FileMR: Rethinking RDMA Networking for Scalable Persistent Memory

Jian Yang, *UC San Diego;* Joseph Izraelevitz, *University of Colorado, Boulder;* Steven Swanson, *UC San Diego*

https://www.usenix.org/conference/nsdi20/presentation/yang

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

FileMR: Rethinking RDMA Networking for Scalable Persistent Memory

Jian Yang*
UC San Diego

Joseph Izraelevitz
University of Colorado, Boulder

Steven Swanson
UC San Diego

Abstract

The emergence of dense, byte-addressable *nonvolatile main memories* (NVMMs) allows application developers to combine storage and memory into a single layer. With NVMMs, servers can equip terabytes of memory that survive power outages, and all of this persistent capacity can be managed through a specialized NVMM file system. NVMMs appear to mesh perfectly with another popular technology, remote direct memory access (RDMA). RDMA gives a client direct access to memory on a remote machine and mediates this access through a memory region abstraction that handles the necessary translations and permissions.

NVMM and RDMA seem eminently compatible: by combining them, we should be able to build network-attached, byte-addressable, persistent storage. Unfortunately, however, the systems were not designed to work together. An NVMM-aware file system manages persistent memory as files, whereas RDMA uses a different abstraction — memory regions to organize remotely accessible memory. As a result, in practice, building RDMA-accessible NVMMs requires expensive translation layers resulting from this duplication of effort that spans permissions, naming, and address translation.

This work introduces two changes to the existing RDMA protocol: file memory region (FileMR) and range-based address translation. These optimizations create an abstraction that combines memory regions and files: a client can directly access a file backed by NVMM file system through RDMA, addressing its contents via file offsets. By eliminating redundant translations, it minimizes the amount of translations done at the NIC, reduces the load on the NIC's translation cache and increases the hit rate by 3.8× - 340× and resulting in application performance improvement by 1.8× - 2.0×.

1 Introduction

How scalable computer systems store and access data is changing rapidly, and these changes are in part motivated by the blurring of lines between traditionally separate system components. Nonvolatile main memory (NVMM) provides byte-addressable memory that survives power outages, blurring the line between memory and storage. Similarly, remote direct memory access (RDMA) allows a client to directly access memory on a remote server, blurring the line between local and remote memory. At first glance, by combining NVMM and RDMA, we could unify storage, memory and network to provide large, stable, byte-addressable network-attached memory. Unfortunately, the existing systems used to manage these technologies are simultaneously overlapping and incompatible.

NVMMs merge memory and storage. The technology allows applications to access persistent data using load/store instructions, avoiding the need for a block-based interface utilized by traditional storage systems. NVMMs are managed by an NVMM-aware file system, which mediates access to the storage media. With an NVMM-aware file system, applications can map a file into their address space, and then access it using loads and stores instructions, drastically reducing the latency for access to persistent data.

RDMA merges local and remote memory. RDMA allows a client to directly access memory on a remote server. Once the remote server decides to allow incoming access, it registers a portion of its address space as an RDMA *memory region* and sends the client a key to access it. Using the key, the client can enlist the server's RDMA network interface (RNIC) to directly read and write to the server's memory, bypassing the CPU. RDMA is popular as it offloads most of the networking stack onto hardware and provides close-to-hardware abstractions, exhibiting much better latency compared to TCP/IP protocol.

Ideally, we could combine NVMM and RDMA into a unified network-attached persistent memory. Unfortunately, NVM file systems and the RDMA network protocol were not designed to work together. As a result, there are many redundancies, particularly where the systems overlap in memory. Only RDMA provides network data transfer and only the NVMM file system provides persistent memory metadata, but both systems implement protection, address translation, naming, and allocation across different abstractions: for RDMA, memory regions, and for NVMM file systems, files. Naively using RDMA and NVMM file systems together results in a duplication of effort and inefficient translation layers between their abstractions. These translation layers are expensive, especially since RNICs can only store translations for limited amount of memory while NVM capacity can be extremely large.

In this paper, we present a new abstraction, called a *file*

*Now at Google

memory region (FileMR), that combines the best of both RDMA and NVM file systems to provide fast, network-attached, file-system managed, persistent memory. It accomplishes this goal by offloading most RDMA-required tasks related to memory management to the NVM file system through the new memory region type; the file system effectively becomes RDMA's control plane.

