. The combination of all the techniques effectively leads to a performance close to the network bound (bare RPC performance). The eventual throughput is more than 1 Mops per core, which is only achievable with our effort in software co-design, considering that additional overhead besides MWs also exists in the system, such as memory management, RPC, and lease management. In theory, we reduce the overhead of managing a full permission lifecycle to one MW operation and one RPC (the last line in Table 4), which doubles the performance as the baseline and, we believe, exploits the true potential of the hardware.

7.2.2 CN-collaborated Extension (G#2)

In this section, we demonstrate the necessity of the extension API and the effectiveness of our CN-collaborated extension technique (§5.3).

Experimental setting. We evaluate three cases: no extension API, the naive RPC-based extension, and our CN-collaborated extension technique. Without extensions, the unexpected permission expiration requires another `acquire` call to get the permission again (denoted as `Re-acquire`). The RPC-based implementation allows extension but in a naive way, i.e., uses an RPC to notify the memory nodes (`+ Extend`). Finally, our technique (`+ CN Extend`) offloads the management overhead to the CN. In the evaluation permission extends eight times.

Result. Figure 7 reports the throughput and latency. Without the extension, the `Re-acquire` brings both extra MW operations and RPC overhead, which bottlenecks the system seriously and gives only 202 Kops. The RPC-based extension implementation, although saves unnecessary MW operations, still introduces the RPC overhead to bottleneck the system. The CN-collaborated technique reduces both the MW and RPC overhead, effectively producing a $\times 6$ performance gain.

7.2.3 Effect of Lease Semantics (G#2)

We evaluate how the lease semantics enables the system to resume progress when the client crashes while holding exclusive permission. We use resizing in RACE hashing [59] as a case.

Experimental setting. In the experiment, clients are concurrently accessing the hash table while resizing occurs. RACE hashing does not allow cascaded resizing, so the resizing client accesses the metadata (i.e., the resizing subtable) in an exclusive way. The resizing client crashes at epoch 240, leaving the orphaned exclusive permission (or an orphaned lock in the vanilla design) in the system, potentially causing deadlocks. We compare PATRONUS against the vanilla design.

Result. Figure 8 *(a)* shows the load factor of the hash table while clients are concurrently loading data into the table and the resizing client crashes. In the vanilla design, the orphaned lock results in deadlock and prevents the following insertions into the table (red line in the figure). With PATRONUS, the exclusive permission to the metadata can be re-granted to the other concurrent clients after the permission expires. The other clients resume progress and load the table full.

7.2.4 Compare QP polling to leases (G#2)

Experimental setting. QP polling (`polling`) is an alternative approach to leases (`lease`) where memory nodes periodically issue RDMA operations to each QP as heartbeats. On heartbeat timeout, which we set to the same value of lease time (100 µs), the memory node suspects that the compute node has crashed and reclaims the permission. We report the throughput of exclusive permission acquisitions under the anomalies where clients fail slow (get hanged due to network traffic or

Name	w/o	w/
Failure Reported	769 µs	
Promote QP	-	78 µs
Notify QP Failure	8 µs	-
Recover QP	1004 µs	-
Summary	1012 µs	78 µs (8 %)

Table 5: Latency breakdown of handling QP faults with (w/) or without (w/o) the spare QPs technique.

scheduling); a zero slow time denotes the case of normal clients.

Result. Figure 8 *(b)* shows the throughput with normal and slow clients. With normal clients (*zero* slow time), `lease` performs slightly better than `polling` (6 % better) because it allows the lease to expire itself, saving one `revoke` call than `polling`. With slow clients, `lease` is able to retain performance by timely expiration while `polling` does not detect it and degrades performance seriously.

7.2.5 Spare QPs for Concealing Interruption (G#3)

In this section, we evaluate how the spare QPs can conceal the interruption caused by QP faults with illegal access. We prepare spare QPs and trigger illegal access (an out-of-bound write) deliberately.

