Remote regions: a simple abstraction for remote memory

Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei, *VMware*

https://www.usenix.org/conference/atc18/presentation/aguilera

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Remote regions: a simple abstraction for remote memory

Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi
Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam
Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, Michael Wei
VMware

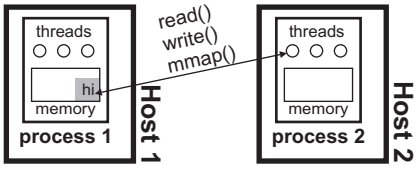
Abstract

We propose an intuitive abstraction for a process to export its memory to remote hosts, and to access the memory exported by others. This abstraction provides a simpler interface to RDMA and other remote memory technologies compared to the existing *verbs* interface. The key idea is that a process can export parts of its memory as files, called *remote regions*, that can be accessed through the usual file system operations (read, write, memory map, etc). We built this abstraction in the Linux kernel, and evaluated it. We show that remote regions are easy to use and perform close to RDMA. We demonstrate it via micro-benchmarks and by adapting two in-memory single-host applications to use remote memory: R and Metis. With R, using remote regions requires no changes to the code and allows R to run with remote memory that exceeds the physical memory of a host. With Metis, the modifications amount to ≈100 lines of code and they allow Metis to scale its performance across 8 hosts.

1 Introduction

Remote memory allows a process to read and write the memory of another process in a different host. This is an exciting idea whose time has come [1]. Remote memory is available now, using RDMA technology over Infiniband or Ethernet [49, 29], and other new technologies are emerging [28, 24, 47]. Many applications are being redesigned to use remote memory (key-value storage systems [43, 14, 30, 15], database systems [50, 6, 64], map-reduce [39], etc).

Unfortunately, remote memory faces two problems now. First, it has no standard interface. Current technology uses the RDMA verbs interface, but new hardware such as Gen-Z and OpenCAPI will have their own interfaces to control mapping, access, etc. Even RDMA is still changing with key innovations, such as DCT [18], that are offered in some implementations but not others. Second, remote memory today is hard to use. With RDMA, even the simplest program to access some data from a remote host requires a complex ritual: code is required to initialize contexts, register memory, establish RDMA connections, create queue-pairs, associate them with connections, transition the queues through various states, exchange RDMA keys, post commands on queues, and poll the queues for completions [7]. Furthermore, RDMA lacks naming and location services that applications need, forcing them to reimplement this functionality every time.



Host 1	Host 2
fd=create("/regions/oz", 0644);	fd=open("/regions/oz", O_RDONLY);
ptr=rmalloc(fd, 1024);	buf=malloc(1024);
sprintf(ptr, "alice");	lseek(fd, pos_buf, 0);
close(fd);	read(fd, buf, 1024); /* gets "alice" */
	close(fd);

Figure 1: Using regions, Host 1 creates a region named *oz* in REGIONFS, allocates a buffer in the region, and populates the buffer. Host 2 then reads host 1's string, similar to an RDMA-read operation, except that developers need not program with RDMA directly (which is complex).

In this paper, we propose a simple idea: to use files as the interface to remote memory, shedding the complexity of RDMA and providing a standard for new technologies. In particular, we propose *remote regions* or, in short, *regions*. With regions, a process exports parts of its memory as *files* in REGIONFS, a file system that a remote host can then access using the usual file operations (read, write, memory map, etc). In addition to a simple interface, regions draw features from file systems to provide functionality lacking in RDMA: name space, timestamps, access control, etc (§4).

Regions are simple because they replace low-level RDMA mechanisms with high-level controls that are operated through a familiar interface. Figure 1 shows how easily a host can use regions to read data in the memory of another host. By contrast, equivalent RDMA logic takes around 300 lines of hard-to-understand code [7].

The main challenge in designing regions is to find the right balance between elegance, expressiveness, and efficiency, while overcoming the limitations of the hardware. To find this balance, we address questions of file semantics, memory allocation, data sharing, memory mapping, page fault preemption, security, data-metadata separation, caching, cache coherence, and sharing granularity, while addressing RDMA limits on memory registration, connections, and keys. The current implementation of regions targets RDMA, but we believe region's interface will be applicable to new upcoming remote mem-

ory technologies [24, 28], providing a common abstraction with which applications can be written in a portable fashion across these new technologies.

We have built regions using RDMA in the Linux kernel v4.8, and we evaluated their cost on a cluster of 8 machines with RoCE. Using microbenchmarks, we see that accessing data via regions is reasonably close to RDMA. We have used regions to extend two applications, R [48] and Metis [41], to use remote memory. R is a system for statistical processing of data, while Metis is an in-memory implementation of map-reduce running on a single host. We use regions to adapt R to operate on large data sets in remote memory exceeding the available local memory capacity. We do this by using R's ff package to store objects in memory-mapped files, and placing these files in REGIONFS. We also use regions to produce a distributed version of Metis that runs across many hosts, sharing in-memory data. This change required only 82 lines of code, and allows Metis to scale to 8 hosts, giving it more memory and improving performance by 3.5× compared to a single host.

2 Related work

Same interface, different goal. The file interface is often used for *remote storage*, where the main goal is to provide durable storage capacity. Several such systems use RDMA to improve performance, such as Octopus [40], Crail [57], Ceph [11], and GlusterFS [25]. DAFS [13] is a file system protocol for RDMA, while [60] is a proposal to run NFS over RDMA. The file interface can also be used to manage large local memories [58]. All of the above works rely on a file interface but have a different goal from our goal of accessing the memory of remote applications.

Different interface, same goal. Prior work provides remote memory with a different interface. LITE [61] provides a kernel interface that offers more flexible protection, and better scalability and isolation than verbs on RDMA. There is much work in distributed shared memory (DSM) (e.g., [9, 33, 46, 2, 51, 53, 56, 8]) including recent work on persistence using non-volatile memory and replication [54]. FaRM [14, 15] provides transactions over RDMA with lock-free reads. All these systems provide a simpler interface than RDMA, but they do not support the well-known file interface, which has many advantages (§4).

Different interface, different goal. Many systems provide remote storage with an interface other than files, including key-value stores (e.g., [43, 14, 30, 15]), Linda tuples [10], distributed objects (e.g., [63, 27, 5]), and database systems. These systems offer a different abstraction from regions. For example, key-value stores provide GETs and PUTs on key-value pairs; Linda provides a tuple interface; distributed objects require applications to declare and manipulate the objects provided by the framework; and database systems use SQL.

Remote memory applications. Several works have proposed replacing disks with remote memory as a faster target for swapping or paging (e.g., [23, 12, 34, 21, 31, 22, 17]). CacheDM [36] uses remote memory as a cache for a network file system, while Infiniswap [26] uses remote memory as a cache for a local swap/paging device. Several of these applications are built with RDMA; they might have been simpler to develop with regions.