With the FileMR abstraction, a client establishes an RDMA connection backed by *files*, instead of memory address ranges (i.e., an RDMA memory region). RDMA reads and writes are directed to the file through the file system, and addressed by the file offset. The translation between file offset and physical memory address is routed through the NVMM file system, which stores all its files in persistent memory. Access to the file is mediated via traditional file system protections (e.g., access control lists). To further optimize address translation, we integrate a *range-based translation* system, which uses address ranges (instead of pages) for translation, into the RNIC, reducing the space needed for translation and resolving the abstraction mismatch between RDMA and NVMM file systems.

Our FileMR design with range-based translation provides a way to seamlessly combine RDMA and NVMM. Compared to simply layering traditional RDMA memory regions on top of NVMM, FileMR provides the following benefits:

- It minimizes the amount of translation done at the NIC, reducing the load on the NIC's translation cache and improving hit rate by 3.8× - 340×.
- It simplifies memory protection by using existing file access control lists instead of RDMA's ad-hoc memory keys.
- It simplifies connection management by using persistent files instead of ephemeral memory region IDs.
- It allows network-accessible memory to be moved or expanded without revoking permissions or closing a connection, giving the file system the ability to defragment and append to files.

The rest of this paper is organized as follows. Section 2 describes the necessary background on RDMA and NVMM file systems. Section 3 describes the design of the FileMR. Section 4 describes our proposed changes to RDMA stack and RNICs, and Section 5 introduce two case studies. Section 6 provides experimental results. Section 7 discusses the applicability of the FileMR on real hardware. Section 8 describes related work, and Section 9 concludes.

2 Background

This section introduces background on both RDMA and NVMM and describes the motivation for introducing a new memory abstraction for RDMA, detailing the issue of redundant memory management mechanisms and the reasons existing systems cannot solve this problem.

2.1 RDMA Networking

RDMA has become a popular networking protocol, especially for distributed applications [2, 20–22, 34, 36, 43, 47, 55, 56, 62]. RDMA exposes a machine's memory to direct access from the RDMA network interface (RNIC), allowing remote clients to directly access a machine's memory without involving the local CPU.

The RDMA hardware supports a set of operations (called *verbs*). *One-sided* verbs, for instance, "read" and "write", directly access remote memory without requiring anything of the remote CPU, in fact, these verb bypasses the remote CPU entirely. *Two-sided* verbs, in contrast, require both machines to post matching requests, for instance, "send" and "receive", which transfer data between registered buffers with addresses chosen by sender and receiver applications locally.

To establish an RDMA connection, an application registers one or more *memory regions* (*MRs*) that grant the local RNIC access to part of the local address space. The MR functions as both a name and a security domain: To give a client access to a region, the local RNIC supplies the MR's virtual address, size and a special 32-bit "rkey". Rkeys are sent with any one-sided verb and allow the receiving RNIC to verify the client has direct access to the region. For two-sided verbs, a send/recv operation requires both the sender and receiver to post matching requests, each attached to some local, pre-registered, memory region, negating the need for rkeys.

To manage outstanding requests, RDMA uses *work queues* derived from the virtual interface architecture (VIA) [10]. After establishing a connection, an application can initiate an RDMA verb through its local RNIC by posting work queue entries (WQEs). These entries are written onto a pair of queues (a queue pair or "QP"); one queue for send/write requests and one for read/receive requests. Once the entry is written to the queue pair, the RNIC will execute the RDMA verb and access the remote machine. Once the verb is completed, the RNIC will acknowledge the verb's success by placing a "completion" in the "completion queue" (CQ). The application can poll for the completion from the completion queue to receive notification that the verb completed successfully.

2.2 Nonvolatile Main Memory

Nonvolatile main memory (*NVMM*) is nonvolatile memory directly accessible via a load/store interface. NVMM is comprised of multiple nonvolatile DIMMs that are attached to the CPU memory bus and sit alongside traditional DRAM DIMMs. One or multiple nonvolatile DIMMs can be combined to form a single contiguous physical address space exposed to the OS [42].

