# RStore: A Direct-Access DRAM-based Data Store

Animesh Trivedi, Patrick Stuedi, Bernard Metzler,
Clemens Lutz and Martin Schmatz
*IBM Research - Zurich*

Thomas R. Gross
*Computer Science Department*
*ETH Zurich*

***Abstract*—Distributed DRAM stores have become an attractive option for providing fast data accesses to analytics applications. To accelerate the performance of these stores, researchers have proposed using RDMA technology. RDMA offers high bandwidth and low latency data access by carefully separating resource setup from IO operations, and making IO operations fast by using rich network semantics and offloading. Despite recent interest, leveraging the full potential of RDMA in a distributed environment remains a challenging task. In this paper, we present RDMA Store or RStore, a DRAM-based data store that delivers high performance by extending RDMA's separation philosophy to a distributed setting. RStore achieves high aggregate bandwidth (705 Gb/s) and close-to-hardware latency on our 12-machine testbed. We developed a distributed graph processing framework and a Key-Value sorter using RStore's unique memory-like API. The graph processing framework, which relies on RStore for low-latency graph access, outperforms state-of-the-art systems by margins of $2.6-4.2\times$ when calculating PageRank. The Key-Value sorter can sort 256 GB of data in 31.7 sec, which is $8\times$ better than Hadoop TeraSort in a similar setting.**

***Keywords*-Next generation networking; Data storage systems; Data processing;**

## I. Introduction

The demands on big data analytics platforms to provide real-time performance has led to memory-centric architectures where more and more data is kept in DRAM for fast access. In this context, several in-memory storage systems such as RAMCloud [1], FaRM [2], key-value stores [3], [4] etc., have been proposed. Consequently, due to the elimination of slow disks, the network has become the new performance bottleneck.

To improve the network performance of in-memory storage systems, researchers have advocated using Remote Direct Memory Access (RDMA) technology [1]–[5]. RDMA networks like Infiniband or iWARP provide high-throughput/low-latency with very low CPU overhead. These performance advantages are mainly achieved by separating the *control path* (or setup) from the *data path* (or access), and making the data path fast by using one-sided network IO semantics and offloading network processing. By deploying RDMA in a limited capacity, one still gets marginal benefits from the offloaded protocol processing and high link rates of specialized interconnects (such as 56Gb/s on Infiniband). However, the full potential of RDMA is closely tied to its separation philosophy that sets up resources a priori to eliminate overheads from memory management, multiplexing and data copies etc., during fast data accesses [6]. Extending this philosophy to a distributed environment mandates a careful network and storage resource management, whose cost, without thoughtful consideration, can easily eclipse any potential performance gains from using RDMA-capable networks [7]. Despite previous efforts, leveraging the full performance of RDMA in a distributed environment still remains a challenging task.

In this paper, we present RStore, a DRAM-based data store that extends RDMA's separation philosophy to a distributed environment to deliver high-performance data accesses. It achieves the separation at the API level by having distinct calls for allocation and access. This setup lets applications pre-allocate and pre-fetch expensive RDMA resources in a distributed environment before the data processing phase begins. RStore is built on two design principles. First, *decouple resource allocation from its abstraction binding*. This decoupling allows RStore to manipulate and reuse expensive resources (e.g. RDMA buffer allocation and registration) independently from the provided storage abstraction. Second, *keep the IO path thin and fast*. A fast and thin IO path helps to deliver very fast data access to applications with their own synchronization logic. RStore further exploits availability of multiple NICs by striping data across servers. As a result, RStore delivers high performance that is very close to the RDMA-network limits.

The basic storage abstraction in RStore is a flat 64-bit distributed access *namespace*. Applications interact with a namespace using RStore's API that resembles the familiar memory-mapped IO (`mmap` and friends). Using the API, applications can reserve, allocate, and map storage capacity in any namespace. To illustrate the power of the abstraction, we have developed two different applications on top of RStore's API. Our first application is a distributed in-memory graph processing framework called Carafe, which imports, processes, and stores graph data, metadata, and messages in RStore. In our experiments, Carafe outperforms state-of-the-art systems by margins of $2.6-4.2\times$ when calculating PageRank. Our second application is a distributed Key-Value sorter called RSort. It stores data in RStore and leverages high-bandwidth data access to deliver high sorting performance. RSort can sort 256 GB of data in 31.7 sec,

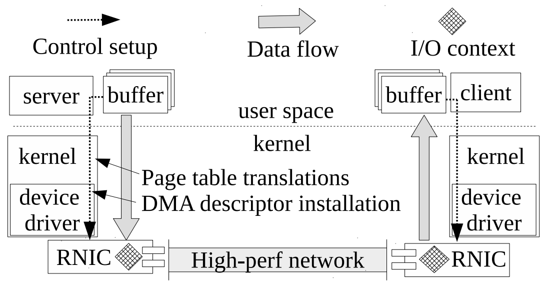


Figure 1: RDMA communication model.

which is 8× better than Hadoop TeraSort in a similar setting.

Our contributions include (a) design and implementation of a DRAM-based data store where RDMA philosophy is integrated as a first-class citizen; (b) illustration of the system capability by developing two different types of applications using RStore's API that allows pre-fetching and pre-allocation of resources out of the performance critical path in a distributed setting; (c) quantification of the distributed control setup cost of RDMA at scale (1000s of connections, TBs of DRAM).

## II. RDMA Communication Model

According to the separation philosophy of RDMA, network resources are allocated and registered with RDMA-capable network controllers (RNICs) on a *control path*. The kernel is involved on this path checking permissions, translating page-table mappings, installing DMA descriptors, and setting up IO context in the RNIC. After resource allocation, applications interacts directly with the RNIC, which provides direct, safe userspace access to offloaded network resources. On a fast *data path*, applications post IO requests on userspace mapped TX/RX queues and receive IO completion notifications directly on a completion queue (CQ) from the RNIC. Data flows between userspace buffers and the network in a zero-copy mode (see Figure 1). There are two types of communication models: (a) send and receive and (b) RDMA operations. Send/receive operations, involve both peers in the communication. The sender sends the data, whereas the receiver pre-posts a receive buffer on the RX queue indicating where it wants to receive data. We use send/receive operations for implementing a high-performance RPC [8]. RDMA operations comprise read, write, and atomics, commonly referred to as one-sided operations. These semantically rich operations only require one peer to actively read, write, or atomically manipulate remote memory buffers. The remote peer is only involved in the preparation of memory buffers for RDMA accesses. Because of its design and minimum peer involvement, one-sided operations have the lowest latency with highest bandwidth. We leverage RDMA one-sided operations for data transfers. The philosophy of path separation, together with rich IO semantics and hardware offloading, is what gives RDMA its performance benefits.