New hardware. Disaggregated memory proposes a new system architecture that detaches memory from machines and places it on a common fabric. The work includes academic papers [26, 37, 20, 4, 23, 45] and upcoming technologies to support it, such as Gen-Z [24] and Omni-Path [28]. Regions could provide an elegant interface to disaggregated memory, though the implementation of regions will differ from the RDMA implementation we give (that will depend on the details of these technologies, which are still work in progress).

3 Assumptions, goals, and motivation

We assume machines are connected to a network with low latency, high bandwidth, and reliable connectivity—such as, for example, machines in a few racks in a data center. We assume a single administrative, trust, and fault domain. We consider deployments with a couple to tens of machines. While some companies have large deployments with thousands of machines, the vast bulk of our customers are enterprises with deployments of 100 or fewer machines in a private facility, and that is our target environment. Network partitions are rare and, when they do occur, it is reasonable for the system to pause as the rest of the system will be unavailable anyways (e.g., network file systems and other servers are unreachable).

Our goal is to provide abstractions for applications to access the memory of other applications across the network. Currently, the standard way to do that is to employ one-sided read and write operations using the verbs library (libibverbs [35]). This interface has three issues that we want to overcome:

- *Complexity.* As we mentioned, verbs operations are complex, and we seek simple and intuitive alternatives.
- *Dependency on existing technology.* There are other remote memory technologies under development other than RDMA, such as Omni-Path [28] and Gen-Z [24]. We would like to find high-level abstractions so that applications can be portable across these technologies.
- *Resource limitations.* RDMA has limitations on resources at the network adapter, such as limited cache sizes for connections and memory translations [14]. We

want to design abstractions that can hide or overcome these limitations without concerning applications.

We expect a simpler interface to have performance costs but want them to be reasonable and we certainly want to understand them, much like a developer needs to understand the cost of other high-level features, such as garbage collection, lambdas, etc.

4 Why files?

By using a file interface, regions get many benefits:

- *Well-known.* All developers know files.
- *Utilities.* The file interface inherits a vast repertoire of utilities: editors, backup, grep, find, cp, cat, sed, awk, etc. Regions allow these to be used with remote memory.
- *Language support.* Most of the functionality of regions is in REGIONFS, and all major programming languages support files. There is only a small library (with synchronization and stub functions) that needs to be ported to a given language.
- *Interposition support.* There are many tools to interpose on file system calls, for tracing, debugging, auditing, and profiling (e.g., DFSTrace [44]). These tools all work with REGIONFS.
- *Name space.* Directories and files make it easy to find and organize data across applications in the network.
- *Users and access permissions.* Applications can use the notion of users from the operating system combined with access permissions to control who has access.

We get these benefits for free because the file interface is well matched to our problem. In contrast, other interfaces to remote memory, such as RDMA, provide none of these benefits.

5 The regions abstraction

We now explain how regions appear to users as an abstraction, and we explain how we arrived at this abstraction. We show how to provide the abstraction in §6.

In its simplest form, a (remote) region is a logically contiguous part of the memory of a process, called the *owner* process. The owner creates a region like a file, and can operate on it by memory mapping, reading, writing, or allocating variables using a special *rmalloc* function (§5.2); these operations refer to data in local memory. Processes in other hosts can also perform these operations, to access data in the memory of the owner.

5.1 Basic functions

Regions provide a file system called REGIONFS mounted in a known location, such as /regions. Each file in REGIONFS is a region stored in memory, either locally or remotely. REGIONFS supports the usual file operations (e.g., creat, unlink, open, close, read, write, chmod, stat) in addition to mmap (§5.7). By default, a region disappears when its creator process terminates or crashes (accessing it results in an I/O error).

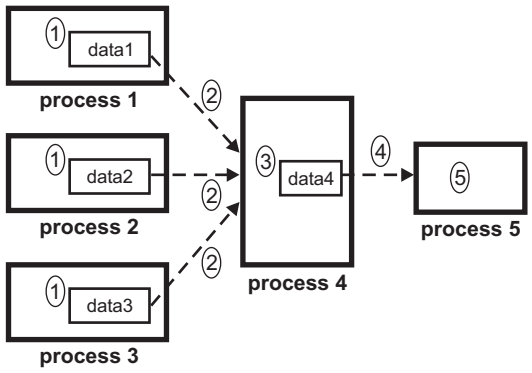


Figure 2: Example of the use of regions on a map-reduce style of computation. (1) Three mapper processes in different hosts create a region each, run some computation, and store the results in their region; (2) a fourth reducer process reads the data in the regions, and (3) creates a region and writes the result there; (4) a fifth process reads that region and (5) produces a graph in the display of a user.

A directory in REGIONFS is not a region but organizes regions, much like regular file systems; but unlike regular file systems, directories carry some special extended attributes that regions inherit upon creation (§5.4).

5.2 Memory allocation

An application often dynamically allocates and destroys many buffers in its lifetime. Rather than creating/deleting a region for each buffer, applications can dynamically allocate/free buffers *within* a region, using these functions:

```
void *rmalloc(int regionfd, size_t len)
int rfree(void *ptr)
```

where *regionfd* is a descriptor for a region open in write mode. Calling *rmalloc* is faster than creating a region: the former executes entirely in memory, while the latter requires contacting a metadata manager over the network (§6.6). In fact, an application might create just one region and then allocate its buffers within that region.

5.3 Example of usage

We illustrate the use of a region with an example with five processes that run a map-reduce style of computation (Figure 2). In existing map-reduce systems, processes exchange data using a distributed file system such as the Hadoop Distributed File System (HDFS) [3]. With regions, processes can exchange data directly in memory, as with RDMA, but with the simplicity of using files. Also, this is distinct from using an RDMA-enabled file system (e.g., [40, 11, 25]), where processes store data in a storage server and use RDMA to access the server; with regions, processes can directly export data in *their* memory and read data from the memory of other processes.

Attribute	Type	Description
OWNERPID	pid_t	pid of process owning region
PERSISTENT	bool	keep region when process ends (§5.5)
MULTIHOSTED	bool	store region across many hosts (§5.6)
FIXVADDR	uint64_t	fixed virtual address (§5.7)
ONEWRITER	bool	only one process can open for writing
HOSTALLOC	ip list	hosts storing region if MULTIHOSTED (§5.6)

Figure 3: Region-specific attributes.

5.4 Region attributes

Beyond the attributes of a typical file (access bits, uid, gid, etc), each region has some additional region-specific metadata that determine certain behaviors (Figure 3). The owner indicates the process who created the region; this is different from the owner uid of a region/file, which is a user. When this process ends, the region is automatically deleted unless the PERSISTENT attribute is set (§5.5). A region gets a fixed virtual address across hosts if FIXVADDR is set (§5.7). A region can be opened for writing by at most one process if ONEWRITER is set; this is enforced across hosts. When a region grows in size, new memory is typically allocated from the host of the owner, but it is possible to allocate it from remote hosts as well if MULTIHOSTED is set, in which case HOSTALLOC indicates the hosts to allocate from (§5.6).

