RDMA Reads: To Use or Not to Use?

Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro
Microsoft Research

Abstract

Fast networks with RDMA are becoming common in data centers and, together with large amounts of main memory in today's machines, they allow unprecedented performance for traditional data center applications such as key-value stores, graph stores, and relational databases. Several systems that use RDMA have appeared recently in the literature, all showing orders of magnitude improvements in latency and throughput compared to TCP. These systems differ in several aspects. Of particular interest is their choice to use or not to use RDMA reads. RDMA reads are appealing as they perform better than RPCs in many cases, but RPCs are more flexible as the CPU can perform additional operations before replying to the client. For example, systems that use RPCs often use indirection per object which makes it easier to de-fragment memory, evict cold objects to a storage tier, support variable sized objects, and perform similar tasks. Indirection per object can be implemented in systems that use RDMA too, but it doubles the cost of object reads, so it is avoided. The designers of RDMA-based systems are left in a dilemma: should they use RDMA reads or not? We help answer this question by examining the trade-off between performance and flexibility that the choice implies.

1 Introduction

Fast networks with RDMA [27, 28] are becoming common in data centers as their price has become similar to the price of Ethernet [12]. At the same time, DRAM has become cheap enough to make it cost-efficient to put 256 GB or more of DRAM in commodity data center servers. Memory sizes are going to increase further as the price of DRAM keeps decreasing and new, higher density, memory technologies such as 3D XPoint [30], PCM [7], and memristors [20] are developed. Main memory is also becoming non-volatile. These new memory technologies are all non-volatile. Non-volatile DIMMs [25, 35] and approaches based on cheap distributed UPS [3] make even DRAM non-volatile. These hardware trends allow building scale-out systems that store data in memory and access it over fast networks to allow unprecedented performance for data center applications.

Scaling out systems like relational or graph databases is challenging with traditional Ethernet networks because networking overheads are so high that scale-out solutions often need to run on tens of machines to match the performance of a single-machine alternative [23]. This makes scale-out with Ethernet unattractive. Partitioning reduces communication, but many workloads, e.g. graphs cannot be ideally partitioned, so they have to access data on remote machines. Batching amortizes communication costs, but it also increases latency, which is not a good trade-off for latency sensitive applications. RDMA networks help reduce the overhead of scaling out by providing micro-second latency access to the memory of remote machines. This is particularly

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

beneficial for applications that have irregular access patterns and that are latency sensitive, like key-value stores, graph stores, and relational databases. RDMA networks also provide high bandwidth and high message rates, enabling systems that have high throughput and low latency at the same time. High message rates are particularly important as modern applications store small objects [1] so network bandwidth is rarely the bottleneck.

Several systems that take advantage of the new hardware have recently appeared in the literature, all showing orders of magnitude improvement in latency and throughput compared to traditional TCP-based systems [3, 7, 12, 15, 21]. They commonly provide a general programming abstraction with distributed transactions to simplify applications built on them. Transactions manipulate objects in memory atomically, allowing applications to ignore concurrency and failures in a distributed system.

Despite having similar application interfaces, RDMA-based systems differ in several aspects, notably in their choice to use or not to use RDMA reads. RDMA reads allow a machine to fetch data from memory of remote machines without involving the remote CPU. This can have significant throughput and latency advantages compared to RPC—2x to 4x in our measurements. For this reason some systems advocate the use of RDMA reads [3, 12, 21, 22]. On the other hand, RPCs are more flexible as the CPU can perform additional operations before replying to the RPC. This is commonly used in systems that only use RPCs to implement a level of indirection per object [10, 7, 8]. This level of indirection makes it easy to transparently perform system-level tasks like memory de-fragmentation, cold object eviction, resizing objects, and similar. Systems that use RDMA reads could implement indirection per object, but this would mean that two RDMA reads need to be issued to read an object—one for the indirection pointer and the other for the object. For this reason they do not implement per-object indirection. Note that this loss of flexibility mainly impacts system-level tasks. Using RDMA reads does not prevent use of RPCs so applications can use them in the same way as on RPC-only systems.

Designers of RDMA-based systems are left in a dilemma: should they use RDMA reads or not? Is the performance benefit worth the loss of flexibility? Our experience suggests that RDMA reads are a good choice in many cases. Applications like relational and graph databases keep enough information about data layout to support de-fragmentation, eviction, and similar tasks and many applications can be modified to do the same. But RDMA reads are not always the right choice. It is sometimes worth sacrificing some performance for flexibility.

We believe that in the future we will not have to make this choice. We expect to see systems with carefully designed software and hardware components that have both the flexibility of RPC and the performance of RDMA reads and writes. How to build these systems is still an open question.