As NVMM is a persistent media, it requires management software to provide naming, allocation and protection, and it is generally managed by a file system. However, unlike traditional file systems built for slower block devices, NVMM-aware file systems play a critical role in providing efficient NVMM access — the DRAM-comparable latency of NVMM means software overhead can dominate performance. As a result, NVMM-aware file systems [7,57,59,60] avoid software overhead along the critical path in two ways:

First, they support the direct access `mmap()` (*DAX-mmap*) capability. DAX-mmap allows applications to map NVMM files directly into their address spaces and to perform data accesses via simple loads and stores. This scheme allows applications to bypass the kernel and file system for most data accesses, drastically improving performance for file access.

However, NVMM resides within the memory hierarchy, which can cause complications since caches are not persistent but can hold data that the application wants to persist. To persist data, cached writes to NVMM must be followed by cache-line flush or clean instructions to ensure the data is actually written back to NVMM, and non-temporal writes can bypass the CPU caches entirely. A store fence can enforce the ordering of the writes and guarantee the data will survive a power failure.

2.3 Managing RDMA and NVMM

Userspace RDMA accesses and NVMM mmapped-DAX accesses share a critical functionality: they allow direct access to memory without involving the kernel. Broadly speaking, we can divide both NVMM file systems and RDMA into a data plane that accesses the memory and a control plane that manages the memory exposed to user applications. The data plane is effectively the same for both: it consists of direct loads and stores to memory. The control plane, in contrast, differs drastically between the systems.

For both RDMA and NVMM file systems, the control plane must provide four services for memory management. First, it must provide naming to ensure that the application can find the appropriate region of memory to directly access. Secondly, it must provide access control, to prevent an application from accessing data it should not. Thirdly, it must provide a mechanism to allocate and free resources to expand or shrink the memory available to the application. Finally, it must perform translation between application level names (i.e., virtual addresses, or memory and file offsets) to physical memory addresses. In practice, this final requirement means that both RDMA and NVMM file systems must work closely with the virtual memory subsystem.

Table 1 summarizes the control plane metadata operations for RDMA and NVMM. These memory management functionalities are attached to different abstractions in RDMA and NVMM file systems. For RDMA we use abstractions such as memory regions and memory windows, and for NVMM file systems we use files.

Role	RDMA / File System	FileMR
Naming	Both (Redundant)	FS Managed
Permissions	Both (Redundant)	FS Managed
Allocation	Both (Redundant)	FS Managed
Appending	Not Allowed	FS Managed
Remapping	Not Allowed	FS Managed
Defragmentation	Not Allowed	FS Managed
Translation	Both (Incompatible)	FS Managed
Persistence	FS Only	FS Managed
Networking	RDMA	RDMA
CPU-Bypass	RDMA	RDMA

Table 1: **Control plane roles for RDMA and NVMM.** This table shows the features provided by RDMA and NVMM vs. FileMR.

2.3.1 Naming

Names provide a hardware-independent way to refer to physical memory locations. In RDMA applications, the virtual address of a memory region, along with its "host" machine's location (e.g., IP address or GID) serves as a globally (i.e., across nodes) meaningful name for regions of physical memory. These names are transient, since they become invalid when the application that created them exits, and inflexible since they prevent an RDMA-exposed page from changing its virtual to physical address mapping while accessible. To share a name with a client that wishes to directly access it via reads and writes, the host gives it the metadata of the MR. For two-sided verbs (i.e., send/receive) naming is ad-hoc: the receiver must use an out-of-band channel to decide where to place the received data.

NVMM-based file systems use filenames to name regions of physical memory on a host. Since files outlive applications, the file system manages names independent of applications and provides more sophisticated management for named memory regions (i.e., hierarchical directories and text-based names). To access a file, clients and applications on the host request access from the file system.

2.3.2 Permissions

Permissions determine what processes have access to what memory. In RDMA, the RDMA contexts are isolated and permissions are enforced in two ways. To grant a client direct read/write access to a memory location, the host shares a memory region specific "rkey." The rkey is a 32-bit key that is attached to all one-sided verbs (such as read and write) and is verified by the RNIC to ensure the client has access to the addressed memory region. For every registered region, the RNIC driver maintains the rkey, along with other RDMA metadata that provides isolation and protection in hardware-accessible structures in DRAM.

Permissions are established when the RDMA connection

between nodes is created, and are granted by the application code establishing the connection. They do not outlive the process or survive a system restart. For two-sided verbs protection is enforced by the receiving application in an ad-hoc manner: The receiver uses an out-of-band channel to decide what permissions the sender has.