Applications set these attributes when the region is created. Since we use the standard creat() call to create regions, which cannot specify attributes, we define an additional function

```
int rsetdefaultattr(int attr, char *val, int len) /* returns error flag */
```

that sets the default attributes of new regions for the calling thread.

5.5 Persistent regions

By default, a region is backed by the memory of a process. If the process terminates, its memory is deallocated and the region is automatically deleted. This could be undesirable in some cases: the process might wish to leave the data in memory for a short while until it is consumed by another process. One solution to this problem is for the process to defer its termination until its data has been consumed. This solution is complex because it requires the process to coordinate with other applications.

We provide a simpler solution: to retain the region contents after the process terminates. Upon termination, a process releases its memory but not the region. We call such regions *persistent* regions. Persistent regions should be deleted by the consuming process later. They are also deleted when the host reboots. To create a persistent region, a process sets the attribute PERSISTENT.

5.6 Multi-hosted regions

A multi-hosted region is a special type of region that is stored across many hosts. These regions can store large data that exceed the physical memory of any single host.

To create a multi-hosted region, a process sets attribute MULTIHOSTED and optionally chooses the hosts where the region will be allocated via attribute HOSTALLOC (§5.4); if this is not set, the default is to use all hosts.

5.7 Memory mapping

Processes can memory map a region using mmap(), so that the region can be accessed by memory operations instead of read() and write(). The function returns a pointer where the region is mapped. If a region is created with the FIXVADDR attribute, it is given a fixed virtual address [54]: it always maps to that address, no matter which process or host maps the region. This ensures that pointers to data in regions remain valid across hosts, allowing regions to store dynamic data structures and other data that require indirection. To implement this feature, we reserve virtual addresses across the cluster (§6.10).

5.8 Performance enhancing functions

Memory mapping of a region on a remote host is implemented using page faults. Page faults have two causes: (1) when a process first accesses a page, to fetch the page; (2) when the process first writes to the page, to mark it dirty. If the first access is a write, one page fault both fetches and marks it dirty. Because page faults are expensive, we provide two ways to prevent them: prefetch and mark-dirty. With prefetch, applications request the system to fetch pages immediately, by calling

```
int rprefetch(void *addr, size_t len, bool sync) /* ret: error flag */
```

which prefetches data in a region starting at *addr* with length *len*; if *sync* is set, it waits until the data it fetched. To avoid page faults due to writes, applications can request the system to mark the page dirty, by calling

```
int rmarkdirty(void *addr, size_t len, bool zero) /* ret: err flag */
```

before writing to a page. If parameter *zero* is true, this function zeroes the pages without reading their contents. This is useful to avoid the overhead of a read-modify-write cycle if the application intends to completely overwrite the pages (see §8.3).

Function rprefetch is just an optimization that does not change application semantics. Function rmarkdirty is also an optimization when parameter *zero* is false; if *zero* is true, rmarkdirty is equivalent to bzero().

5.9 Synchronization

When using regions, one might need to synchronize processes across hosts (e.g., to share data, as in §5.3). We provide several distributed synchronization primitives: barriers, mutexes, and door bells (Figure 4). These are

Function	Description
rbarrier_init(name, n)	Create barrier for *n* callers
rbarrier_wait(name)	Wait for barrier
rmutex_init(name)	Create mutex
rmutex_lock	Acquire mutex
rmutex_unlock	Release mutex
rbell_init(name)	Create door bell
rbell_ring(name)	Increment bell value
rbell_wait(name)	Wait for new value, return it
rdelete(name)	Deallocate

Figure 4: Available synchronization primitives.

Type	Regions	RDMA
Owner-remote	Owner writes to region and remote process reads, or remote process writes to region and owner reads	One process writes locally and another RDMA-reads, or one process RDMA-writes and another reads locally
Remote-remote	A remote process writes to region and a remote process reads from region	One process RDMA-writes to third party's memory and another RDMA-reads

Figure 5: Two patterns of sharing data between processes in different hosts using regions and the analogue using RDMA.

offered in a user library, since a file system has no such functionality. A barrier has a parameter *n* and the caller blocks until the barrier has been called at least *n* times. This serves to synchronize a group of processes. A mutex ensures at most one caller gets the mutex at once. A door bell has an initial value 0; the ring function increments it; the wait function waits for it to be incremented since its last call, returning the current value.

5.10 Caching

When a process uses a region of a different host, the system locally maintains a page cache of data that has been recently read or that has been modified. The cache is a write-back cache (modifications are propagated back to the region in the background). The system does not provide cache coherence, because it is too expensive; rather applications can obtain coherence at the moments of their choice by explicitly using two mechanisms, flushing and clearing caches:

```
int msync(void *addr, size_t len, int flags) /* returns error flag */
int rclearcache(void *addr, size_t len) /* return error flag */
```

where addr is the address within one of the open regions. Flushing [msync] causes dirty pages to be written back to the region, so the owner can observe the modified data. Clearing pages [rclearcache] removes them from the cache, so that the calling process subsequently obtains fresh data from the owner. These functions produce an effect only at a process remote to the region or for multi-hosted regions, as the owner of single-hosted region does not have a cache. After clearing a page, a process might invoke rprefetch() to avoid a page fault (§5.8).

Processes sharing a region must follow some discipline on how to use these functions to avoid data corruption. We propose a simple and effective scheme in the next section.

5.11 Sharing data

To correctly share data, processes must flush and clear their caches carefully. Doing so is not easy in general, but we now describe a simple scheme that works well in the use cases that regions are designed for. To explain how this is done, we broadly classify sharing of data alongside two dimensions.

The first dimension is who participates in the sharing relative to who owns the data. There are two possibilities: owner-remote sharing and remote-remote sharing (Figure 5). With owner-remote sharing, one of the processes sharing owns the region or the memory. With remote-remote, the process that owns the region or memory is a third party. Owner-remote sharing is simpler to deal with, because there is only one cache involved (the cache of the remote process), while remote-remote sharing involves two caches, one for each remote process.

The second dimension is what we call the *granularity of sharing*. With fine-grained sharing, processes interleave their execution often and share small bits of data (e.g., one or a few variables) at a time, with frequent coordination. For example, in a mutual exclusion algorithm, two processes frequently read and write common variables containing the state of flags or counters, often changing the role of who reads and writes the shared information. With coarse-grained sharing, one process produces a large chunk of data before another process consumes it; for example, in the map-reduce computation of Figure 2, the mappers produce large outputs that are later consumed by the reducer.

We anticipate that regions will be used for both owner-remote and remote-remote sharing alongside the first dimension, but only for coarse-grained sharing alongside the second dimension, because fine-grained sharing over the network is generally too costly. Coarse-grained sharing does not require the cache to be coherent very often: it suffices to be coherent in the instant after the producer has finished writing and before the consumer starts reading. Accordingly, processes can flush or clear their caches at that moment, as follows. With owner-remote sharing, the remote process either flushes or clear its cache, depending on whether it is producing or consuming data. With remote-remote sharing, the remote process that produces data flushes its cache, while the remote process that consumes data clears its cache.