In this paper we examine the implications of using RDMA reads by providing background on RDMA networks (Section 2), discussing the trade-off between performance and flexibility (Sections 3 and 4), contrasting the design of RAMCloud [10, 13, 15, 19], an RPC-only system, and FaRM [3, 5], a system that uses RDMA reads, to illustrate the differences (Sections 5 and 6), and describing challenges that lay ahead (Section 7).

2 RDMA

RDMA networks implement transport protocols directly in hardware and expose them in user mode. The latest NICs have 100 Gbps of bandwidth, 1-3 micro-second latency, can process 200 million messages per second and can issue . a network operation using less than 1500 cycles in our measurements. This is orders of magnitude better than TCP in all respects. RDMA networks implement reliable connected (RC) and unreliable datagram (UD) transports. The former is reliable, in-order transport that supports large transfers, similarly to TCP. The latter is unreliable, unordered transport, that supports datagrams up to 4 kB, similarly to UDP.

In addition to send/receive, RC transport supports RDMA reads and writes. RDMA requests are sent to the remote NIC which performs them without involving the CPU. To enable RDMA access to a block of memory, it has to be registered with the NIC. Registration pins the memory, assigns it an identifier, and maps it in the NIC. Machines that are sent the identifier can access the block of memory using RDMA any number of times without further involvement from the server's CPU. Memory can be de-registered to prevent further RDMA accesses.

Several problems need to be solved to achieve maximum performance with RDMA reads [3]. Most NICs have little memory, so they keep state in system memory and use NIC memory as a cache. This allows supporting larger clusters with little memory, but results in performance degradation when too much state is used at the same time, as the state needs to be fetched from system memory over PCI bus. We have encountered issues with caching of mapping tables for registered memory blocks and connection state. When memory blocks are allocated using a standard allocator, they get mapped using 4 kB pages, so mapping just a few MBs fills the cache. We have measured 4x performance impact with just 64 MB of registered memory. To solve this problem, memory needs to be registered using bigger pages. For instance, FaRM [3] uses a special allocator that allocates 2 GB naturally aligned pages. This enables the NIC to map each 2 GB page as a single entry, allowing FaRM to register hundreds of GBs of memory without impacting performance. Similarly, using too many connections to fit in the cache degrades performance—we measured up to 4x performance degradation. To alleviate this problem, connections can be shared between threads, but this introduces synchronization overheads—we measured 2-3x increase in CPU cycles to issue RDMA read when sharing connections.

The most widely deployed types of RDMA networks today are Infiniband [27] and RoCE [28]. Infiniband offers slightly better performance than RoCE, but it is also more expensive. Infiniband supports link-level flow control, but does not provide other forms of congestion control. This makes it suitable to clusters of up to a few hundred machines connected with a single switch. Infiniband does not offer good performance to applications that use traditional TCP/IP stack, meaning that most legacy applications perform poorly. In contrast, RoCE runs on lossless Ethernet with priority flow control. This provides good support for applications that use TCP/IP. RoCE supports congestion control for connected transport [15, 24]. This means that RoCE scales better than Infiniband and can support data center scale networks. For these reasons, RoCE is deployed in data centers where it is used for traditional data center workloads using TCP/IP or RDMA stacks. In this paper we are mainly considering RoCE as it is more widely available.