Access control for NVMM uses the traditional file system design. Permissions are attached to each file and designated for both individual users and groups. Unlike RDMA memory regions and their rkeys, permissions are a property of the underlying data and survive both process and system restart.

2.3.3 Allocation

RDMA verbs and NVMM files both directly access memory, so allocation and expansion of available memory is an important metadata operation.

For NVMM file systems, the file system maintains a list of free physical pages that can be used to create or extend files. Creation of a file involves marshalling the appropriate resources and linking the new pages into the existing file hierarchy. Similarly, free pages can be linked to or detached from existing files to grow or shrink the file. Changing the size of DAX-mmap'd files is easy as well with calls to `fallocate` and `mremap`.

Creating a new RDMA memory region consists of allocating the required memory resources, pinning their pages, and generating the rkey. Note that although many RNICs are capable of handling physical addresses [32], the physical address of a memory region is often out of the programmer's control (it depends, instead, on the implementation of `malloc`), and the page is pinned once the region is registered, leading to a fragmented physical address space.

In addition, changing the mapping of a memory region is expensive. For example, to increase the memory region size, the host server needs to deregister the memory region, reregister a larger region, and send the changes to any interested clients. The `rereg_mr` verb combines deregistration and registration but still carries significant overhead. MPI applications with public memory pool often use memory windows to provide dynamic access control on top of a memory region. This approach does not blend with NVMM file systems since it still requires static mappings of the underlying memory region.

Alternatively, programmers can add another memory region to the connection or protection domain. However, as memory regions require non-negligible metadata and RDMA does not support multi-region accesses, this solution adds significant complexity.

This fixed size limitation also prohibits common file system operations and optimizations, such as appending to a file, remapping file content, and defragmentation.

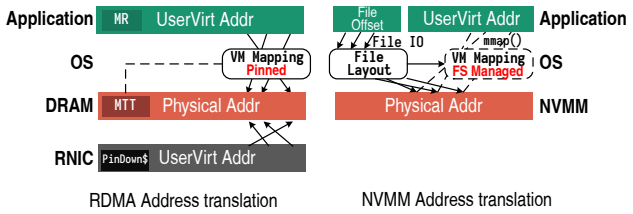

Figure 1: **Address translation for RDMA and NVMM.** RDMA (left) uses NIC-side address translation with pinning, while NVMM (right) allows the file system to maintain the layout of a file mapped to user address space.

2.3.4 Address Translation

RDMA and NVMM file system address translation mechanisms ensure that their direct accesses hit the correct physical page.

As shown in Figure 1, RDMA solves the problem of address translation by *pinning* the virtual to physical address, that is, as long as a memory region is registered, its virtual and physical addresses cannot change. Once this mapping is fixed, the RNIC is capable of handling memory regions registered on virtual address ranges directly: the RNIC translates from virtual addresses to physical addresses for incoming RDMA verbs. To do this translation, the NIC maintains a *memory translation table* (MTT) that holds parts of the system page tables.

The MTT flattens the translation entries for the relevant RDMA accessible pages and can be cached in the RNIC's onboard SRAM [54] to accelerate lookups of this mapping. The pin-down cache is critical for getting good performance out of RDMA — the pin-down cache is small (a few megabytes), a miss is expensive, and due to its addressing mechanism, most RNICs require all pages in a region be the same size. To circumvent these limitations, researchers have done significant work trying to make the most of the cache for addressing large memories [14, 22, 35, 36, 43, 48, 56, 62]. While complex solutions exist, the most common recommendation is to reduce the number of translations needed (e.g., addressing large contiguous memory regions with either huge pages or physical addresses).

The NVMM file system handles address translation in two ways, both different from RDMA. For regular reads and writes, the file system translates file names with offsets to physical addresses; this translation is done in the kernel during the system call. For memory mapped accesses, `mmap` establishes a virtual to physical address mapping from userspace directly to the file's contents in NVMM, loading the mapping into the page table. The file system only interferes on the page fault handling when a translation is missing between the user and physical addresses (i.e., a soft page fault); the file system is bypassed on normal data accesses.