5.12 Pseudo file system

Regions have more metadata than files. We expose this metadata to users in a pseudo file system /proc/regions,

File name	Description
hosts	List of hosts using regions
memusage	Aggregate memory usage of regions hosted locally
pools	List of pools (§6.7) with used/free space and daemon's logical address
daemon	pid of daemon process (§6.4)
procs	pid of local processes using regions
files/*pathname*	Metadata of region in *pathname*: vaddr, maximum size, pool, attributes

Figure 6: Region metadata stored in pseudo file system /proc/regions.

accessible only by root. Available Information includes local memory usage of regions, a list of local pools, and the local processes using regions (Figure 6). Moreover, for each region *r*, /proc/regions/files/*r* indicates the region's fixed virtual address, maximum size, pools from which memory is allocated, and attributes.

5.13 Limitations

Regions have a limitation: a process cannot use them to export data in its stack or static variables, because the process must allocate data in regions using rmalloc. However, these limitations may not matter: it is easy to change static variables to heap variables, and it is probably a bad idea for applications to export data in the stack, since that data disappears when its call frame is deleted.

6 Realizing regions using RDMA

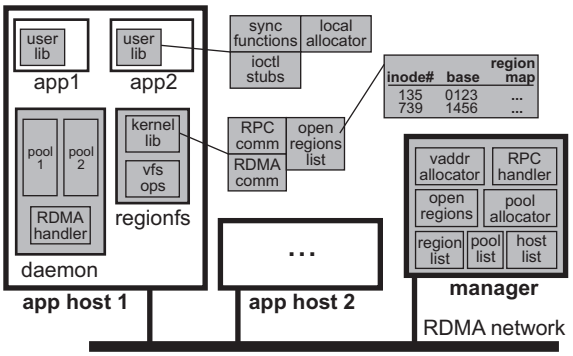
We now describe how we realize regions using RDMA. While the design is centered around RDMA, we expect that its key ideas will be applicable to future disaggregated memory hardware.

6.1 Basic architecture

Figure 7 shows the architecture of regions. There are four main logical components: REGIONFS file system, user library, daemon, and manager. Broadly, the REGIONFS file system component implements the VFS kernel operations required of a file system, while the user library implements synchronization and performance-enhancing functions. The first module is instantiated once per host; the second, once per application. The daemon (one per host) allocates and maintains large pools of memory in which regions are allocated, and shares these pools with both local and remote processes. The manager provides the control plane, handling every file system operation except reading and writing data. The manager has one instance but it is replicated for high availability using standard mechanisms, such as Paxos state machine replication [32, 52]. We provide more details in the next sections.

6.2 RegionFS file system component

This component is a kernel module that implements the file system for region, with functionality to drive the



Module	Description	Section
regionfs	file system for regions	§6.2
user lib	user-level file system	§6.3
daemon	holds and exports memory pools	§6.4
manager	handles control operations	§6.5
sync funcs	barrier, mutex, doorbell	§6.3
local alloc	kmalloc	§6.3
ioctl stubs	functions without VFS analogues	§6.3
kernel lib	communication library	§6.12
RPC comm	for communicating with manager	§6.12
RDMA comm	for communicating with daemons	§6.12
open regions list	list of regions that client has opened	§6.2
region map	tracks where a region is stored	§6.8
vfs ops	VFS interface to file system	§6.6
RDMA handler	accepts RDMA connections	§6.12
vaddr allocator	allocates virtual addresses	§6.10
RPC handler	handles requests from clients	§6.12
open regions	all open regions in the system	§6.5
pool allocator	allocates cluster memory	§6.9
region list	all regions in the system	§6.5
pool list	all pools in the system	§6.5
host list	keep track of hosts	§6.5

Figure 7: Architecture. Region components are in gray. The figure shows two application hosts, but we expect a few dozens of them. There is a single manager, and it is replicated for fault tolerance. The manager is involved only in infrequent control operations, staying out of the performance-critical data path.

execution of file system operations, coordinating with the manager and the other hosts' daemons. The module keeps an important data structure, the open region list, which tracks all regions that the application has opened, with their virtual addresses, and map for locating the data within the region. We detail the VFS operations in §6.6 after some more background, but they fall into two categories: Data operations (read, write, readpage, etc) execute locally or over RDMA, depending on where a region resides. Metadata operations (directories, file attributes, etc) are similar to the implementation of a network file system (e.g., to create a directory, it calls the manager, which then records information about the directory).

6.3 User library component

The user library provides the synchronization functions (§5.9), an allocator for rmalloc (§5.2), and ioctl stubs. The synchronization functions issue an RPC to the

manager, which implements the actual functionality. The RPC call blocks until the synchronization occurs (e.g., a barrier gets all its participants). For the rmalloc allocator, we memory map the region (if it has not been mapped already) and organize the space using a buddy allocator. The allocator uses a magic number at the beginning of the region to know if the allocator structures must be initialized. The ioctl stubs provide region-specific functions without a corresponding VFS operation (rsetdefaultattr (§5.4), rprefetch (§5.8), rmarkdirty (§5.8), rclearcache (§5.10)). The stubs translate these functions into ioctl's that get handled by VFS.

6.4 The daemon

The daemon serves three purposes. First, it overcomes resource limitations of the RDMA network adapter, which cannot keep many connections or export many buffers because its internal cache is small [14]. To address these problems, the daemon allocates big pools of physical memory (§6.7) and then allocates regions within these pools. Thus, a host exports a few pools (rather than many regions) and a remote process can connect just with the daemon to access the data of all applications in the host (instead of connecting to each application). Second, the daemon allows a host to offer memory to multi-hosted regions (§5.6) even if the host has no running applications. Third, the daemon supports persistent regions (§5.5) by holding the region's data when a process terminates.

6.5 Manager

A central manager handles all control operations: creation, opening, closing, deletion, and memory-mapping of regions; file system metadata operations (create and delete directories, set and get inode attributes); allocation of memory for regions; and allocation of global virtual addresses. To do that, the manager keeps track of the hosts in the system, file system metadata (inodes and directory contents), memory usage of all pools at each host, allocation of regions, allocation of virtual addresses, and list of all open regions. The manager is not involved in reading and writing data in regions—the performance critical operations—so it is not a bottleneck. However, a larger system might require distributing the manager.

6.6 VFS operations

To open a region [open()], the client calls the manager; if the region exists, the manager returns its starting virtual address and region map (§6.8); the client adds the region to its open regions list and stores its virtual address and region map. To create a region, the client calls the manager to check if the region already exists, to preallocate an initial set of pages to it, to allocate virtual addresses (§6.10), and to return the starting virtual address and region map, which the client stores.