3 Performance

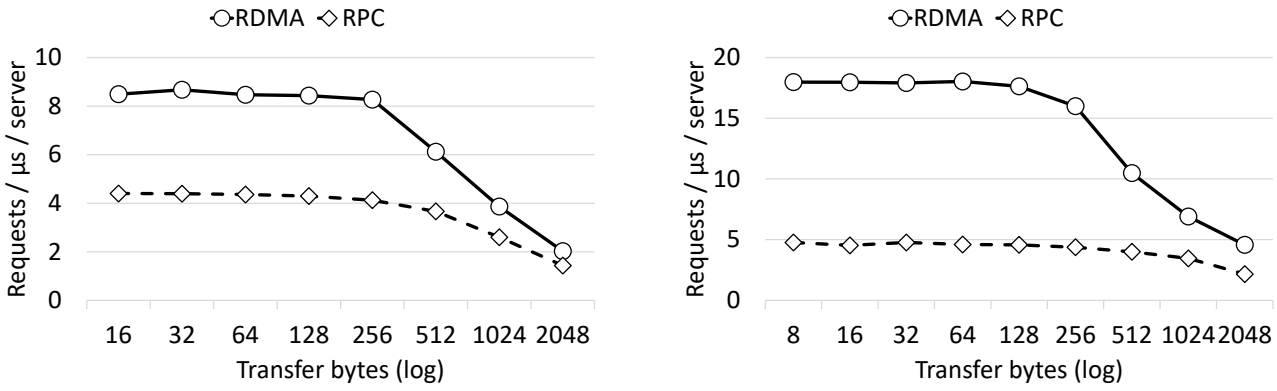
3.1 Connected transport

RDMA reads are more CPU efficient than RPCs for simple operations, like reading an object from memory of a remote machine, because they do not use any CPU cycles on the machine where the object resides. These simple operations are predominant in many applications, like graph databases, key-value stores, and OLTP workloads, which makes RDMA appealing. CPU efficiency translates to performance gains on modern hardware, as most systems are CPU bound—storage and network are out of the performance path, so CPU is the main bottleneck. With NICs that can process 200 million messages per second, even the latest Intel's server CPUs[1] provide a budget of just above 1000 cycles per network operation, which is not enough even to issue all the operations the NIC can process. Systems are expected to remain CPU bound in the future too, as advances in network technology are projected to outpace advances in CPUs [29]. RDMA reads also improve latency, especially on a busy system, as they do not involve the CPU on the target machine, and hence do not have to wait for the CPU to become available.

In addition, RDMA reads exchange fewer network messages than RPC implemented on RC transport. This means that RDMA reads perform better than RPC even when CPU is not the bottleneck, such as with the previous generation of network hardware, which is predominantly in use today.

Figure 1 compares performance of RDMA reads and RC RPCs for small reads. We ran an experiment where each machine in a cluster reads data from memory of all other machines. Each read is to a randomly selected machine and a random piece of memory in it. Reads are performed either directly with RDMA or using RPC implemented with RDMA writes [3]. Figure 1a shows per-machine throughput on a cluster of 20 machines each

[1] At the time of writing Intel's Xeon E7-8890 v4 with 24 cores at 2.2 GHz.



(a) 1 NIC (network-bound)

(b) 2 NICs (CPU-bound)

Figure 1: Per-machine RDMA and connected RPC read performance.

with a single 40 Gbps Mellanox ConnectX-3 NIC. RDMA reads outperform RPC by 2x because the bottleneck in this setup is the NIC's message processing rate. Figure 1b shows what happens when CPU is the bottleneck. It shows per-machine throughput on the same benchmark running on a 90-machine cluster with two 56 Gbps Mellanox ConnectX-3 NICs in each machine. With the higher network message rates provided by two NICs, using RDMA reads provides even more benefit as it outperforms RPC by 4x. This is because the system is CPU bound. Using RPC makes it almost CPU bound on this simple benchmark even when using a single NIC, so providing more network capacity does not help improve RPC throughput significantly.

3.2 Unreliable transport

Using unreliable datagrams for RPC has some performance benefits over using connected transport on current hardware [7]. The main advantage is that UD does not use connections and is, therefore, not impacted by connection caching issues and the overheads of connection sharing described in Section 2. In addition, UD RPC can exploit application-level semantics to send fewer messages than RC RPC. For instance, an RPC response can be treated as the ack to the request meaning that a separate ack message does not need to be sent. RC RPC cannot do this as these application-level semantics are not available to the NIC which is responsible for sending acks. UD also allows more batching when issuing sends and receives to the NIC than RC. UD can send any group of requests as a batch whereas RC can batch only requests on the same connection, which amortizes costs of PCI transactions more efficiently.

The disadvantage of using UD is that, unlike RC, it does not implement congestion control, reliable delivery, and packet fragmentation and re-assembly in hardware, so all of this has to be done in software, which introduces CPU overheads. Of these, congestion control is the most challenging to support. Several promising approaches for congestion control in RoCE exist [15, 24], but they do not work with UD. Without congestion control, UD cannot be used in data centers. Packet re-transmissions, fragmentation and re-assembly can be implemented in software, but this would introduce overheads e.g. because they might require additional message copies.

Work on FaSST [7] has shown that UD RPC that does not implement any of the above features can even outperform RDMA reads in some specific scenarios. To achieve maximum performance, FaSST RPC exposes a very basic abstraction— unreliable send/receive of messages that can fit in a network packet (2 kB for RoCE). If messages are lost, FaSST crashes the machine that observes the loss. FaSST also does not implement any congestion control. This design was motivated by the setup the authors used—a single-switch cluster with an expensive Infiniband network and simple benchmarks that only send small messages. This is very different from large RoCE deployments. For instance, the authors have observed almost no message loss in their exper-

iments, which is why they decided to handle message loss by crashing the machine that detects it instead of re-transmitting messages.