Result. We break down the latency with and without the technique in Table 5. After illegal access triggers QP faults, the case without spare QPs needs to go through the QP re-bootstrap procedure, introducing significant overhead (Recover QP, 1004 µs). Since QP recovery needs effort from both sides, the client needs to notify the manager (Notify QP Failure, 8 µs). On the other hand, for the case with spare QPs, only the promotion of spare QPs is required (Promote QPs, 78 µs), which involves a handy resource swap in the software. Therefore, it reduces the interrupted time to 8 %.

The illegal request takes much longer for the RNIC to complete (Failure Reported, 769 µs), measured between posting the request and the error being notified. Unfortunately, this procedure purely happens in the RNIC firmware and is confidential; we are unable to analyze and optimize it. Nevertheless, we expect that this period can be significantly shortened by simple modifications to the firmware for future RNICs.

7.3 Case Study: One-sided Data Structures

Next, we reveal the performance of PATRONUS under realistic workloads. In this section, we focus on two remote one-sided data structures, i.e., RACE hashing [59] and the concurrent queue (adopted from [17, 24]). We omit the QP-based mechanism in the figures because it has much worse performance and does not allow scaling beyond 32 clients per CN (not allow clients to share QP).

7.3.1 Hash Table

Experimental setting. We adopt PATRONUS to RACE hashing [59], the state-of-the-art one-sided extendable hash table. Since RACE hashing is not open-sourced, we implement RACE hashing following the original paper, with all the optimizations described in the paper enabled. We verified that the performance of our version is on par with the one reported in the paper. In the evaluation, we set the size of KV blocks to 4 KB. The key follows Zipfian distribution with skewness parameter 0.99. We also consider memory allocation in the critical path as the extended version of the paper does [59], with a pre-allocation factor of four to amortize allocation overhead. The lease time is set to 100 µs. The detailed adoption of PATRONUS is described in §6.1.

Result. Figure 9 shows the throughput under read-only, 50% read-write and write-only workloads (*a-c*) and the read-only latency (*d*) respectively. PATRONUS performs the best and has a reasonable price for memory protection. The protection only introduces 4 µs to the median latency (+29 %), which is an acceptable price for most use cases. The MR-based mechanism is not scalable under all workloads. It is because insert and query to RACE hashing involve a lot of memory access, generating a flood of permission requests in turn. The MR-based mechanism is unable to meet the performance requirements. The P99 latency of MR degrades severely because of the synchronous API it exposes. The RPC mechanism gets better, but it is still inferior due to the memory copy overhead.

7.3.2 Concurrent Queue

Experimental setting. The concurrent queue is implemented as a lock-free linked list of segments; each segment contains 1024 entries. The lease time is set to 100 µs. The implementation and adoption of PATRONUS are described in §6.1.

Result. Figure 10 reports the throughput and latency of inserts with the variety of producers. The performance of PATRONUS is very close to the theoretical upper bound without protection. The reason for the high efficiency is that the overhead of one PATRONUS operation can be amortized into multiple insert operations. Nevertheless, the MR-based solution is still insufficient in terms of throughput because its overhead is too high to amortize (17.5 % throughput as PATRONUS). The RPC-based solution also gets inferior performance because it uses two-sided verbs in its data path, introducing overhead to the limited CPU cores on memory nodes. We also vary segment size from 64 to 1024 to reveal the effect; PATRONUS is $\times$5.18 to $\times$1.78 better than MR (not shown for space limits).

7.4 Case Study: Function as a Service

Description. To evaluate how PATRONUS performs with the FaaS platform, we adopt ServerlessBench [54], a thorough benchmark with representative realistic serverless workloads. We consider two typical applications in TC4 of ServerlessBench, i.e., image processing and data analysis. The former is one of the most popular workloads in the cloud [5], which comprises five functions in the chain to extract metadata and generate the thumbnail of the input image. The data analysis application is a workflow that analyses the salary of employees, triggered by data alteration in the database.