To read a region [read()], the client consults the region map and issues RDMA read(s) to the proper host(s); it then copies the result to the user-provided buffer. To write a region [write()], the client checks if the write falls outside the preallocated space for the region; if it does, it contacts the manager to extend the region and the region map; then, the library copies the user-provided buffer into RDMA-registered memory, consults the region map to determine the host(s) to contact, and issues RDMA-write(s) to the proper host(s).

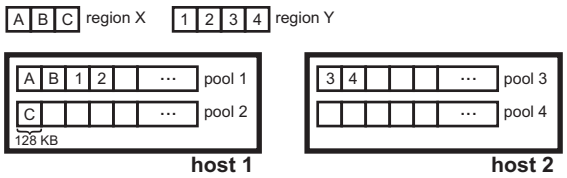


Figure 8: Four pools in two hosts. Region X stores its data in two pools of a host. Region Y is multi-hosted, spanning pools of two different hosts.

To prefetch data [ioctl for rprefetch()], the client consults the region map to determine where to read the data from, reads over RDMA, and places it in the file system cache. Similarly, to write back [msync()] a page, the client consults the region map, write-protects the page, writes the page over RDMA, and marks the page clean. To mark a page dirty [ioctl for rmarkdirty()], the client sets the dirty bit for the page. To clear a page from the cache [ioctl for rclearcache()], the client evicts it from the file system cache.

6.7 Pools

A *pool* is a chunk of physical memory, at one of the hosts, that is used to store a region or parts thereof (Figure 8). Pools are allocated by the daemon (§6.4) and are shared with local processes (using shared memory) and with remote processes (using RDMA). To share locally, the daemon allocates its pools using anonymous files [38], which are chunks of anonymous memory that can be memory mapped at many processes. More precisely, the daemon creates a pool using memfd_create; then, an application process can memory map the pool at the addresses that correspond to a region that the process needs. Regions need not be contiguous within a pool; however, to reduce the number of memory maps, the daemon allocates the region in large contiguous chunks.

To share its pools with remote hosts, the daemon RDMA-registers each pool so that it can be read and written over RDMA. RDMA provides access control through a key for each buffer that a host exports. Because a pool is a single buffer, this mechanisms is coarse-grained: it provides identical access to all data in the pool.

6.8 Finding region data

A *region map* tracks where a region is stored, by mapping offsets in a region to a host, a pool in that host, and

an offset in that pool. The map has entries, each representing a fixed-length contiguous chunk of memory on one pool at one host. There are two aspects to this map: its granularity and how to represent hosts and pools.

Granularity. The granularity involves a trade-off between the size and the flexibility of the map. A small grain leads to a prohibitively large map; a large grain leads to internal fragmentation. A natural size might be the page size (4 KB), but that causes a significant 0.2% space overhead for the map (e.g., 2 GB for a 1 TB region). We chose the granularity to be 128 KB for an overhead of 64 KB for a 1 TB region.

Target representation. We want map entries to take at most 64 bits. We use 47 bits to represent a 64-bit address within a host, by dropping the lower 17 bits and aligning over 2^{17} =128 KB chunks, which coincides with the map granularity above. We use the remaining 17 bits as a global identifier that maps to a host and a pool in that host; this map is kept by the manager ("pool list" box in Figure 7) and cached by the user library.

6.9 Managing memory

There are two aspects to memory management: local and cluster allocation.

Local allocation. Each host has limited memory and one needs to decide how much to reserve for pools and regions. We make this determination locally at each host, where the daemon allocates and frees pools as needed.

Cluster allocation. When an application needs memory, one needs to decide which pool(s) to use; for multi-hosted regions, one needs to also decide which hosts will provide memory. We make this determination in a centralized fashion: the manager knows about all participating hosts, their pools, and the free space in each pool ("pool allocator" box in Figure 7). The manager receives requests to create new regions, with an initial space to preallocate for future region growth. It then decides from what pools to allocate the memory using some allocation policy. The current policy is as follows. For regions in a single host, the manager picks from the pools in that host; if the host does not have enough memory, it asks the daemon to create more pools; if the daemon is unable, the request to create or expand a region fails. For multi-hosted regions (§5.6), the manager picks memory from the hosts in a round-robin fashion, allocating ALLOCSIZE$\geq 2^{17}$ bytes at a time. Note that ALLOCSIZE becomes the maximum contiguous size that a client can transfer in one RDMA request. We pick ALLOCSIZE to be 2 MB—a value large enough to offset the initial fixed costs of an RDMA transfer (with a 40 Gbps network, the initial cost to transfer 2 MB is 0.3% of the total cost).

6.10 Allocating virtual addresses

Regions are assigned a fixed and unique virtual address (§5.7). Therefore, we must ensure that (a) different regions get assigned disjoint virtual addresses, even if they are created by different applications in different hosts, and (b) an application will not use a region's virtual addresses for other purposes. To ensure (a), we use centralization: region creation goes through the manager, who knows about all virtual addresses in use by regions. The manager assigns unique virtual addresses to each region ("vaddr allocator" box in Figure 7). To ensure (b), we reserve a range of virtual addresses for regions using the dynamic linker responsible for loading binaries. There are many ways to do that in Linux. First, we can specify an ET_EXEC object file type in the ELF binary and then create a program header with attributes p_vaddr and p_memsz, indicating the address and size of the virtual address to reserve [19]; this requires statically linking all libraries. Second, we can use a custom dynamic linker that avoids the virtual addresses reserved for regions; we do that by including in the ELF binary an INTERP program header with the path to the linker [16]. These approaches require modifying the application binary. A third approach, which requires no binary changes, is to modify the default dynamic linker, ld-linux.so.

Are there enough virtual addresses? Today, Intel processors use page tables with four levels, addressing 48 bits of addresses; one bit is used by the Linux kernel, leaving 47 bits for applications. If we reserve another bit for regions, that leaves 64 TB for each application and 64 TB for all regions. If that is not enough, Intel plans to support five-level page tables, which add 9 bits of virtual addressing [55]; reserving one bit for regions gives 32 PB for each application and 32 PB for all regions.

6.11 Security

We enforce access control using the file system, which assumes that the kernel is trusted. This provides reasonable security against damage from bugs and human errors, but an attacker of a host gets access to the regions in every host. Providing stronger security is future work.

6.12 Other modules

The kernel lib consists of two kernel modules: (1) RPC comm module implements RPC's to the manager, and (2) the RDMA comm modules establishes a reliable RDMA connection to remote hosts and implements one-sided RDMA read and write. The RPC handler module at the manager handles RPC requests from clients. The RDMA handler at the daemons registers the pools with RDMA, reports the RDMA key and pool address to the manager so that clients can later access the pool, and accepts reliable RDMA connections from remote daemons.

System	Description
rdma	RDMA read or write
nfs-tmpfs	NFS to a ramdisk
tmpfs	local ramdisk
rr	regions
rr+	regions with prefetching

Figure 9: Baselines (top) and systems under study (bottom).