There are two problems with UD RPC performance argument. First, using it in realistic deployment scenarios would require implementing the missing features, which is not trivial and would introduce overhead. Until these are implemented, it is hard to make any conclusions about performance. In contrast, RC RPCs and RDMA reads can readily be used in any setting. Second, the core of the performance argument relies on the specifics of the hardware available today and new hardware is likely to make it invalid. Having more memory on NICs or better handling of connection cache misses would reduce or even eliminate the need to share connections across threads which would reduce overhead [33]. It was already shown that a NIC integrated with the CPU can support several thousands of connections with no overhead [2].

4 Flexibility

RPCs are more flexible than RDMA reads because the CPU can perform additional operations before sending the response. Systems that only use RPCs use this to implement a level of indirection per object (typically as a hashtable), which allows them to move objects transparently to the application to e.g. change their size, de-fragment memory, or evict them to cold tier. It is possible to implement indirection per object with RDMA reads too. A system could store a pointer to an object in a known location and update that pointer whenever the object moves. Object reads would require two RDMA reads—one to read the pointer and another to read the object. This would make the system more flexible, but would also double the cost of reads, eliminating much of the performance benefit of using RDMA reads. As Figure 1b shows, such a scheme would still be faster than using RC RPCs in CPU-bound workloads, but systems typically avoid it to achieve maximum performance.

To support efficient reading of objects over RDMA, each machine in the cluster needs to know exactly where each object lives in the memory of its host without additional communication steps. To achieve this, machines cache addressing meta-data on the first access and update it only when it changes. Storing per-object addressing meta-data for all objects on all machines would introduce prohibitive communication and memory overheads.[2] To reduce these overheads addressing is done at a coarser granularity. This is different from reading with RPC where the reader only needs to know which machine holds the object so it can send an RPC to it. The machine storing the object is the only one that needs to know the exact location of the object in memory, making indirection per object local and efficient.

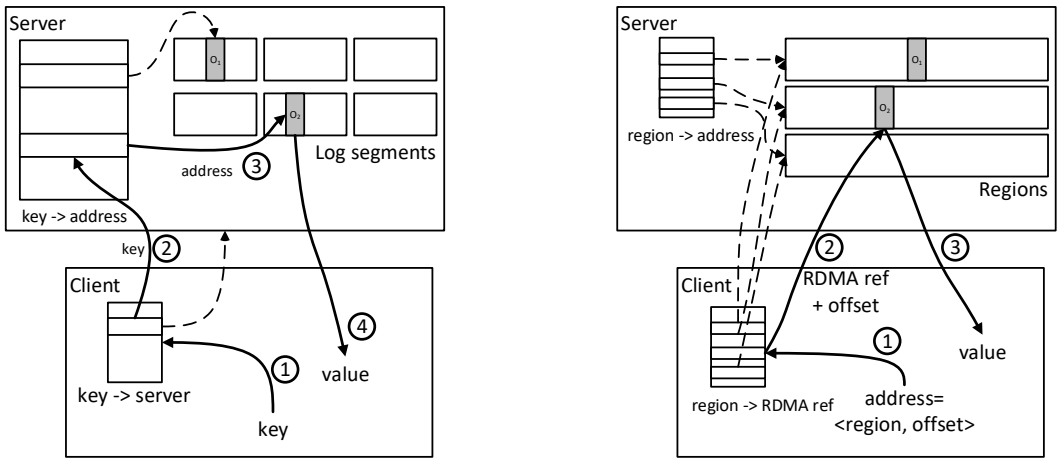
Using RDMA reads also makes it harder to track accesses to objects from remote machines. The NIC does not maintain detailed access information and the CPU on the host machine does not observe incoming read requests. This makes it much harder to identify cold or hot objects and to keep performance metrics, as this has to be done in a distributed fashion.

It is important to note that these flexibility limitations mainly impact system-level tasks. Nothing prevents the application from using RPCs when beneficial e.g. to exploit locality by shipping an operation to the machine that holds the data the operation accesses.

5 RAMCloud

Several RDMA systems, such as RAMCloud [10, 13, 15, 19], HERD [8], and FaSST [7], are RPC-only. We describe RAMCloud here in more detail to illustrate the design of RPC-only systems.

[2] In a workload with average object size of 128 bytes and addressing meta-data of 8 bytes per object, clusters with more than 16 machines would store more addressing meta-data than actual data when meta-data is fully replicated.