The different translation schemes interfere with each other to create performance problems. If a page is accessible via

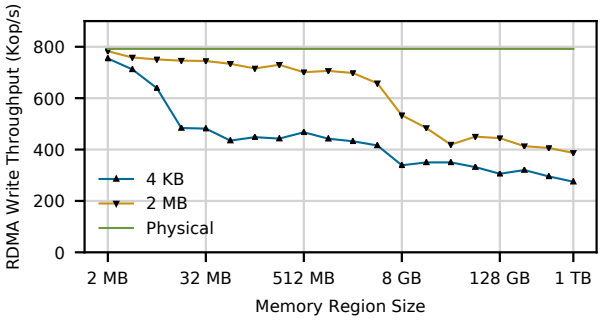

Figure 2: **RDMA Write performance over different memory region sizes.** This figure shows the throughput of 8-byte RDMA writes affected by the pin-down cache misses. Data measured on Intel Optane DC Persistent Memory with an Mellanox CX-3 RNIC.

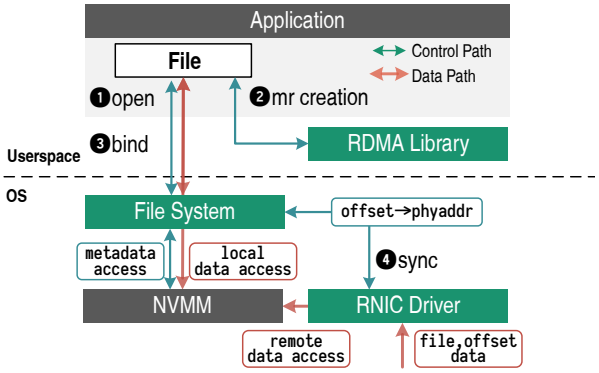

Figure 3: **FileMR: Control path and data path.** The user application communicates with the RDMA libraries and file system in control path, and access local and remote NVMM directly in datapath.

RDMA, it is pinned to a particular physical address, and furthermore, every page within the region must be the same size. Therefore the file system is unable to update the layout of the open file (e.g., to defragment or grow the file). As RDMA impedes defragmentation of files and prohibits mixing page sizes in RDMA accessible memory, memory regions backed by files must use many small pages to address large regions, overwhelming the pin-down cache and decimating RDMA performance.

Figure 2 shows the impact of pin-down cache misses on RDMA write throughput. Each work request writes 8 bytes to a random 8-byte aligned offset. When the memory region size is 16 MB, using 4 kB achieves 61.1% of the baseline (sending physical addresses, no TLB or pin-down cache misses) performance compared to 95.2% when using 2 MB hugepages. When the region size hits 16 GB, even 2 MB pages is not sufficient — achieving only 61.2% performance.

3 Design

FileMR is a new type of memory region that extends the existing RDMA protocol to provide file-based abstractions for NVMM. It requires minor changes to existing RDMA protocol and does not rely on any specific design of the file systems. FileMR can coexist with conventional RDMA memory regions, ensuring backward compatibility.

As shown in Table 1, the FileMR resolves the conflicts between RDMA and NVMM file systems that cause unnecessary restrictions and performance degradation through several innovations.

- **Merged control plane:** With an RDMA FileMR, a client uses a *file offset* to address memory, instead of a virtual or physical address. The FileMR also leverages the naming, addressing, and permissions of the file system to streamline RDMA access.
- **Range-based address translation:** The FileMR leverages the file system's efficient, extent-based layout description mechanism to reduce the amount of states the NIC must hold. As files are already organized in continuous extents, we extend this addressing mechanism to the RNIC, allowing the RNIC's pin-down cache to use a space efficient translation scheme to address large amounts of RDMA accessible memory.

The rest of this section continues as follows. We begin by describing the assumptions and definitions of FileMR, followed by the core mechanisms. Then, we describe the system architecture required to support our new abstraction.

3.1 Assumptions and Definitions

FileMR acts as an efficient and coordinated memory management layer across the userspace application, the system software, and the RDMA networking stack. This paper assumes the NVMM is actively managed by system software, and we describe it as a *file system*. Note that the concept of a file system is loosely defined: FileMR can be integrated with a kernel file system, a userspace file system, or a userspace NVMM library that accesses raw NVMM (also known as device-DAX) and provides naming, where a *file* maps to a corresponding entity.