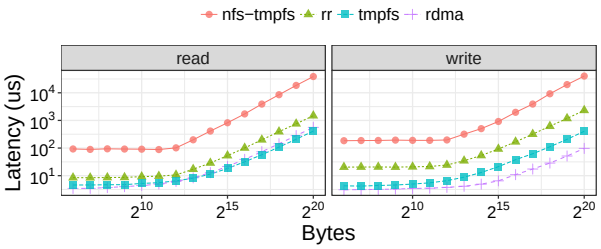


Figure 10: Latency for transferring data (no caching).

6.13 User-level file interface

An earlier version of REGIONFS was implemented as a user-level file system [42] and a user-level page-fault handler [62]. We found that data operations were faster, because they could use RDMA's user-level interface. However, page fault handling was slower. As future research, it would be interesting to explore a hybrid design that provides both user-level and kernel interfaces to the *same* file system to get the best of both worlds.

7 Implementation

We implemented a prototype of regions for the Linux kernel v4.8 with 7700 lines of C/C++. Our current implementation differs from the design in a few significant ways: (1) we do not replicate the manager, (2) at each daemon, we have a fixed number of pools, hence a fixed amount of memory for regions, and (3) our VFS file system implements only the functionality needed to run our applications and benchmarks.

8 Evaluation

Our goal is to understand how well do regions perform, and how easy it is to use them in practice. To answer these questions, we use micro-benchmarks, examine code complexity, modify two applications to use regions, and measure their performance.

8.1 Testbed

Our testbed has 8 machines connected to a 100 Gbps RoCE switch. Each server has 128 GB RAM, a 800 GB SATA SSD, dual Intel Haswell-EP 2.4 GHz processors with a Mellanox ConnectX-4 NIC and Linux kernel 4.8.

8.2 Baselines

We compare the performance of regions against three baselines (Figure 9). RDMA offers a different interface to remote memory (RDMA verbs). Nfs-tmpfs and tmpfs provide a similar interface as regions (files), but without access to remote memory: nfs-tmpfs accesses files in a RAM disk of the NFS server, while tmpfs accesses files in a local RAM disk without network overheads, representing an upper bound on achievable performance.

We consider two variants of regions (rr and rr+) without and with performance enhancements that we describe in each experiment. In all experiments, we configure a region to be stored remotely from the benchmark or application operating on it, and so they access the region over the network rather than locally.

8.3 Performance of memory-mapped access

Setup. We study the time it takes for regions to read and write memory-mapped data. In an experiment, we memory map a file or region, and then sequentially read or write bytes. We choose an operation type (read or write), and operation size (number of bytes to read or write), and repeat the operation 100 times, measuring the latency of each operation. We compare regions against the baselines (nfs-tmpfs, tmpfs, rdma); RDMA does not support memory-mapping, so we instead read or write the data using one-sided RDMA verbs. We consider two variations of region. One variant (rr) performs raw operations without caching: we drop the cache after every operation, so that every operation must go over the network. The other variant (rr+) caches the most recently accessed page, so that consecutive operations on the same page access the cache, and writes to a page are buffered until the entire page is written. We also consider a variant of nfs-tmpfs that caches a page in the same way (nfs-tmpfs+).

Results. Figure 10 shows the results. For reads (left), we see that nfs-tmpfs and rr are flat from 64 bytes until 4K; this is because the file system operates at a page granularity, so it fetches an entire page even if the request needs fewer bytes. RDMA and tmpfs have the lowest latency, at 41% and 54% of rr's latency on 64 bytes, and 38% and 28% on 1MB. This is because rr suffers from overheads of page faults, 4KB-transfer granularity, and the file cache; tmpfs also incurs those overheads, but it compensates by avoiding the network latency. nfs-tmpfs is the worst due to higher network overheads. For writes (right), results are qualitatively similar; for rr and nfs-tmpfs, writes are slower than reads because the file system performs a read-modify-write operation, where it first reads the page before it writes it, requiring two network round trips. With RDMA, writes are faster than reads because RDMA writes complete as soon as they are posted on the PCIe bus at the remote host, whereas reads

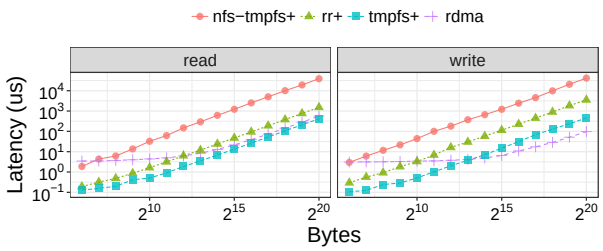


Figure 11: Latency for transferring data (caching).

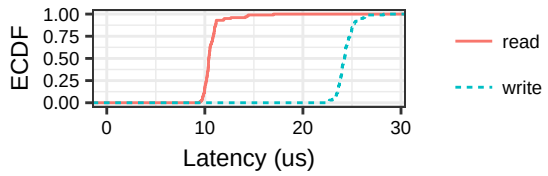


Figure 12: Distribution of latency for reading and writing 4 KB on a memory-mapped region using rr.

need to get data from memory.

These comparisons are unfair to file systems: unlike RDMA, they fetch full 4KB pages even for small requests, and cache them; doing so benefits applications that use the page later, but Figure 10 gives no credit for that. So, we now consider the effects of having a cache. Figure 11 shows the results with caching of the last page accessed. We see much improvement for small requests. Here, rr+ performs better than RDMA up to 4KB (reads) or 1KB (writes). We include RDMA in the graph for comparison, but it has no cache. Theoretically, an application can implement its own caching for RDMA, but doing so makes RDMA even more complex. In contrast, caching comes for free with a file system, without any application effort.

Figure 12 shows the cdf for reading or writing 4 KB of data using rr (no cache). We see a concentration from 10–11 us for reads, with the 95-percentile at 12 us; and a concentration from 23–26 us for writes, with the 95-percentile at 25.8 us. As we pointed out, writes are slower because the file system performs a read-modify-write operation (with memory-mapping, the system does not know that the application will eventually overwrite the entire page). This overhead is avoided by calling rmarkdirty (§5.8) prior to writing a page (not shown).

8.4 Performance of the file system

Setup. We run Sysbench, a standard file IO benchmark [59], to measure the performance of reading data from REGIONFS. We configure Sysbench with a single thread that reads from a 2GB file and reports throughput, average latency, and 95% latency. We study sequential and random reads of 16 KB chunks. We compare REGIONFS against nfs-tmpfs and local tmpfs.

System	Seq Tput (MB/s)	Seq LatAve (ms)	Seq Lat95 (ms)	Rnd Tput (MB/s)	Rnd LatAve (ms)	Rnd Lat95 (ms)
nfs-tmpfs	4871	0	0.01	4247	0	0.01
rr	5432	0	0.01	4821	0	0.01
tmpfs	6556	0	0	6048	0	0

Figure 13: Sysbench file IO benchmark results. Seq refers to sequential reads, Rnd to random reads.