### A. Challenges in RDMA Deployment

Leveraging the full performance of RDMA in a distributed environment is a challenging task for multiple reasons. The first and foremost question is how to extend the separation philosophy of the RDMA networks to a distributed data store. Traditional APIs (such as Key-Value or Files) have narrow interfaces that do not allow resource setup requirements to be captured prior to an access. In the absence of sufficient information, a system cannot leverage RDMA operations to deliver the highest performing IO stack.

Second, in a distributed data store that might offer separation, how can a data-processing application leverage it? What is the right abstraction? The right IO abstraction can help applications, which can identify, create, and prefetch necessary distributed objects upfront, to gain significant performance gains when performance really matters.

Lastly, distributed RDMA resource management is a complex task. It includes "when and how" to open RDMA connections to servers, share offloaded resources, register buffers on multiple RNICs etc., in a distributed setting. If not managed carefully, the high cost of resource setup has been shown to dominate any potential performance gains from using the high-performance RDMA networks [7]. In comparison, previous RDMA integration efforts have only dealt in part with these challenges. For example, RAMCloud [1] delivers low-latency IO for small objects but does not let applications manage their resources. On the other hand, FaRM [2] gives the possibility to pre-allocate and manage transactional objects but does not deliver lowest possible latency because of object layout adjustments in the IO path.

## III. RStore

RStore is a distributed in-memory data store that delivers high performance by extending RDMA's separation philosophy to a distributed environment. In this section we describe its goals, design principles, and implementation details.

### A. Scope and Goals

We consider a rack-scale deployment of RStore, where data is imported from a persistent storage, processed by thousands of cores while the working set is held in distributed DRAMs, and finally results are written back to the storage. The high-level goals of the system are:

- *Efficient distributed setup path:* Even though the RDMA setup cost is a part of the separated control path, it should not be prohibitively high. This cost is one of the primary concerns with RDMA deployments in distributed settings.
- *Light-weight abstraction:* The proposed abstraction should be intuitive, light-weight and general-purpose. This property is key to building applications with different consistency and performance requirements.

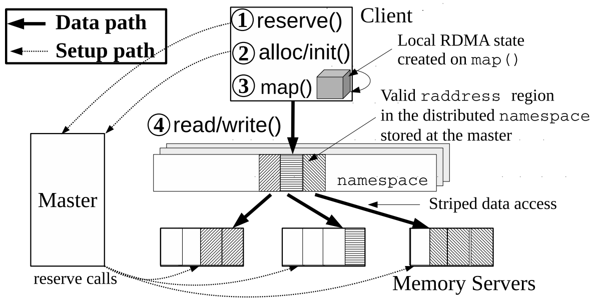


Figure 2: Design and components in RStore.

- *High performance:* The system should be able to deliver high bandwidth and low latency to a single client by leveraging all networking hardware.

However, in our pursuit of achieving these goals, we sacrifice a few system properties. RStore's fault-tolerance handling is primitive. It does not strive for data durability in the case of arbitrary memory server failures. Memory server replication or fast recovery [9] can be used to deal with this issue, but we have not yet done so. We are implementing a copy-on-write mode (see section III-G) to provide a strong atomicity guarantee under concurrency and failures. We first start here by discussing the abstraction and design principles, and how they help us to achieve RStore's goals.

## B. Components

RStore has three main components: a master, a set of memory servers, and a client-side user library. The master is a logically centralized entity that acts as the system arbiter and stores all metadata in DRAM for fast servicing. The metadata consists of created namespaces, allocated memory regions and their locations, permissions, active client states etc. Data is distributed and stored in DRAMs of participating memory servers. Their primary responsibility is to allocate and prepare DRAM buffers to be accessed by RStore applications. Applications interact with RStore using a client-side API. Applications and memory servers communicate with the master using a fast and scalable RPC implementation [8]. Figure 2 captures the interaction among the components.

## C. Abstraction and Principles

The basic access abstraction in RStore is a flat, 64-bit, byte-addressable storage address space called *namespace*. RStore's applications create, join, and allocate storage capacity in multiple namespaces. An allocated memory segment is identified by a globally visible <address, length> tuple, called `raddress`. Applications use raddress segments to store, and share distributed data structures, tables, data-blobs etc. RStore is built on the following two design principles:

1) *Decouple resource allocation from its abstraction binding:* The distributed control path in RStore consists of setting up two types of resources (a) systems IO resources; (b) metadata associated with RStore. The first involves allocating memory, registering buffers with RNICs, opening up RDMA connections etc. The latter involves binding these resources to application-visible RStore namespaces and addresses. We made RStore's distributed setup path efficient by decoupling these two resources. Applications can manage costly system resource allocation separately from the performance critical path, which may encompass relatively cheaper metadata manipulations in RStore. For example, by reserving TBs of DRAM memory independently from binding them to a `raddress`, applications can quickly allocate and free `raddress` segments without having to allocate and pin memory buffers repeatedly.
2) *Keep the IO path thin and fast:* RStore does not interpret or impose any structure on the stored data. Consequently, it avoids the overhead of the implicit synchronization associated with data structure accesses [10]. Distributed applications with explicit coordination logic among workers do not require additional synchronization overheads from the storage. While maintaining the same considerations as RDMA, RStore translates IO requests to RDMA operations and avoids context switches, memory allocations, data touch operations (e.g. copies or layout adjustments) etc., during a data access. It delivers large aggregate bandwidth by striping data across multiple RNICs without overwhelming a single server or link.

## D. RStore API and the Path Separation

RStore achieves the separation using discrete API calls to set up and access resources in a distributed environment. There are three explicit calls, namely, `reserve()`, `alloc()`, and `map()` to manage resources within a namespace. Following the decoupling principle, `reserve()` allocates DRAM buffers for RDMA accesses at the memory servers, `alloc()` binds them to a `raddress` at the master, and `map()` associates the `raddress` to local memory at the clients. An allocated `raddress` region is physically served by DRAMs from multiple memory servers. Multiple applications can concurrently map and access the same `raddress` region. These three calls collectively constitute the distributed control setup (or resource setup) in RStore. After the control setup, fast data-path calls, i.e., `read()` and `write()` in mapped regions, do not involve any resource allocations. Table I gives an overview of RStore's API and lists its main objects and associated calls.

## E. Distributed Memory Management

RStore manages distributed DRAMs internally in a granularity of *chunks*. Applications, however, can allocate and map `raddress` regions of any size. A list of free chunks, allocated chunks to `raddress` regions, their access permissions, mapping types, reference counts, and active clients states etc., are maintained as system metadata at the master.

| Object | RStore API calls | Description |
|---|---|---|
| Master | `connect(struct sockaddr*)` | Connects to the RStore master |
| | `create_ns(string name)` | Creates a namespace with a string identifier, returns Namespace *obj |
| | `join_ns(string name)` | Joins an already created namespace, returns Namespace *obj |
| | `delete_ns(string name)` | Destroys a passed namespace |
| Namespace | `reserve(u64 size)` | Allocates and prepares DRAM on memory servers for RDMA access |
| | `release(u64 size)` | Releases memory reserve from a namespace |
| | `alloc(u64 size, int flags)` | Allocates a globally visible address segment, returns `raddress` *obj |
| | `free(raddress*)` | Deletes a `raddress` allocation segment at the master |
| RAddress | `init(namespace*, u64 addr, u64 size)` | Initializes a `raddress` object to a valid global address and length |
| | `map(int flags)` | Allocates local network/memory resources, returns a local void* ptr |
| | `unmap()` | Frees all resources associated with this `raddress` mapping |
| | `read(void *p, u64 size)` | Reads a mapped memory region for a passed length |
| | `write(void *p, u64 size)` | Writes a mapped memory region for a passed length |

Table I: RStore client API. The `reserve()`, `alloc()` and `map()` calls constitute the control path in the distributed setting. The `read()` and `write()` calls are fast data-path calls, in which no global or local resource allocations happen.