(a) RAMCloud

(b) FaRM

Figure 2: RAMCloud and FaRM memory layout. Dashed lines are pointers. Solid lines with numbers next to them are steps to read an object. In RAMCloud, the client maps the object key to the server using a local table in step 1, it sends the message to the server, which uses a local hashtable to find the address of the object in step 2, the server fetches the object from local memory in step 3, and sends the value back to the client in step 4. The server's CPU is involved in steps 2, 3, and 4. In FaRM, the client maps the region part of the address to the RDMA reference of the region using a local region mapping table in step 1, adds the offset to the the reference and issues an RDMA read in step 2, and the server's NIC returns the value in step 3. The server's CPU is not involved in any step.

5.1 Overview

RAMCloud was the first storage system to keep all data in main memory. It exposes a key-value store with distributed transactions to the users. RAMCloud is optimized for storing objects of several hundred bytes. It keeps only the primary copy of data in main memory. Backups are written to secondary storage. It relies on a small amount of non-volatile memory on backups to batch writes to disk to make logging efficient. RAMCloud achieves latency of 4.7 micro-seconds for reads and 13.4 micro-seconds for small object updates [10]. Backups of data are spread across the cluster to make failure recovery fast. RAMCloud recovers 35 GB of data from a failed server in 1.6 seconds [15].

5.2 Memory management

RAMCloud organizes its memory as a log both in memory and on disk (Figure 2a). Log is split into 8 MB segments. RAMCloud stores each object's address only on the primary, in a hashtable that maps the identifier of the table the object belongs to and its key into object's address. When an object is inserted or updated, a new value of the object is appended to the log and its address is updated in the hashtable.

RAMCloud de-fragments the logs to achieve high memory utilization. It uses two-level de-fragmentation to ensure both high memory space utilization and low disk bandwidth utilization. In-memory de-fragmentation is done at the primary, without communicating with backups or accessing disk. The primary picks a segment and copies live objects from it into a number of 64 kB seglets. This saves memory while keeping the contents of the logical segment unchanged and thus does not require communication with backups. Using small fixed-size seglets instead of allocating a block of memory that can fit all the objects keeps fragmentation low. Full de-

fragmentation is done by scanning the segments in memory, copying live objects from them into new segments, and sending the new segments to backups to be written to disk. RAMCloud runs full de-fragmentation only if running out of space on disk or if there are too many object tombstones in the log. RAMCloud achieves 80-90% memory utilization with little performance impact.

5.3 Flexibility

Having indirection per object through the hashtable gives RAMCloud flexibility. It allows RAMCloud to transparently move objects to de-fragment its logs. RAMCloud can also change size of an object if an application requires it. Evicting objects from memory into a cold tier would also be straightforward—it is enough to simply move the object and update its location in the hashtable.

6 FaRM

A number of RDMA systems, such as Pilaf [12], FaRM [3, 5], DrTM [21], and NAM-DB [22], read objects using RDMA reads. We describe FaRM here in more detail to illustrate the trade-offs involved.

6.1 Overview

FaRM is an in-memory, distributed computing platform that exposes the aggregate memory of a cluster of machines as a single shared address space. Applications manipulate objects stored in the address space by invoking FaRM APIs to transactionally allocate, free, read and write objects. Transactions are fully general and strictly serializable. Similarly to RAMCloud, FaRM is optimized for small objects. Servers in a FaRM cluster both store data and execute application code. FaRM supports atomic reads of remote objects. It inserts cache-line versions into stored objects and uses them to ensure object reads are atomic. In addition to supporting direct reads of remote objects, FaRM allows for function shipping by using efficient RDMA-based RPCs. It also supports allocating new objects close to existing objects to enable locality-aware optimizations.

FaRM protocols were carefully designed and optimized to achieve high throughput and low latency. For instance, FaRM achieves throughput of 140 million transactions per second with median latency of 60 micro-seconds in the TATP benchmark [16] on a 90 machine cluster. It also does 270 million tpmC on a TPC-C [34] like benchmark. Latency of a single small key-value store lookup is 4.3 micro-seconds. FaRM recovers from failures quickly. It takes just 50 ms to get back to maximum throughput after a single-machine failure.

6.2 Memory management

FaRM's memory management scheme is designed to support efficient RDMA reads and low-overhead object allocation. It minimizes the amount of addressing meta-data that needs to be stored across the cluster and eliminates communication during common-case object allocation.

FaRM organizes its address space as 2 GB memory regions (Figure 2b), which are the unit of addressing, replication, and recovery. Regions are replicated in memory, typically on three machines, but replication degree can be configured per region. Region placement is done by a centralized region allocator which assigns region replicas to machines. It places each replica in a different failure domain, and balances the number of regions per machine while attempting to respect placement hints the application can provide. The region allocation information is replicated across the machines in the cluster during allocation and is reconstructed during region allocator recovery. Each region is assigned a 33-bit region identifier in sequence. Addresses of FaRM objects consist of the region identifier and a 31-bit offset inside the region. This addressing scheme is designed to keep addressing meta-data small and to avoid frequent updates. Addressing information is kept per region, so each

machine can cache it for all regions in the system with small space overhead. Region placement changes only when machines fail or join the system, so this cache does not need to be updated frequently.