Functionality	Description	regions loc	RDMA loc
Initialization	Code needed in every application	6	229
Producer-consumer	Simple message queue	29	103
Linked list	Traverse linked list	18	68
Hash table	Lookup operation of hash table	14	78
Access revocation	Remove access from a host	1	37

Figure 14: Equivalent functionality in regions and RDMA.

Results. Figure 13 shows the results. For throughput, tmpfs performs the best: 6.5 and 6.0 GB/s for sequential and random reads, respectively, while REGIONFS is within 83% and 80%—reasonably close to the in-memory performance of tmpfs, despite going to the network. Nfs-tmpfs is the worst at 4.9 and 4.2 GB/s. For latency, the resolution of the benchmark is 0.01ms, which is too large to reflect the difference between the different systems. (Please refer to the previous experiments, where we report other latency numbers.)

8.5 Code complexity

Setup. To study code complexity, we implement functionality that is commonly used in remote memory applications, and compare the number of lines of code (LOC) to implement them using REGIONFS and RDMA verbs.

Results. Figure 14 shows the complexity results. We see that region code has much fewer lines of code – 4.2 times on average, excluding initialization and revocation. For initialization, region requires just opening and memory mapping a file, while RDMA requires initializing contexts, memory registration, establishing connections, creating, transitioning and initializing queue pairs, key exchange, and more. For the other functionality, regions are similarly simpler, requiring just memory or file operations, while RDMA code must manually submit requests to queue pairs, monitor for completions, etc. In addition, RDMA verbs require explicit management of a partitioned global address space, which translates to more work at the application level. This complexity makes it hard for new developers to even get started on RDMA.

Next, we further study complexity, by using regions to adapt two applications to use remote memory.

8.6 Application: R

R is a statistical processing system for data in memory. Using regions, we adapt R to use remote memory. R

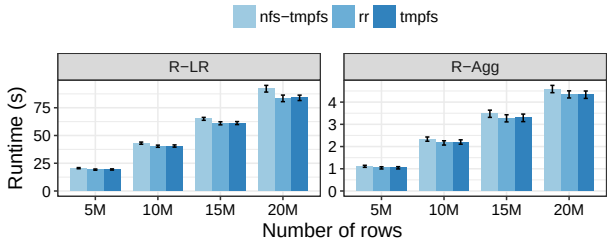


Figure 15: R application. Bars show total runtime (smaller is better). Error bars show std deviation.

has a large number of packages to extend its core functionality, including a package *ff* that extends R's memory capacity using memory-mapped files. ff provides objects that are each stored in a file; to limit memory consumption, small parts of the file called *sections* get memory mapped and unmapped as needed, with at most one section per file mapped at a time. We set up ff to use the files in REGIONFS, and to use sections that are 128KB wide.

Setup. In each experiment, we choose a workload for R and an input size. We use R to process the workload with an input of the given size, and we measure the time it takes. The workload mimics a data analyst that has a large data set and uses R to analyze parts of it. The data set is a large matrix stored in a host different from the one running R, representing data generated elsewhere in the network that does not fit in R's memory. The matrix is stored as several ff objects, one per column, with 200 columns and a number of rows that varies from 5 to 20 million. We consider two workloads:

- *R-Agg.* Compute an aggregation (mean) over ten columns of the matrix. This workload represents an extreme in terms of the ratio of computation to memory accesses: it almost entirely performs memory accesses, by reading data and only computing a sum.
- *R-LR.* Compute a linear regression over ten columns of the matrix. The algorithm accesses the rows of the matrix several times, but performs significant more computation than R-Agg, representing a balance between memory accesses and compute.

We consider three systems: rr, tmpfs, and nfs-tmpfs.

Results. Figure 15 (right) shows the R-Agg workload. Regions approach tmpfs within 1%, despite having to read the input from a remote host. In comparison, nfs-tmpfs is within 6% of tmpfs.

For the R-LR workload (Figure 15 left), we see a similar trend but with a larger running time incurred by linear regression. Regions are again within 1% of tmpfs, while nfs-tmpfs is within 9% of tmpfs.

While the performances of tmpfs and nfs-tmpfs are similar to rr, tmpfs and nfs-tmpfs do not permit R to run with a large memory because their capacity is limited by the available memory locally or in the nfs server. By contrast, regions can be multihosted (§5.4) to aggregate the memory of many machines.

We also ran R and placed the ff-generated files in an SSD rather than in REGIONFS, as a way to obtain more space. The running time of R increased by 2.5× (for R-agg) and 2.7× (for R-LR) relative to rr.

As for code complexity, we made no changes to R's ff package; we just set it up to use files in REGIONFS.

8.7 Applications: Metis

Metis is an in-memory map-reduce processing framework. Metis reads its initial input from a file, and launches many threads to run map-reduce. In map-reduce, the data is partitioned across a set of mappers, each producing an output; the outputs of all mappers are grouped based on a key, and the groups are partitioned across the reducers; each reducer produces some output, and all outputs are aggregated in the end. Metis runs on a single host, using work queues to distribute map and reduce tasks among many threads.

We modify Metis to run across many hosts, using regions to share its input and output. More specifically, the modified Metis does three things: (1) reads the initial input from remote hosts using regions, representing data produced by another computation, such as a previous map-reduce job[1], (2) runs threads across many hosts, with each host writing the output to a region to make it available to the other hosts, and (3) collects the regions with the results from all the hosts and aggregates the output. In effect, we produce a distributed version of Metis, while retaining its in-memory processing.

Setup. In each experiment, we run a map-reduce job to produce a histogram. In this job, each mapper processes a partition of the input and produces a partial histogram; the reducers then aggregate the bins, each reducer responsible for a disjoint set of bins; a final stage collects the bins from the reducers.

We consider two systems: the original Metis and our distributed version. We vary the number of threads in each system; for the distributed version, we vary the number of hosts. We measure the time to run the map-reduce computation. The input is a 2.6 GB image file, and the output produces 403 bins. Metis has an option called *prefault* to initially preload all input to memory.

Results. Figure 16 (left) shows the results for a single host as we increase the number of threads. We see that rr is faster because it reads the input from remote memory, while Metis reads from an SSD. For one host and one thread, rr is 2.0× faster than Metis; for 4 threads, it is 3.5× faster. By increasing the number of threads to 4,

[1]The motivation is that many map-reduce applications run a chain of map-reduce jobs, each consuming the output of the previous job.