Storing and accessing data in RStore are multi-step processes. First, an application must reserve sufficient DRAM capacity by calling `reserve()`. If the free chunk list at the master has sufficient capacity, the RPC returns immediately; otherwise the master chooses a set of memory servers to reserve the memory capacity requested. The current implementation uses a primitive round-robin policy to distribute the load uniformly. Upon receiving a `reserve()` RPC call from the master, the memory servers allocate and register chunks of DRAM to an RDMA device and communicate RDMA credentials (STag, virtual address and length) back to the master. These chunks are added to the free chunk list.

Second, an `alloc()` call stitches together reserved memory chunks from different memory servers (for best parallelism) under a single distributed `raddress` region. This globally visible binding between a `raddress` region and the memory chunks is created and stored at the master. The newly created `raddress` region together with the location of its chunks are returned to the client as the result of the `alloc()` RPC. For a previously allocated region, a client can initialize a `raddress` object by calling `init()` with a valid <address, length> range. The call fetches the chunk locations from the master. The master maintains appropriate reference counting on objects to avoid freeing them while they are still mapped and in use at clients. When a client's connection aborts, the master cleans up the associate states and references. These metadata manipulations are atomic as they are serialized by the master using appropriate locks.

Lastly, a valid `raddress` object requires a local mapping before the IO operations. Similarly to the POSIX `mmap()` call, a `map()` call returns a local DRAM address, which is made RDMA-ready by the RStore's client library. A client uses this address for staging and modifying data in RStore. When a valid `raddress` object is destroyed, it notifies the master for reference cleanup.

## F. IO Operations and Synchronization

The fast data path in RStore consists of `read()` and `write()` calls. These calls do not involve any data touch operations or resource allocations anywhere in the system, which is the key to deliver high performance to applications. A `read/write()` call is divided at the chunk boundary and individual chunk IO requests are then translated into one-sided RDMA operations for zero-copy network transfers. The byte-granular nature of IO operations fits perfectly with the message-oriented nature of RDMA operations: an RNIC understands message boundaries and only notifies an application when the complete message has been sent or received. Hence, the transfer time depends on the IO size, rather than on the mapped memory size.

With its key focus on separation and performance, RStore does not provide any form of global IO synchronization. RStore's clients must coordinate among themselves to define the concurrency access model. We argue that this is not an unusual feature as many distributed applications tend to have their own synchronization mechanism using external services, e.g. Zookeeper. We illustrate in section V-A how one can achieve global barrier coordination using RStore.

## G. Copy-on-write for Machine Failures

The zero-copy architecture of RStore modifies data in-place. Hence, a failure of a writer client leaves data in an inconsistent state. To provide atomic updates in case of a memory server failure, we are implementing a copy-on-write (COW) type `raddress` segment (indicated by a flag in the `alloc()` call). In the COW-mode, a small amount of per-chunk metadata (8 bytes) is maintained at the memory servers to communicate the chunk state to clients. This metadata is accessed during IO operations using RNIC's scatter-gather capability.

While mapping a COW region, a writer client synchronizes with the master to get a time-bounded (default: 10 sec) lease for new memory locations in a multiple of chunks for every write operation. The master enforces a policy of one writer with concurrent readers by rejecting giving new leases for an already mapped COW region, to another writer. After local modifications, the data is written out in multiples of chunks to the new chunk locations. Upon a successful `write()`, the writer notifies the master to update the `raddress` metadata to point to the newly written chunk locations. If any of these steps fail, the master retains the last known chunk locations for the `raddress` region. When a lease expires (owing to a client's failure or inactivity), the master garbage collects the lease's chunk locations. After a successful write notification from the client, the master marks the per-chunk metadata at the old locations as tainted.

A concurrent reader in a COW-mapped region can be in one of three states. First, it reads clean data while a write is in progress. As the new data is written out of place, this read returns the last known consistent value for the data. Second, the reader sees a subset of chunk locations as tainted while the master is updating the per-chunk metadata. Third and last, the reader sees all per-chunk metadata as tainted. Note that in the last two cases, the presence of tainted metadata only notifies the reader about the availability of new data, but the old data locations are not over-written or invalidated as the new data is written out of place in new locations. If it chooses so, a reader can still work with a consistent copy of the data stored at the old locations. Alternatively, the reader, upon detecting the tainted metadata, can proceed to obtain the new data chunk locations from the master. When an old chunk location has no active client mappings, the master uses it for new allocation requests.

### *H. Discussion on RStore API and Abstraction*

RStore's unique memory-like API has some critical advantages that differentiates it from state-of-the-art systems:

- **Explicit Distributed Resource Management:** RStore's API gives applications control over RDMA and memory setup. Using this explicit control, applications can allocate, prefetch, and prepare stateful objects associated with TBs of DRAM by calling `init` (or `alloc` for a fresh allocation). These calls fetch chunk locations from the master, and can be called separately from the `map` call, which involves local memory commitment. Furthermore, applications can use this hierarchical setup of control calls (`reserve, alloc, map`) to progressively distribute the resource allocation cost in a most efficient way as suited to their workload requirements.
- **Unified Network and Application Buffers:** By using an explicit `map` call, RStore integrates network and application buffers. As explained, the `map` call returns a local `void*` pointer to a memory buffer, which is made RDMA-ready by the client-side library. This buffer is known to the network for IO and to the application for data access, thus eliminating a data copy which is typically done to move data between network and application buffers. Hence, RStore's IO stack is a true zero-copy stack on both, TX and RX sides. Furthermore, due to the data copy elimination, one-sided RDMA operations in RStore deliver data directly into the application buffers with highest bandwidth and lowest latency.
- **Expressive Memory API:** The raw byte-addressable memory abstraction of RStore is the most expressive, general-purpose storage abstraction. This memory abstraction enables the building of distributed linked data structures with pointer arithmetic for offset calculations. The `map` call, which supports mapping partial address ranges, allows large memory objects to be partially mapped and updated. Both of our applications use this facility to build and access distributed data structures. This would not have been possible on an object or Key-Value store that only permits full object updates with their `get/set` API calls. For example, it is not possible to update a pointer at a particular offset in a value of a key in a Key-Value store.

| | |
|---|---|
| CPU | Dual Xeon E5-2690, 2.9 GHz cores |
| DRAM | 256 GB, DDR3 1600MHz |
| NIC | 3 Dual port Chelsio T4 iWARP RNICs |
| Network BW | 60 Gb/s ($3 \times 2 \times 10$ Gb/s) |
| Network Latency | 9.6 $\mu$sec, 8B RDMA read latency |

Table II: 12-machine testbed configuration.