This paper assumes NVMM is mapped to application address space in its entire lifecycle: As described in Section 2.2, the most prominent feature of NVMM is to have fine-grained persistency at a very low cost [63]. The design goal of FileMR is to enable remote NVMM accesses while retaining the simplicity and efficiency of local NVMM accesses. An alternative approach is to build holistic systems that manage both storage (NVMM) and networking (RDMA), these related work will be discussed in Section 8.2.

3.2 FileMR

Our new abstraction, the FileMR, is an RDMA memory region that is also an NVMM file. This allows the RDMA and NVMM control planes to interoperate smoothly. RDMA accesses to the FileMR are addressed by file offset, and the file system manages the underlying file's access permissions, naming, and allocation as it would any file. NVMM files are always backed by physical pages managed by the file system, so, when using a FileMR, the RDMA subsystem can simply reuse the translation, permission, and naming information already available in the file system metadata for the appropriate checks and addressing.

Figure 3 shows an overview of metadata and data access with FileMR. For metadata, before creating a FileMR, the application opens the backing file with the appropriate permissions (step ❶). Next, the application creates the FileMR (step ❷) and *binds* (step ❸) the region to a file, which completes the region's initialization. Binding the FileMR to the file produces a *filekey*, analogous to an rkey, that remote clients can use to access the FileMR. Once the FileMR is created and bound to a backing file, the file system will keep the file's addressing information in sync with the RNIC (step ❹).

For data access to a remote FileMR and its backing NVMM file, applications use the FileMR (with the filekey to prove its permissions) and a *file offset*. The RNIC translates between file offsets and physical addresses using translation information provided by the file system. In addition to one-sided read and write verbs to the FileMR, we introduce a new one-sided append verb that grows the region. When sending the append verb, the client does not include the remote address, and the server handles it like an one-sided write with address equal to current size of FileMR. It then updates the FileMR size and notifies the file system. As an optimization, to prevent faulting on every append message, the file system can preallocate translation entries beyond the size of a file. Even while the backing file is opened and accessible via a FileMR, local applications can continue to access it using normal file system calls or mmapped addresses — any change to the file metadata will be propagated to the RNIC.

3.3 Range-based Address Translation

NVMM file systems try to store file data in large, linear extents in NVMM. FileMR uses *range-based address translation* within the MTT and pin-down cache through a *RangeMTT* and *range pin-down* cache, respectively. This change is a significant departure from traditional RDMA page-based addressing. Unlike page-based translations, which translate a virtual to physical address using sets of fixed size pages, range-based translation (explored and used in CPU-side translation [5, 11, 23, 38]) maps a variably sized virtual address range to physical address. Range-based address translation is useful when addressing large linear memory regions (which NVMM file systems strive to create) and neatly leverages the preexisting extent-based file organization.

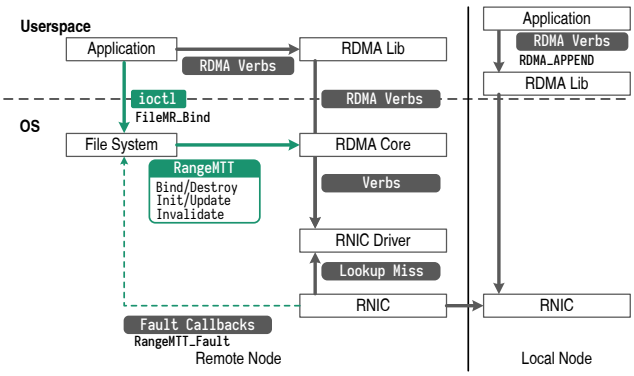

Figure 4: **Overview of FileMR components.** Implementing FileMR requires changes in file system, RDMA stack and hardware.

For the FileMR, range-based address translation has two major benefits: both the space required to store the mapping and the time to register a mapping scale with the number of variable-sized extents rather than with the number of fixed size pages. Registering a page in the MTT and pin-down cache takes about 5 μs, this process requires locking memory descriptors and is hard to parallelize. As a result, a single core can only register memory at 770 MB/s with 4 kB pages. For NVMMs on the order of terabytes, the result registration time will be unacceptably long.