Regions are split into 1 MB slabs. Object are allocated from slabs. Each slab contains objects of the same size. FaRM supports 256 different size classes between 64 bytes and 1 MB. Slabs are allocated on demand when threads run out of slabs for allocating objects of a particular size. Each active slab is owned by a single thread to avoid thread synchronization during allocation. Slab meta-data is replicated to machines storing region replicas during slab allocation and deallocation. This meta-data is used during recovery.

Object allocations are performed on the primary without any communication with the backups. An objects is marked allocated by setting the allocated bit in its header. Setting the bit does not require any additional messages—it gets set on backups during the commit of the allocating transaction. The allocated bits are used to reconstruct object allocator meta-data on the new primary after failures.

6.3 Flexibility

FaRM's object allocation scheme achieves the goals it was designed for, but it is not as flexible as a typical RPC-only system—it maintains indirection per region instead of per object. This makes it hard for the system to support variable sized objects, memory de-fragmentation, cold object eviction, and similar.

To perform these tasks, FaRM relies on the application and its knowledge of the data layout. The application can move an object as long as it knows where all the pointers to the object being moved are. To do so, it executes a transaction to atomically allocate new space for the object, copy the contents into it, update all pointers to the object, and free the old object. By using a transaction, the application deals with the issues of consistency and fault tolerance during the move—other transactions either see the old pointers and the old version of the object or the new ones. This is suitable for applications such as relational and graph databases that keep track of data layout and pointers between objects, which makes it easy to move objects. Applications that allow arbitrary pointers between objects, on the other hand, cannot easily move objects.

Next we discuss how FaRM and applications using it deal with some of the flexibility limitations.

Object size. When reading an object, the application needs to supply the object's address and size. FaRM cannot infer the size from the address, as the slab-level meta-data required to do this is kept only on the machines storing object replicas, not on all machines in the cluster. In most cases this is not a significant challenge. Applications that impose schema on stored objects, such as databases or language runtimes, know the size of the objects they are reading. Applications that do not typically store a 1-byte FaRM size class with the pointers to the object and use it when performing the read. Some of our applications adopt this approach and it serves them well. By storing the size class in the pointer we can avoid increasing the pointer size and still support 2^{56} bytes in the address space.

Variable sized objects. FaRM's memory management approach prevents it from resizing objects in place. To resize an object, the application needs to move it into a larger (or smaller) newly allocated space. Alternatively, variable sized objects can be represented as a list of chunks. This works particularly well when the object consists of a fixed-sized and a variable-sized part. In those cases it is common that the majority of operations only access the fixed-sized part, so the list-of-chunks approach improves performance. For this reason, some of applications use it even when they have enough information to move objects.

Memory de-fragmentation. FaRM's slabs can get fragmented when an application allocates a large number of objects of a certain size and then frees most of them. This reduces memory efficiency because most slabs will be mostly empty. A typical solution to this problem is to de-fragment slabs by moving live objects to as few slabs as possible.

	Reliable connections	Unreliable datagrams
Scalability	Limited	Good
Congestion control	Supported	No
Reliable	Yes	No
In-order delivery	Yes	No
Message size	2 GB	2 kB
RDMA reads and writes	Yes	No

Table 1: RDMA transport capabilities

De-fragmentation requires knowing which objects to move and how to move them. The former is known by FaRM and the latter by the application, making it impossible for either to perform de-fragmentation on its own. To solve this, we are planning to extend FaRM to invoke application-defined callbacks with addresses of objects that should be moved for de-fragmentation and have the application move the objects. To obtain the space for the new object during de-fragmentation, the application can simply invoke FaRM's allocator. The allocator always prefers mostly full slabs for new allocations to reduce fragmentation.

Dealing with de-fragmentation at the application level in this way instead of performing it transparently by the system can be beneficial as application-level knowledge can be used to optimize the process. For instance, the application can choose to postpone de-fragmentation to avoid increasing load in busy periods, or to completely ignore the callback if it will soon need to allocate objects of the same size.

The application needs more information about object pointers to perform de-fragmentation than for resizing—the callback is invoked just with the address and size of the object to be moved, without providing a path to the object as is the case when resizing. This means that there needs to be a simple way to find the pointers to the object given just its address. For instance, to support de-fragmentation, the application might need to use a doubly-linked list where it would otherwise use a singly-linked list.