## IV. Performance Evaluation

RStore is implemented in Linux (for `3.13.11` kernel) in about 15K LOC of C++ which contains code for the master (4K), memory servers (1K), the application-side library (3.5K) and common subsystems including RDMA (6.5K). RStore also follows the best practices of RDMA resource caching and sharing as recommended in the literature [2], [8]. The performance evaluation is done on our 12-machine iWARP/Ethernet testbed as shown in Table II. For all benchmarks, clients and memory servers (72 servers in total, 6 per machine) are co-located on every machine. The master runs on a separate, dedicated machine. The chunk size is set to 64 MB. The key performance evaluation highlights are:

- **Efficient distributed control setup path:** RStore reserves and allocates 1.15 TB of DRAM in 22.1 sec (52.1 GB/sec). Clients need 19.9 sec to map this memory in their local DRAM.
- **High performance by maintaining the separation philosophy in a distributed setting:** RStore delivers bare-metal latency (12 $\mu$sec, in comparison the iWARP network latency is 9.6 $\mu$sec) and high aggregate cluster bandwidth (705 Gb/s for 12 servers, 58.7 Gb/s per server).
- **High application performance using RStore's API:** Carafe, our distributed in-memory graph processing framework, is $2.6 - 4.2\times$ faster than state-of-the-art

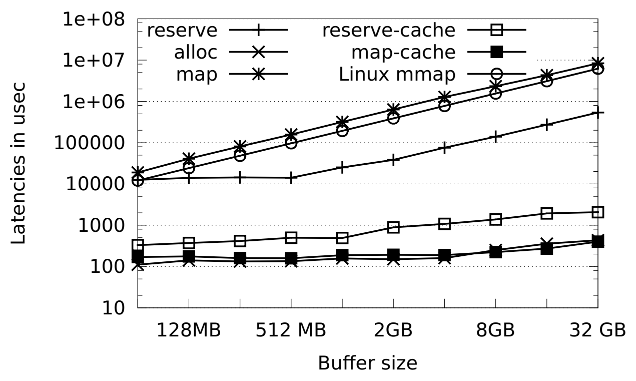


Figure 3: Control setup cost for a single client.

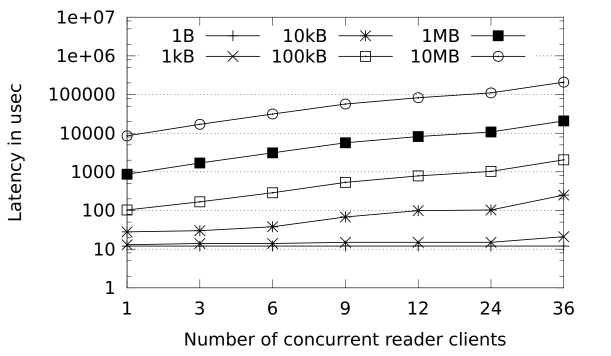


Figure 5: Read latencies wrt. concurrent readers.

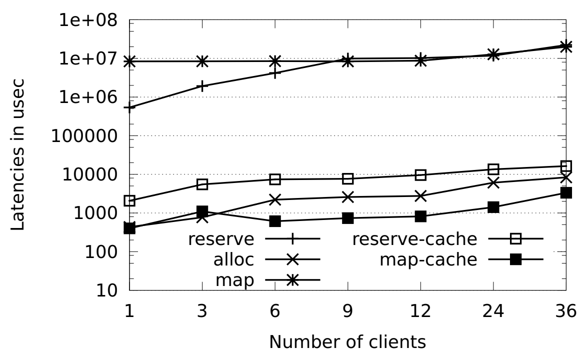


Figure 4: Scaling of the control path setup cost.

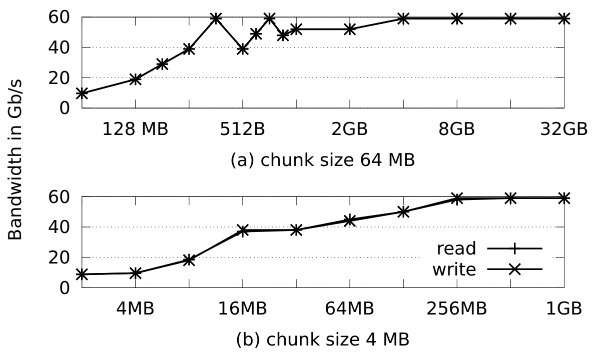


Figure 6: Single-client IO bandwidth.

systems in calculating PageRank on a graph containing millions of vertices. RStore sort (RSort) sorts 256 GB of data in 31.7 sec, which is 8× better than Hadoop TeraSort in a similar setting.

## A. Cost of the Distributed Control Path

In this section, we quantify the cost of the distributed control setup path, which consists of `reserve()`, `alloc()` and `map()`. Figure 3 shows the performance for a single client. The y-axis shows the latency of the operations in comparison to the buffer size (on the x-axis). The `reserve()` and `map()` calls are the most expensive calls, as they involve costly memory allocation operations (see Linux mmap cost in Figure 3). The `reserve()` call benefits from spreading the memory allocation across multiple memory servers.

Figure 4 shows the scaling of the control cost in comparison to the number of clients. A variable number of clients (shown on the x-axis) concurrently start to reserve, allocate, and map 32 GB memory regions in their own namespaces. As shown in Figure 4, the setup cost scales gracefully, and 36 clients can prepare 1.15 TB of DRAM in approximately 22.1 sec. Mapping this memory takes 19.9 sec (18.7 sec are from concurrent `mmaps`), and involves opening, in total, 2,592 (72 servers × 36 clients) RDMA connections between machines.