3.4 Design Overview

The implementation of the FileMR RDMA extension requires coordination and changes across several system components: the file system, the RNIC, the core RDMA stack, and the application. Figure 4 shows the vanilla RDMA stack (in grey) along with the necessary changes to adopt FileMR (in green).

To support the FileMR abstraction, the file system is required to implement the `bind()` function to associate a FileMR and a file, and, when necessary, notify the RDMA stack (and eventually the RNIC's RangeMTT and pin-down cache) when the bound file's metadata changes via callbacks (see Table 2). These callbacks allow the RNIC to maintain the correct range-based mappings to physical addresses for incoming RDMA requests.

Optionally, the file system can also register a set of callback functions triggered when RNIC cannot find a translation for an incoming address. This process is similar to on-demand paging [28, 29] and is required to support our new append verb, which both modifies the file layout and writes to it.

Supporting the FileMR abstraction also requires changes to the RNIC hardware. With our proposed RangeMTT, RNIC hardware and drivers would need to adopt range-based addressing within both the MTT and pin-down cache. Hardware range-based addressing schemes [5, 15, 23, 38] can be used to implement range-based address lookup. In our experiments

API	Description
`cm_bind()`	To notify RNIC of new bound file
`cm_init()`	To initialize RangeMTT entries
`cm_update()`	To update a RangeMTT entry
`cm_invalidate()`	To invalidate a RangeMTT entry
`cm_destroy()`	To destroy a file binding

Table 2: **File system to RNIC callbacks for FileMRs.** These callbacks are used by the file system to notify the RDMA stack and RNIC that file layouts (and consequently address mappings) have changed.

API	Description
`bind()`	Binds an opened file to FileMR
`ibv_reg_mr()`	Creates a FileMR with `FILEMR` flag
`ibv_post_send()`	Posts append w/ `APPEND` flag (`uverb`)
`ib_post_send()`	Posts append w/ `APPEND` flag (`kverb`)

Table 3: **New/changed RDMA methods.** These methods in the RDMA interface are new or have new flags under the FileMR system.

we simulate these changes using a software RNIC (see Section 4).

The FileMR also adds incremental, backwards compatible changes to the RDMA interface itself (see Table 3). It adds a new access flag for memory region creation to identify the creation of a FileMR. After its creation, the FileMR is marked as being in an unprovisioned state — the subsequent `bind()` call into the file system will allocate the FileMR's translation entries in the RangeMTT (via the `cm_bind` callback from the file system). The `bind()` method can be implemented with an `ioctl()` (for kernel-level file systems) or a library call (for user-level file systems). The FileMR also adds the new one-sided RDMA `append` verb. Converting existing applications to use FileMRs is easy as the applications only need to change its region creation code.

4 Implementation

We implemented the FileMR and RangeMTT for both the kernel space and userspace RDMA stack in Linux, and our implementations support the callbacks described in Table 2 and the changed methods in Table 3. The kernel implementation is based on Linux version 4.18, and userspace implementation is based on rdma-core (userspace) packages shipped with Ubuntu 18.04. Table 4 summarizes our implementation of FileMR with RangeMTT.

For our FileMR implementation on the NIC side, our implementation is based on a software-based RNIC: Software RDMA over Converged Ethernet (Soft-RoCE) [4, 25]. Soft-RoCE is a software RNIC built on top of ethernet's layer 2 and layer 3. It fully implements the ROCEv2 specification. Future research could work to build a FileMR compatible

	Item	Description
FileMR implementation on RDMA stack		
K	`ibcore`	Range-based TLB and FileMR kverbs
U	`libibverbs`	FileMR verbs in userspace
FileMR support on soft-RoCE		
K	`rxe`	Device driver and emu. RangeMTT.
U	`librxe`	Userspace driver
FileMR support for file system		
K	`nova`	a NVMM-aware file system
Applications adapting FileMR		
U	`novad`	Function stubs for remote file accesses
U	`libpmemlog`	NVMM log library

Table 4: **Summary of FileMR implementation.** This table shows the components modified to implement FileMR. The first column indicates the change is in kernel space (K) or userspace (U).

RNIC in real hardware.

To implement our RangeMTT, we followed the design introduced in Redundant Memory Mappings [23]: each FileMR points to a b-tree that stores offsets and lengths, and we use the offsets as indices. All RangeMTT entries are page-aligned addresses, since OS can only manage virtual memory in page granularity.