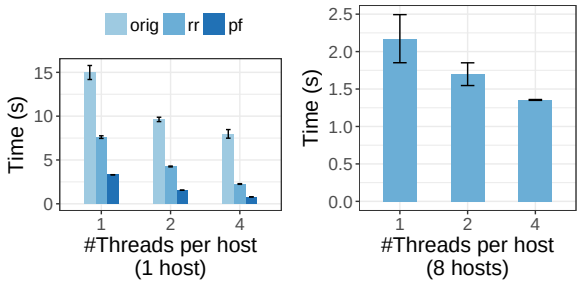


Figure 16: Metis application performance. On the left, there is a single host; *rr* refers to our distributed version of Metis, while *orig* and *pf* refer to the original Metis, where *pf* enables Metis's prefault option. On the right, we show our distributed version of Metis on 8 hosts (there are no bars for the original Metis since it runs only on 1 host). The y-axis is running time (smaller is better). Error bars show std deviation.

Metis and rr improve by 1.91× and 3.4×. If we exclude reading the input (with prefault), performance improves by 2.3–2.9× compared to rr.

Figure 16 (right) shows the results for our distributed version of Metis running on eight hosts and 1–4 threads per host. We see that performance improves compared to running with one host. Using 8 hosts, rr is 3.5, 2.5, 1.7× faster for 1, 2, 4 threads than rr using a single host with the same number of threads. The improvement comes from the hosts running the computation in parallel. The scalability is not linear because the running time is only a few seconds and so the overhead of synchronizing the hosts between phases is relatively high.

As for code complexity, the modifications to Metis consist of 82 lines. This is small, as the changes amount to changing a centralized system into a distributed one.

9 Conclusion

In this paper, we applied the Unix idea that "everything is a file" to remote memory, obtaining an abstraction in which a process exports parts of its memory as a file that remote processes can access. We studied the design behind this abstraction, described a prototype that achieves reasonable performance, and showed that applications can easily benefit from it.

References

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *ACM Symposium on Cloud Computing*, pages 121–127, Sept. 2017.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] Apache Hadoop. https://hadoop.apache.org/, July 2008.

[4] K. Asanovic and D. Patterson. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Keynote USENIX Conference on File and Storage Technologies*, Feb. 2014.

[5] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[6] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *International Conference on Management of Data*, pages 1463–1475, May 2015.

[7] T. Bedeir. Building an RDMA-capable application with IB verbs. RDMA read and write with IB verbs. https://thegeekinthecorner.wordpress.com/2013/02/02/rdma-tutorial-pdfs, Aug. 2010.

[8] C. G. Bell and I. Nassi. Revisiting scalable coherent shared memory. *IEEE Computer*, 51(1):40–49, Jan. 2018.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[10] N. Carriero and D. Gelernter. The S/Net's Linda kernel (extended abstract). In *ACM Symposium on Operating Systems Principles*, page 160, Dec. 1985.

[11] Ceph file system. https://ceph.com/ceph-storage/file-system.

[12] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Usenix Summer 1990 Technical Conference*, pages 127–136, June 1990.

[13] Direct access file system (DAFS) protocol. http://www.dafscollaborative.org.

[14] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation*, pages 401–414, Apr. 2014.

[15] A. Dragojević, D. Narayanan, E. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles*, pages 54–70, Oct. 2015.

[16] U. Drepper. How to write shared libraries. https://www.akkadia.org/drepper/dsohowto.pdf.

[17] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *International Parallel Processing Symposium*, pages 153–159, Apr. 1999.

[18] Dynamically connected transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf, 2014.

[19] Executable and linkable format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[20] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems*, pages 17–17, May 2015.

[21] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *ACM Symposium on Operating Systems Principles*, pages 201–212, Dec. 1995.

[22] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4):281–293, Oct. 1999.

[23] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation*, pages 249–264, Oct. 2016.

[24] Gen-Z draft core specification—december 2016. http://genzconsortium.org/draft-core-specification-december-2016.

[25] GlusterFS documentation. http://docs.gluster.org/en/latest.

[26] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *Symposium on Networked Systems Design and Implementation*, pages 649–667, Mar. 2017.

[27] Y. C. Hu, W. Yu, A. Cox, D. Wallach, and W. Zwaenepoel. Runtime support for distributed sharing in safe languages. *ACM Transactions on Computer Systems*, 21(1):1–35, Feb. 2003.

[28] Intel Omni-Path. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[29] iWARP. https://en.wikipedia.org/wiki/IWARP.

[30] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 295–306, Aug. 2014.

[31] S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *IEEE International Symposium on High Performance Distributed Computing*, pages 301–308, Aug. 1999.

[32] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[33] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[34] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over InfiniBand: An approach using a high performance network block device. In *IEEE International Conference on Cluster Computing*, pages 1–10, Sept. 2005.

[35] libibverbs. http://www.rdmamojo.com/2012/05/18/libibverbs.

[36] E.-J. Lim, S.-Y. Ahn, Y.-H. Kim, G.-I. Cha, and W. Choi. Design of cache backend using remote memory for network file system. In *International Conference on High Performance Computing and Simulation*, pages 864–869, July 2017.

[37] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture*, pages 267–278, June 2009.

[38] Linux man page for memfd_create (2).

[39] X. Lu, N. S. Islam, M. Wasi-ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop RPC with RDMA over infiniband. In *IEEE International Conference on Parallel Processing*, pages 641–650, Oct. 2013.

[40] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference*, pages 773–785, July 2017.

[41] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-2010-020, MIT, May 2010.

[42] D. Mazières. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, pages 261–274, June 2001.

[43] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, June 2013.

[44] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. Technical Report CMU-CS-94-213, Carnegie Mellon University, Nov. 1994.

[45] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *Symposium on Networked Systems Design and Implementation*, pages 17–33, Mar. 2017.

[46] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference*, pages 291–305, July 2015.

[47] OpenCAPI consortium. http://opencapi.org.

[48] The R project for statistical computing. https://www.r-project.org.

[49] RDMA over converged ethernet. https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet.

[50] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proceedings of the VLDB Endowment*, 9(4):228–239, Dec. 2015.

[51] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Oct. 1996.

[52] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[53] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, Oct. 1994.

[54] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *ACM Symposium on Cloud Computing*, pages 323–337, Sept. 2017.

[55] K. A. Shutemov. [patchv4 9/9] x86/mm: Allow to have userspace mappings above 47-bits. http://www.spinics.net/lists/linux-mm/msg125594.html.

[56] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *ACM Symposium on Operating Systems Principles*, pages 170–183, Oct. 1997.

[57] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Engineering Bulletin*, 40(1):38–49, Mar. 2017.

[58] M. M. Swift. Towards O(1) memory. In *Workshop on Hot Topics in Operating Systems*, pages 7–11, May 2017.

[59] sysbench. https://github.com/akopytov/sysbench.

[60] T. Talpey and C. Juszczak. Network file system (NFS) remote direct memory access (RDMA) problem statement. RFC 5532, RFC Editor, May 2009.

[61] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *ACM Symposium on Operating Systems Principles*, pages 306–324, Oct. 2017.

[62] Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt.

[63] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems, Nov. 1994.

[64] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, Feb. 2017.