## B. Efficiency of the Resource Caching

Memory allocation is the primary contributor to the overall control setup cost. It is heavily influenced by the number of co-located clients and memory servers, the size of the allocation request, the number of free hugepages, memory fragmentation etc. However, a part of the memory allocation cost is only incurred upon a cold start. For example, after the first `reserve()` call to a memory server, which involves allocating new memory segments via mmap, the RDMA credentials are used repeatedly between subsequent `release()` and `reserve()` (also between `free()` and `alloc()`) calls. The master caches and maintains a configurable number of free chunks. The memory caching is most effective once a workload has hit its peak working set size and reserved sufficient DRAM. Similarly, the caching also helps on the client side with repeated `map()` and `unmap()` calls. Figure 3 and Figure 4 also show the cached performance with `reserve-cache` and `map-cache` lines. In the cached mode, RStore can reserve and map 1.15 TB of DRAM in 16.3 msec and 3.3 msec respectively — a three orders of magnitude improvement

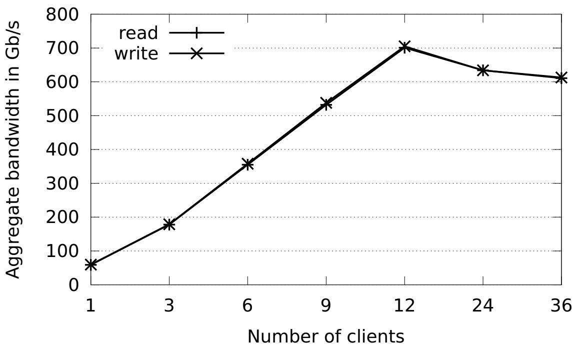


Figure 7: Aggregate cluster IO bandwidth of RStore.

over the cold-start performance.

RDMA connection caching at the client-side is very effective in hiding the high RDMA connection setup cost (about 2.5 msec/connection). However, as the number of servers increases so does the cost of maintaining the connection cache (with tied-up associated offloaded network resources). A typical RNIC can support upto 64K offloaded connections. In our setup we have not yet reached these limits.

## C. Performance of Data-Path Operations

**Latency:** As RStore's data calls map directly to RDMA operations, their performances are very close to the network limits. On our iWARP cluster (with network round trip latency of 9.6 $\mu$sec), it takes 12 $\mu$sec to read 8 bytes of remote data (not shown). We expect this performance to improve further with ultra-low latency interconnects such as Infiniband. Figure 5 shows the effect on the read latency (in $\mu$secs on the y-axis) as we increase the number of concurrent reader clients (on the x-axis). For small requests (less than a kB), RStore is capable of delivering constant IO latencies. As the size increases, the request becomes bandwidth bounded and the latency increases.

**Bandwidth:** The chunk size determines the level of network parallelism that clients can achieve in RStore. For large buffer sizes, a client gets the peak bandwidth of 59.1 Gb/s, less than 2% below the theoretical maximum of 60 Gb/s. Figure 6(a) shows the performance of a single client with respect to the various buffer sizes. The sawtooth shape of the bandwidth curve can be explained as follows. Between 64 MB and 384 MB (which is $6 \times 64$ MB), the bandwidth increases linearly in multiples of a single NIC capacity (10 Gb/s) and hits the peak at 384 MB. Then the bandwidth falls because larger buffer sizes create an uneven distribution of chunks on our 6 NICs/machine setup. Hence, whenever the total transfer size is a 6-multiple of 64MB, the client gets the full bandwidth. To confirm this hypothesis, we have also repeated the experiment with 4 MB chunk size (see Figure 6(b)). We observe a similar pattern with the client getting the full bandwidth above a 256 MB buffer size. For sufficiently large sizes, the transfer time dictates the overall completion time, and this unevenness no longer matters.

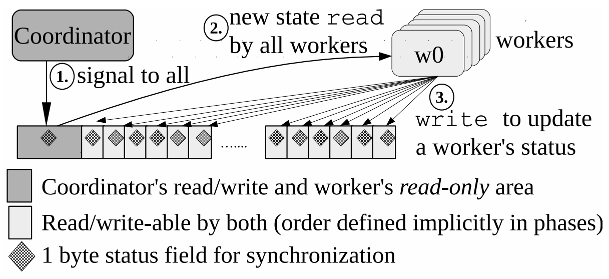

Coordinator
1. signal to all
2. new state read by all workers
w0
workers
3. write to update a worker's status
.......
Coordinator's read/write and worker's read-only area
Read/write-able by both (order defined implicitly in phases)
1 byte status field for synchronization
Figure 8: Distributed barrier coordination.

Figure 7 shows the aggregate system bandwidth as the number of clients increases. The performance scales linearly and peaks at 705 Gb/s (theoretical maximum: 720 Gb/s) until we host multiple clients per machine. We are investigating the QoS management of RDMA traffic. Our initial analysis suggests that one reason for the drop is the co-location of memory servers and clients on the same physical machine in our limited 12-machine testbed.

## V. Applications

We have developed two different applications, a distributed graph processing framework called Carafe, and a distributed Key-Value sorter called RSort. As both of these applications require coordination among workers, we have also built a simple coordination mechanism by leveraging RStore's byte-addressability property to atomically access one-byte flags in a distributed metadata structure.

## A. Global Barrier Synchronization

The coordination mechanism consists of a central coordinator and multiple workers. These entities communicate through a distributed shared data structure, laid out in RStore by the coordinator. This data region is mapped in the memory of every worker. The region contains a one byte status field together with application specific data. Byte accesses in RStore are atomic as one byte is the smallest unit of data access. The layout of the data region is shown in Figure 8. It contains a coordinator-specific global field and a per-worker field. The per-worker field is written by both the coordinator and a client. At the start, the coordinator writes application-specific data in the per-worker area (depends upon the application logic) and marks their status fields as valid. It then proceeds to signal all workers by writing its status byte field as "start". Upon reading "start" in the coordinator field, workers then read their area to get an assigned workload, execute it, and mark their status byte as "finished" or "error". The coordinator, upon reading that every worker has finished its assigned work, can decide to start the next phase of work. Our simple coordination mechanism avoids write-write synchronization

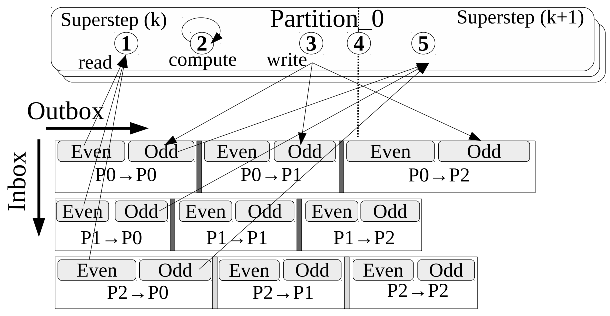


Figure 9: Message delivery and Pregel execution in Carafe. Assuming k is even (1) collect incoming messages from the even containers; (2) invoke `compute` on vertices with messages; (3) write messages to odd containers; (4) synchronize the superstep; (5) collect messages from even containers for (k+1)$^{th}$ superstep.

by taking turns to read or write byte status fields atomically. Carafe uses this mechanism for superstep synchronization and message delivery among workers. RSort uses it for work assignment and moving from phase one (classification) to phase two (sorting).

## B. Carafe: Distributed Graph Processing

Carafe, our distributed in-memory graph processing framework, stores, accesses, and manipulates the graph structure inside RStore. The key performance requirement for Carafe is low-latency access to the graph structure, metadata, and messages. We have implemented an online [11] and a Pregel-like [12] graph processing engine.