Cold objects. Despite the decreasing price of main memory, it still remains much more expensive than SSDs and disks. To reduce costs, some applications store only frequently accessed objects in memory and keep the rest on disk. A system that uses RDMA reads needs to solve two challenges to support this kind of tiering: identifying cold objects and evicting them to the storage tier. Identifying cold objects is hard as the CPU on the host machine does not observe remote accesses and the NIC does not keep detailed access stats. Alternatively, the application can decide which objects to move to the cold tier and when using application-level semantics. For instance, the application might periodically evict objects older than some threshold. Objects can be moved in a similar fashion to de-fragmentation—pointers to the object are either replaced by cold-tier pointers or invalidated when the objects is evicted.

7 Future challenges

7.1 Networking

Table 1 summarizes the advantages and drawbacks of using RDMA RC and UD transports, as discussed in Section 3. It shows that neither transport provides all desired features. UD provides good scalability, but it leaves dealing with congestion control, re-transmissions, and packetization and re-assembly to software. It also does not support RDMA reads and writes. On the other hand, RC has scalability issues because of limitations of available hardware. Building a network transport that provides all desired features remains a challenge.

To build it, we need to first rethink what the important features are. Not all features from Table 1 are equally

important and a simpler transport can be more efficient. For instance, most distributed systems robustly deal with reordered messages, so providing in-order delivery guarantees in the network is not useful. Re-transmission can be more efficient when message semantics are known, so it makes sense to either implement it in software or expose the semantics to hardware. For instance, idempotent operations, like a read-only RPC, can be safely executed multiple times, so there is no need to reliably respond to it, as the operation can simply be re-executed if the response is lost. It might also be viable to build hardware support for the common case, and leave corner cases to software. For instance, optimizing hardware for RDMA reads that fit in a network packet or even not supporting bigger RDMA reads at all would be acceptable to many applications.

NIC hardware can be improved to reduce the CPU cost of issuing RDMA operations regardless of the abstraction it implements. The predominant overheads when interacting with the NIC are PCI transactions used to issue and complete operations. These overheads can be reduced by batching—issuing or completing multiple operations with a single PCI transaction [6]. There are opportunities to allow more batching than today, for instance by allowing applications to issue batches containing operations on different connections, which is not supported today. Alternatively, instead of waiting for PCI messages from the CPU, the NIC could periodically poll request queues in memory. This would remove the need for the CPU to initiate PCI transactions and CPU overheads of initiating them, but it would also increase latency, so it might not always be a good trade-off.

7.2 Hardware acceleration

Widespread hardware acceleration is going to change the way we think about and build systems in the future. With the end of Moore's law nearing, big companies are heavily investing in hardware acceleration. For instance, Microsoft is deploying FPGA Catapult boards [18, 3], Google is deploying custom TPU ASICs to speed up machine learning [31], Intel has announced work on integration of Xeon processors with coherent FPGAs in a single package [26], and Mellanox has a product brief for a ConnectX-3 Pro card with FPGA [32]. This creates an opportunity to build hardware that makes distributed systems better.

The main challenge with hardware specialization is in selecting the right primitives to move to hardware, because of cost and complexity of development, even when using FPGAs. For this reason, primitives like encryption and compression are the best targets—they are relatively simple and they would benefit many applications. Time synchronization, which would make time-based protocols better, or remote read tracking, which could help with detecting cold or hot objects, are also good candidates, as they would be useful in many cases. Implementing parts of a general, widely-used distributed computation platform in hardware would also benefit many applications. We have experimented with hardware implementation of FaRM atomic reads and have seen relatively small but consistent performance improvements. Other operations used by such a platform like pointer chain following, hashtable lookups, and even parts of transaction commit protocol are good candidates for acceleration in hardware.

With the right hardware primitives system builders would not have to choose between performance and flexibility, like they do today with RDMA reads. A system that stores data in a hashtable and implements hashtable lookups in the NIC would give us the best of both worlds—performance of hardware lookups and flexibility of RPC-only systems. Some work on offloading hashtable functionality to FPGA has already been done and the results are promising [8, 9]. The main remaining questions are what the right software-hardware design is, how close to today's NIC's request rates can performance of a hardware hashtable get and how much flexibility will have to be traded off to support efficient hardware implementation.