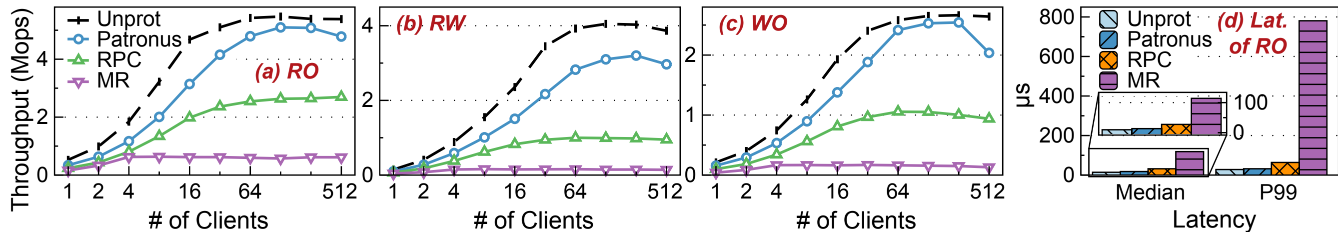

Figure 9: Performance of RACE hashing. *(a-c)* The throughput under read-only (RO), 50%-mixed read-write (RW), and write-only (WO) workloads. *(d)* The latency under RO workload.

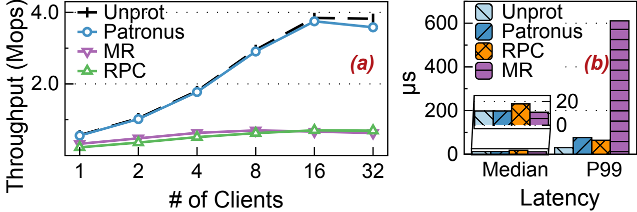

Figure 10: Throughput of producers for the one-sided concurrent queue.

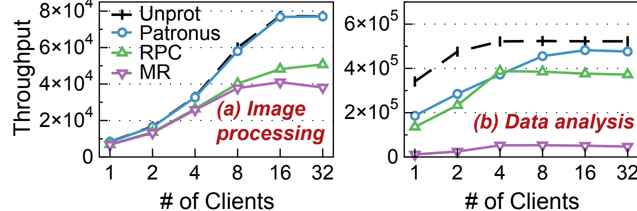

Figure 11: Performance of two serverless applications.

Experimental setting. In the evaluation, we launch enough functions to saturate the system. We assume hot starts for all the functions, excluding the overhead of disk IO and container bootstrap. The lease time is set to multiple times of function lifetime so that lease extensions are rare. The adoption of PATRONUS is described in §6.2.

Result. For the image processing application (Figure 11 *(a)*), PATRONUS has a performance close to the unprotected case. The reason is that generating the thumbnail is a CPU-intensive task, and thus the bottleneck shifts from the protection overhead to the CPU computation. Besides, MW has a constant overhead over the memory size; therefore, the protection performance remains constant even with larger images. On the contrary, the overhead of MR and RPC is so high that they still bottleneck the system even in this CPU-intensive case.

For the data analysis workload (Figure 11 *(b)*), it is more IO-intensive than the previous workload and therefore a wider performance gap is shown between PATRONUS and the unprotected case. Nevertheless, PATRONUS still performs the best and the gap is shortened with more concurrent clients (≥ 8) in the system.

8 Related Work

The development of fast networks, especially the emergence of RDMA, leads to a wide discussion on resource disaggregation [18, 21, 23, 44, 45]. Among them, remote memory is one of the most typical forms of disaggregation, and it has gained much research interest in the last decade [2, 15, 31, 32]. To our best knowledge, no prior RM system has provided efficient protective interfaces with commodity RNICs.