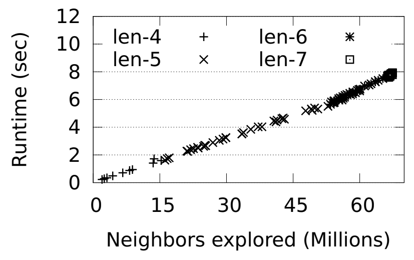
**Graph Storage Format:** Carafe imports edges and vertices in separate RStore namespaces. The edges are stored in an adjacency list format. Edges are directional, and bi-directional edges are split and stored twice. Vertices are stored as a single contiguous array. A single vertex structure contains its internal id (array index), externally associated id given by the graph file, offset of its adjacency list in the edge namespace, and size of the list etc. It also contains a graph property map, which contains associated contextual information and run-time data from algorithms. The graph is represented by a `CarafeGraph` class that contains high-level pertinent information about the graph and the associated RStore metadata about vertex and edge namespaces. At this moment, Carafe does not support graph mutability.

**Online Graph Exploration:** For online graph scans Carafe provides a `VertexHandle` and an `EdgeIterator`. A `VertexHandle` is initiated by passing a vertex id to the `CarafeGraph`. This action involves calculating the right offset into the vertex namespace, mapping and reading that vertex from RStore, and then initializing an `EdgeIterator` to its edge list in the edge namespace. An `EdgeIterator` provides a mechanism to iterate through neighboring vertex ids. Applications can use a neighboring vertex id to initialize a new `VertexHandle`. Once done, the application must release associated RStore resources by calling unmap on these graph objects. We have implemented the Dijkstra shortest path algorithm using this facility.

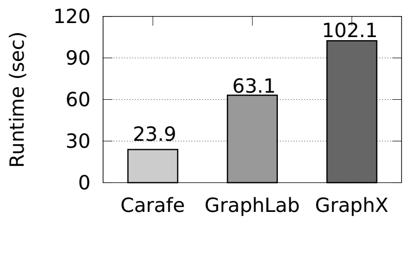
**Graph Partitioning and Pregel Model:** Carafe does weighted vertex partitioning to divide vertices into equal weight partitions. Our initial attempt to partition the graph just using edges resulted in uneven computation. With this partitioning scheme, some partitions ended up containing too many sparsely connected vertices whose access time dominated the computation. In the weighted vertex partitioning, Carafe assigns weight to a vertex (represents access overhead) as well as an edge (represents access and messaging overheads). The graph is then divided into equal weight partitions containing vertices. These partitions are then assigned to worker machines. Worker machines access vertices using a specialized `SequentialEdgeIterator` that loads and reads graph data on a partition size granularity. Each partition contains a partition worker thread to execute the Pregel logic that invokes an equivalent `compute` function on vertices with messages from neighbors. After each superstep, new vertex data is written back to RStore and every partition worker coordinates with the Pregel master for the next superstep. We have implemented PageRank using the Pregel model.

**Message Passing:** Message passing represents a major overhead in the Pregel computation model. RStore's efficient network IO enabled us to implement a shared mailbox schema to send and receive messages in Carafe. In the shared mailbox model, each partition worker reads and writes messages into RStore namespaces. Messages in Carafe are delivered individually. The Carafe mailbox is divided into as equal a number of rows and columns as there are workers. These `segments` ($i^{th}$ row and $j^{th}$ column) contain messages sent from individual partitions (from Partition$_i$ to Partition$_j$). Individual segments are further divided into two areas called `even` and `odd` containers. The size of each container is determined by the number of edges between partitions. For the $i^{th}$ partition, $i^{th}$ row and $i^{th}$ column represent its Outbox and Inbox, respectively. During the $k^{th}$ (assuming $k$ is even) superstep, workers read from `even` Inbox containers and write new messages to `odd` Outbox containers. The `even` and `odd` containers switch their roles in the next supersteps. This setup achieves read-write coordination at the expense of more storage. Figure 9 shows an example of the mailbox setup for 3 partition workers.

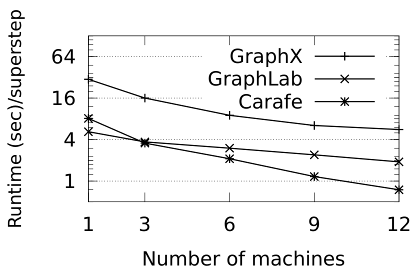
**Evaluation:** We evaluate performance of Carafe on the LiveJournal social network graph from the Stanford Network Analysis Project [13]. The graph contains 4.8 million vertices and 68.9 million edges. Our first benchmark calculates the shortest path between 100 randomly chosen vertex pairs using the Dijkstra shortest path algorithm. Figure 10a shows the runtime of the algorithm in comparison to the number



(a) Dijkstra shortest path



(b) PageRank runtime for 32 supersteps



(c) PageRank scaling

Figure 10: Performance of Carafe on the LiveJournal social network graph [13].

of neighbors explored and the discovered path lengths. As shown, the cost of neighborhood exploration increases linearly with the number of vertices accessed. For example, Carafe explored 67 million vertices in 7.9 seconds for a path length of 7.

Our second benchmark consists of executing the PageRank algorithm on the LiveJournal graph in Carafe's Pregel engine. We compare the performance of Carafe against GraphLab [14] and GraphX [15], which are two of the fastest distributed graph processing systems in the literature. Figure 10b shows the runtime of PageRank for the first 32 supersteps in all three systems. Carafe outperforms GraphLab and GraphX by a margin of $2.6\times$ and $4.2\times$, respectively. Carafe also scales linearly on our 12-machine cluster as shown in Figure 10c.

### C. RSort: Distributed Sorting on RStore

RSort, is a distributed record sorter for key-value (KV) records from http://sortbenchmark.org/ in the Indy mode. As RSort does not perform any locality aware optimizations, its performance is governed by the data access bandwidth. RSort implements a two phase bucket sorter. In the first phase, input data is classified into range buckets, and in the second phase, individual buckets are sorted locally.

**Phase One (Classification):** In phase one all workers first read and then classify the input data into multiple *bucket* namespaces. A bucket, private to a worker, represents a specific key range for which its namespace should contain records. Having private bucket namespaces eliminates the need for write-write synchronization when multiple workers try to append a record in a same key range bucket namespace. The names of bucket namespaces follow a syntax of $worker\_i\_bucket\_j$, which uniquely identifies the $j^{th}$ bucket namespace for the $i^{th}$ worker. The key range for the $j^{th}$ bucket for every worker is defined globally, and each worker has the same number of buckets. Consequently, the total number of buckets in RStore is $buckets\_per\_worker \times total\_workers$.