8 Conclusion

So, should you use RDMA reads or not? As always, the answer is: it depends on the workload. RDMA reads can deliver benefits both in terms of latency and throughput, but they are less flexible than RPCs. We discussed

ways to overcome these flexibility limitations in applications with well defined schemas and data layout like relational and graph databases. RDMA reads are a good choice for those applications. On the other hand, using RDMA reads complicates the system and puts additional requirements on the application, at the extreme wasting space and burning through some of the performance benefit it brings. In the future we might have it all—a fast hashtable implemented in hardware can give us performance of hardware and flexibility of a key-value store, but this will require overcoming a number of challenges.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS 2012.*

[2] V. Basant, P. Rahoul. Oracle's Sonoma processor: Advanced low-cost ' processor for enterprise workloads. *HOTCHIPS 2015*.

[3] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, D. Burger. A Cloud-Scale Acceleration Architecture. *MICRO 2016*.

[4] A. Dragojević, D. Narayanan, M. Castro, O Hodson. FaRM: Fast Remote Memory. *NSDI 2014.*

[5] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. *SOSP 2015.*

[6] M. Flajslik, M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. *USENIX ATC 2013.*

[7] S. Hudgens. Overview of Phase-Change Chalcogenide Nonvolatile Memory Technology. *MSR bulletin, 2004, vol 29, number 11.*

[8] Z. István, D. Sidler, G. Alonso. Building a distributed key-value store with FPGA-based microservers. *FPL 2015*.

[9] Z. István, D. Sidler, G. Alonso, M. Blott, K. Vissers. A Flexible Hash Table Design For 10Gbps Key-value Stores on FPGAs. *FPL 2013*.

[10] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, Diego and S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, S. Yang. The RAMCloud Storage System. *TOCS, September 2015, vol 33, number 3*.

[11] A. Kalia, M. Kaminsky, D. G. Andersen. FaSST: Fast, Scalable, and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. *OSDI 2016*.

[12] A. Kalia, M. Kaminsky, D. G. Andersen. Using RDMA Efficiently for Key-Value Services. *SIGCOMM 2014*

[13] C. Lee, S.J. Park, A. Kejriwal, S. Matsushita, J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. *SOSP 2015*.

[14] C. Mitchell, G. Yifeng, L. Jinyang. Using one-sided RDMA reads to build a fast, CPU-efcient key-value store. *USENIX ATC 2013*.

[15] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *SIGCOMM 2015*.

[16] S. Neuvonen, A. Wolski, M. Manner, V. Raatikka. Telecom Application Transaction Processing benchmark. `http://tatpbenchmark.sourceforge.net`.

[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, M. Rosenblum. Fast Crash Recovery in RAMCloud. *SOSP 2011*.

[18] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, S. Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ISCA 2014*.

[19] S. M. Rumble, A. Kejriwal, J. Ousterhout. Log-structured Memory for DRAM-based Storage. *FAST 2011*.

[20] D. Strukov, G. S. Snider, D. R. Stewart, S. R. Williams. The missing memristor found. *Nature, 2008, vol 453, number 7191*.

[21] X. Wei, J. Shi, Y. Chen, R. Chen, H. Chen. Fast In-memory Transaction Processing using RDMA and HTM. *SOSP 2015*.

[22] E. Zamanian, C. Binnig, T. Kraska, T. Harris. The End of a Myth: Distributed Transactions Can Scale. *VLDB 2017*.

[23] X. Zhu, W. Han and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. *USENIX ATC 2015*.

[24] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, M. Zhang. Congestion Control for Large-Scale RDMA Deployment. *SIGCOMM 2015*.

[25] Agiga Tech. `http://www.agigatech.com`.

[26] D. Bryant. Disrupting the Data Center to Create the Digital Services Economy. `http://itpeernetwork.intel.com/disrupting-the-data-center-to-create-the-digital-services-economy/` *Intel IT Peer Network*, June 18, 2014.

[27] Infiniband Trade Association. `http://www.infinibandta.org`.

[28] Infiniband Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE) *2010*.

[29] Infiniband Trade Association. InifiniBand Roadmap. `http://www.infinibandta.org/content/pages.php?pg=technology_overview`.

[30] Intel. 3D XPoint Unveiled—The Next Breakthrough in Memory Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html`.

[31] N. Jouppi. Google supercharges machine learning tasks with TPU custom chip. `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html` *Google Cloud Platform Blog*, May 18, 2016

[32] Mellanox Technologies. Programmable ConnectX-3 Pro Adapter Card. `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Programmable_ConnectX-3_Pro_Card_EN.pdf`

[33] Mellanox Technologies. Connect-IB: Architecture for Scalable High Performance Computing. `http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf`

[34] Transaction Processing Performance Council (TPC). TPC BENCHMARK C: Standard Specification. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`.

[35] Viking Technology. `http://www.vikingtechnology.com`.