Performance-oriented RM. A wide range of research on RM focuses on optimizing performance with customized hardware. Kona [11] eliminates the virtual memory overhead with a new architecture. Other work [4, 7, 10] extends the RDMA interface for richer semantics. StRoM [47] adopts the idea of near-data processing by performing task offload to the smart remote memory. On the contrary, PATRONUS focuses on the less-discussed access protection issue, which is neglected by these systems. PATRONUS runs on unmodified commodity hardware, allowing a lower cost and a wider deployment.

Protective interfaces. Some RM systems provide access protections with customized hardware, such as programmable switches [29], FPGA [22], and architectural modifications [28]. PATRONUS is designed for commodity hardware, serving as a ready-to-use solution for existing data centers. On the other hand, transactional RM systems [12, 52, 55] also provide protection for data consistency with the transaction interface. However, the transaction semantics is overkilled for most cases (e.g., data structures) because it introduces the expensive overhead of transaction logging and distributed commit protocol.

9 Conclusion

In this paper, we designed, implemented, and evaluated PATRONUS, a protective remote memory system. PATRONUS achieves high performance under all situations by hardware and software co-design. Deployed to realistic applications, it performs $\times 5.2$ better than all the competitors and introduces acceptable overhead ($\leq 27.7\,\%$).

Acknowledgments

We sincerely thank our shepherd Hyungon Moon and the anonymous reviewers for their valuable feedback. This work is funded by the National Natural Science Foundation of China (Grant No. 61832011, 62022051) and the National Key R&D Program of China (Grant No. 2021YFB0300500).

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 121–127, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[4] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 120–126, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[6] 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*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Apache OpenWhisk. http://openwhisk.apache.org/. Accessed: 2022-09-01.

[9] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[10] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. Prism: Rethinking the rdma interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 228–242, New York, NY, USA, 2021. Association for Computing Machinery.

[11] 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*, ASPLOS '21, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery.

[12] 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, Seattle, WA, April 2014. USENIX Association.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[16] Fn Protect. https://fnproject.io. Accessed: 2022-09-01.

[17] Folly UnboundedQueue. https://github.com/facebook/folly/blob/main/folly/concurrency/UnboundedQueue.h. Accessed: 2022-09-01.

[18] 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, Savannah, GA, November 2016. USENIX Association.

[19] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.

[20] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. Resource-centric serverless computing. *arXiv preprint arXiv:2206.13444*, 2022.

[22] 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*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014.

[26] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[28] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. Deact: Architecture-aware virtual memory support for fabric attached memory systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 453–466. IEEE, 2021.

[29] 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*, SOSP '21, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[31] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, jun 2009.

[32] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[33] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[34] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[35] Teng Ma, Dongbiao He, and Ning Liu. Hybridskiplist: A case study of designing distributed data structure with hybrid rdma. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 68–73, 2021.

[36] 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, San Jose, CA, June 2013. USENIX Association.

[37] 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*, SOSP '21, page 212–227, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[39] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[40] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[41] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, jan 2010.

[42] RDMA Memory Window. `https://docs.nvidia.com/networking/pages/viewpage.action?pageId=25138102`. Accessed: 2022-09-01.

[43] 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. USENIX Association, November 2020.

[44] 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, Carlsbad, CA, October 2018. USENIX Association.

[45] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 18–28, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[47] 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*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] 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. USENIX Association, July 2020.

[49] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery.

[50] 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*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Tinggang Wang, Shuo Yang, Hideaki Kimura, Garret Swart, and Spyros Blanas. Efficient usage of one-sided rdma for linear probing. In *Eleventh International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (AMDS'20)*, 2020.

[52] 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, Carlsbad, CA, October 2018. USENIX Association.

[53] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, Carlsbad, CA, July 2022. USENIX Association.

[54] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[55] 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, Santa Clara, CA, February 2022. USENIX Association.

[56] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 15(4):766–779, dec 2021.

[57] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

[58] 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*, SIGMOD '19, page 741–758, New York, NY, USA, 2019. Association for Computing Machinery.

[59] 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. USENIX Association, July 2021.