**Phase Two (Assembly and Sorting):** In phase two, a key range is exclusively assigned to a particular worker by the coordinator. Recall that a bucket with name $worker\_i\_bucket\_j$ contains all records that a worker $i$ has seen for the key range assigned to bucket $j$. Upon getting the assigned key range, which might belong to bucket index $j$, a worker joins all bucket namespaces of the name syntax

$$worker\_i\_bucket\_j, where\ i \in [0, total\_workers - 1].$$

The worker then reads and assembles all records in a final output namespace. The final namespace has name syntax $final\_j$, where $j$ represents the bucket and associated key range. After this step, each worker has all records belonging to a particular key range copied in the final namespace. A worker then performs a local sorting operation (GNU parallel quick sort) on this assembled record data. After sorting, the data is written out to the final namespace and the worker's status is updated. When all workers have finished writing data in the final namespaces, the sorting is over.

**Evaluation:** The sorting time reported is the time between the signalling of phase one until phase two is signaled completed by all workers. The numbers reported are the average of three runs.

We evaluate weak-scaling performance of RSort by fixing the per-worker input size to 40 million records of 100 bytes each. The total data size increases with the number of workers. For 64 workers, the total data size is 256 GB. Figure 11 shows the scaling performance of RSort. For the comparison, we also show the performance of GNU's parallel sorting implementation marked as `_gnu_parallel::sort()`. The local sorting does not have any network IO or resource allocation overheads. On a single server, 64 GB is the largest data size that we can sort in memory. As shown in the figure, RSort exhibits very good scaling performance. It scales linearly between 1-8 workers, until we host multiple workers per machine. For data sets larger than 8 GB, RSort is $1.9 - 5.4\times$ faster than the parallel local sorting.

**RSort vs. Hadoop TeraSort:** We now compare the performance of RSort with that of Hadoop TeraSort (version 2.2) under similar circumstances running on RAMDisks. We configured YARN to give maximum cluster resources (cores and DRAM) to Hadoop. Table III shows the configuration of

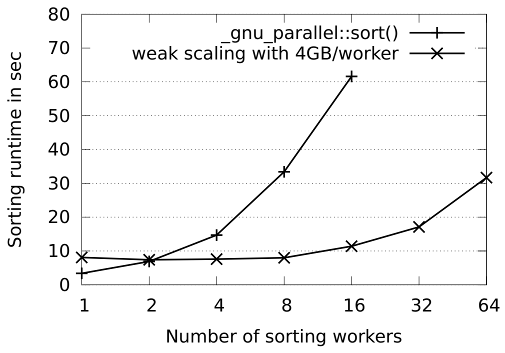


Figure 11: Weak scaling performance of RSort.

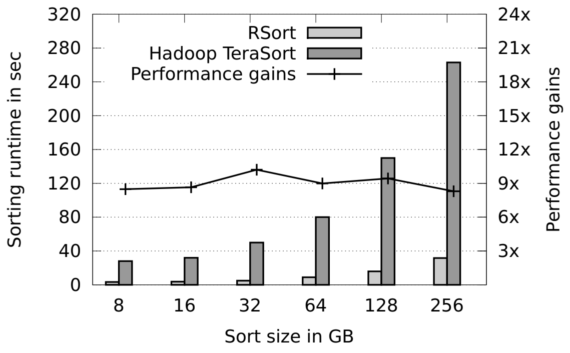


Figure 12: Performance of RSort and Hadoop TeraSort.

| Property | Comments/Value |
|---|---|
| dfs.block.size, dfs.replication | 64 MB, 1 |
| io.file.buffer.size | 1 MB |
| hadoop.temp.dir | on RAMDisk, 128GB |
| mapreduce.tasktracker. map/reduce.tasks.maximum | 32 (no of CPU cores) |
| mapreduce.job.maps/reduces | 256 |
| mapreduce.input. fileinputformat.split.minsize | 512 MB |
| mapreduce.reduce.shuffle. parallelcopies | 32 |
| yarn.nodemanager.resource. memory-mb | 128 GB |
| yarn.scheduler. minimum/maximum-allocation-mb | 8 GB, 128 GB |

Table III: Hadoop TeraSort configuration. The values are configured to give maximum cluster resources to Hadoop.

our Hadoop cluster. Figure 12 compares their performances for variable-size data sorting. As shown, RSort consistently outperforms Hadoop TeraSort by margins of $8 - 10\times$.

## VI. Experiences with RStore

*What made RStore and its applications faster?* While developing Carafe and RSort, we kept the focus on preserving the separation philosophy of RDMA. This was made possible only by RStore's API that exposes this separation to our applications. Using the API, our applications successfully identified and prepared resources (but not all) upfront in a distributed setting. Hence, we managed to eliminate overheads stemming from unnecessary buffer allocations, memory registration, data touch operations, scheduling, context switches, RDMA connection openings, metadata fetching from the master etc., from the fast data processing iterations. For data access, Carafe benefited from RStore's highly optimized small IO performance for graph data access; while RSort leveraged RStore's high bandwidth for accessing GBs of KV data.

*Was decoupling of allocation from binding effective?* Following the principle of decoupling helped us to build an efficient caching layer. The caching layer, which caches raw RDMA resources, not RStore's objects, helped us to amortize the control setup cost that cannot be avoided in the fast data access path. For example, the cache hit rate in phase one of sorting was only 10%. In phase two it was 95%, as it was able to reuse cached RDMA-ready memory from phase one.

*Is global synchronization needed on the fast data path?* The thin and fast IO path of RStore does not provide any global synchronization between IO requests. We illustrated that applications with their own explicit coordination mechanism can benefit vastly from very fast data access. However, we realize that such a clean and implicit access ordering and path separation is not always possible to achieve. We are working on building higher-level synchronization primitives such as transactions as illustrated by FaRM [2].

## VII. Related Work

Modern RDMA networks are built upon a large body of work from the 1990s [17]–[20]. RStore leverages and extends these ideas to deliver high end-to-end performance in a distributed setting. RStore's simple master-servers architecture is inspired by the Google filesystem [21]. RStore's use of network striping to achieve high bandwidth is in a similar spirit to Flat Datacenter Storage [22].

Previous attempts to integrate RDMA into distributed systems have looked into using its offloaded technology and high link speed to compensate for the low performance of the socket/TCP stack. These efforts include transparently using RDMA for socket send/recv [23], integration in traditional filesystems [24], [25], and use in MPI applications [26]. RStore's IO operations are similar to get/put operations in Separate Memory Model of Partitioned Global Address Space (PGAS) in MPI-3 [26]. However, unlike MPI-3 applications, RStore's applications separate compute and storage concerns, and use RStore to store data. The computation, which is not tied to data storage, is handled separately by

| | Application Programming Interface | Prefetching, Caching of Dist. Objs. | Unified Bufs. for Zero-copy | One-Sided RDMA Ops. Usage | Low Latency Ops. | Multi-NIC Utilization | Failure and Fault Recovery |
|---|---|---|---|---|---|---|---|
| RDMA/HDFS [16] | HDFS File IO | API Limitation, Implicit | No | None | N/A, SSD bounded | No | Yes, with SSDs |
| Pilaf [3] | Key-Value Store | API Limitation, Implicit | No | Yes, data transfers | $2\times$RDMA READ RTT | No | Async. SSD Logging |
| RAMCloud [1] | Tables, Key-Value Store | API Limitation, Implicit | No | None | Send/Recv. RTT | No | Yes, fast recovery |
| FaRM [2] | TX on Distributed Shared Memory | Possibly | No | Yes, message passing | RDMA RTT + layout adj. | No | Yes, SSD Logging |
| RStore | IO on Distributed Shared Memory | Explicit, App-driven API calls | Yes | Yes, data transfers | RDMA RTT | Yes, IO striping | COW mode (not eval.) |

Table IV: Comparison of related work to integrate RDMA in distributed data storage and processing.

applications. The work from Islam et al. is one of the first ones to propose integration of Infiniband network into the HDFS design [16]. However their proposal does not use one-sided RDMA operations, performs data copies, and suffers from inefficiencies found in HDFS/Java stack for high-performance networks [8], [27]. Recent interest in RDMA has led to many Key-Value store implementations [3]–[5]. However, due to the limitations of the API, these systems do not expose or give control of resource allocation to applications. Furthermore, RStore, in comparison, is a more general-purpose RDMA memory management platform on top of which these applications can be built.

The recently proposed FaRM system is closest to our approach [2]. It provides a general computing platform that leverages RDMA to provide high-performance transactions and lock-free reads to applications. However, FaRM's opaque object API does not let applications pre-allocate or pre-set expensive RDMA resources to avoid their overheads in the object access. Further, FaRM's internal representation of objects (laced with metadata at every cache line) is different from an application's view. Consequently, FaRM must either copy or adjust the layout before giving access to an application. For small object requests (a few KBs), for which FaRM delivers good performance, overheads from DRAM management, together with layout adjustments (data touch operations), are small. Though for large (MBs or GBs) data accesses, these overheads can easily dominate any performance gains. FaRM's transactional object API is arguably better for storing critical system metadata.

Other high-performance in-memory abstractions include RAMCloud [1], Sinfonia [10] and Resilient Distributed Datasets [28]. RAMCloud aims to provide very-low data-access latencies by storing entire data sets in DRAM, and aggregating main memories across hundreds of servers. It uses Infiniband, a high-performance interconnect for fast inter-machine communication, which is limited to fast message passing only [9]. Sinfonia and RDD, despite storing data in memory, do not leverage RDMA for data access, although, they provide more facilities such as fault-tolerance, durability, and straggler handling etc., that RStore lacks. Table IV compares RStore to other high-performance network integration efforts over properties that are discussed throughout the paper.

## VIII. Conclusion

In this paper, we have presented RStore, a distributed, in-memory data store that delivers performance (12 $\mu$sec latency, 705 Gb/s aggregate bandwidth on an iWARP cluster) very close to the network limits. RStore adheres to the separation philosophy of RDMA networks and is built using two design principles: (a) decouple resource allocation from its abstraction binding; (b) keep the data access path thin and fast. RStore's API exposes and extends these ideas to applications, which benefit from pre-allocating and pre-fetching resources in a distributed setting. We demonstrated RStore's capabilities by developing two types of applications on it. Our first application Carafe, which is a low-latency graph processing system, outperformed state-of-the-art systems by margins of $2.6-4.2\times$. Our second application RSort is an order of magnitude faster than Hadoop TeraSort for sorting distributed key-value tuples. The key reasons for the high application performance are the design and abstraction choices that RStore is based upon. We believe that these abstractions provide a solid foundation for building future network-centric data-processing systems.

## Acknowledgement

We thank the anonymous reviewers for their helpful comments and suggestions. We also would like to thank Steve Wise, Felix Marti, Robert Birke, and Joerg-Eric Sagmeister for their support with the infrastructure. Sebastiano Spicuglia helped with the initial implementation of RSort. Anne-Marie Cromack helped with proof-reading of the paper.

## References

[1] J. Ousterhout *et al.*, "The case for ramclouds: Scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[2] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14, 2014, pp. 401–414.

[3] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *Proceedings of the 2013 USENIX ATC*, pp. 103–114.

[4] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost Memcached," in *Proceedings of the 2012 USENIX ATC*, pp. 347–353.

[5] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. Islam, X. Ouyang, H. Wang, S. Sur, and D. Panda, "Memcached design on high performance rdma capable interconnects," in *International Conference on Parallel Processing (ICPP)*, 2011, pp. 743–752.

[6] P. W. Frey, "Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks," in *PhD dissertation (NO. 19001) submitted to ETH Zurich*, 2010.

[7] P. W. Frey and G. Alonso, "Minimizing the hidden cost of rdma," in *Proceedings of the 29th IEEE ICDCS*, 2009, pp. 553–560.

[8] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "DaRPC: Data Center RPC," in *The Proceedings of the Fifth ACM Symposium on Cloud Computing*, ser. SoCC '14, 2014.

[9] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the 23rd ACM SOSP*, 2011, pp. 29–41.

[10] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *Proceedings of 21st ACM SOSP*, 2007, pp. 159–174.

[11] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD*, 2013, pp. 505–516.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD*, pp. 135–146.

[13] "Livejournal social network at Stanford Network Analysis Project, https://snap.stanford.edu/data/soc-livejournal1.html."

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX OSDI*, 2012, pp. 17–30.

[15] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX OSDI*, 2014, pp. 599–613.

[16] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High Performance RDMA-based Design of HDFS over InfiniBand," in *Proceedings of the SC '12*, pp. 35:1–35:35.

[17] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proceedings of the 21st ISCA*, 1994, pp. 142–153.

[18] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes, "An implementation of the hamlyn sender-managed interface architecture," in *Proceedings of the 2nd USENIX OSDI*, 1996, pp. 245–259.

[19] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: A user-level network interface for parallel and distributed computing," in *Proceedings of the 15th ACM SOSP*, 1995, pp. 40–53.

[20] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, ser. SIGCOMM '94, 1994, pp. 2–13.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th SOSP*, 2003, pp. 29–43.

[22] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage," in *Proceedings of the 10th USENIX OSDI*, 2012, pp. 1–15.

[23] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets direct protocol over infiniband in clusters: Is it beneficial?" in *Proceedings of the IEEE ISPASS*, 2004, pp. 28–35.

[24] B. Li, P. Zhang, Z. Huo, and D. Meng, "Early Experiences with Write-Write Design of NFS over RDMA," in *IEEE International Conference on Networking, Architecture, and Storage*, July 2009, pp. 303–308.

[25] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber, "Structure and Performance of the Direct Access File System," in *Proceedings of the 2002 USENIX ATC*, pp. 1–14.

[26] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in MPI-3," in *ACM Trans. Parallel Comput. (Mar. 2013)*.

[27] P. Stuedi, B. Metzler, and A. Trivedi, "jverbs: Ultra-Low Latency for Data Center Applications," in *The Proceedings of the Fourth ACM Symposium on Cloud Computing*, ser. SoCC '13, 2013, pp. 10:1–10:14.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th NSDI*, 2012, pp. 15–28.