Unlike page-aligned RangeMTT, FileMR supports arbitrary sizes and allows sub-page files/objects. Each RangeMTT entry consists of a page address, a length field and necessary bits. These entries are non-overlapping and can have gaps for sparse files.

To support the `append` verb, the FileMR allows translation entries beyond its size. The append is one-sided but does not specify remote server addresses in the WR. On the server side, the RNIC always attempts to DMA to the current size of the FileMR and increases its size on success. When the translation is missing, the server can raise an IO page fault when IOMMU is available and a file system routine will be called to fulfill the faulty entries. Alternatively, if such support is unavailable, the server signals the client via a message similar to a receiver not ready (RNR) error.

Soft-RoCE manages the MTT entries as a flat array of 64-bit physical addresses with lookup complexity of O(1). We found similar design is implemented in hardware RNIC driver such as `mlx4`. For FileMR with a range pin-down cache miss, the entry lookup will traverse the registered data structures with higher time complexity (O(log(n))).

Soft-RoCE does not have a pin-down cache since the mappings are in DRAM. To emulate the RangeMTT, we built a 4096-entry 4-way associative cache to emulate the traditional pin-down cache, and a 4096-entry, 4-way associative range pin-down cache for FileMR. Each range translation entry consists of a 32 bit page address and a 32-bit length, which allows a maximal FileMR size of 16 TB (4 kB pages) or 8 PB (2 MB pages).

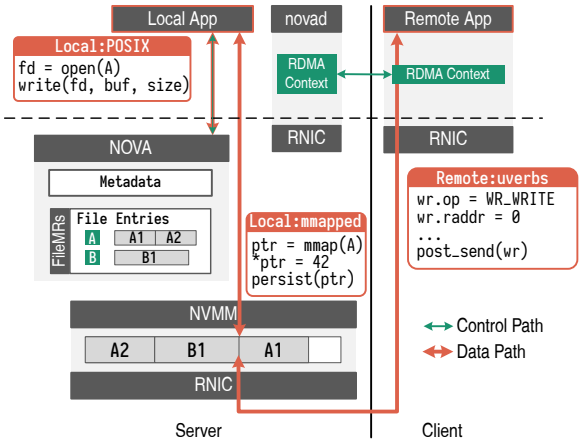

Figure 5: **Enabling remote NOVA accesses using FileMR.** Using FileMR, remote file accesses share a similar interface over RDMA as local NVMM accesses.

We adapted two applications to use FileMR. For a kernel file system, our implementation is based on NOVA [59], a full-fledged kernel space NVMM-aware file system with good performance. We also adapted the FileMR to libpmemlog, part of pmdk [40], a user-level library that manages local persistent objects, to build a remotely accessible persistent log.

5 Case Studies

In this section we demonstrate the utility of our design with our two case studies. In Section 5.1, we demonstrate how to use FileMR APIs to enable remote file accesses with consistent addressing for local and remote NVMM. In Section 5.2, we extend libpmemlog [40], a logging library designed for local persistent memory into a remotely accessible log, demonstrating how FileMR can be applied to userspace libraries.

5.1 Remote File Access in NOVA

In this section, we demonstrate an example usage of our FileMR by extending a local NVMM file system (NOVA [59]). By combining the NVMM file system, RDMA, and our new FileMR abstraction, we can support fast remote file accesses that entirely bypass the kernel.

NOVA is a log-structured POSIX-compliant local NVMM file system. In NOVA, each file is organized as a persistent log of variably sized extents, where the extents reside on persistent memory. The file data is allocated by the file system through per-cpu free lists and maintained as coalescing entries.

To handle metadata operations on the remote file system, we added an user-level daemon `novad`. The daemon opens the file to establish an FileMR, and receives any metadata updates (e.g. directory creation) from remote applications and applies them to the local file system.

On the client side, an application opens the file remotely by communicating with `novad` and receiving the filekeys. It can then send one-sided RDMA verbs to directly access remote NVMM. At the same time, applications running locally can still access the file with traditional POSIX IO interface, or map the file to its address space and issue loads and stores instructions.

Our combined system can also easily handle data replication. By using several FileMRs, we can simply duplicate a verb (with the same or different `filekeys` depending on the file system implementation) and send to multiple hosts, without considering the physical address of the files (so long as their names are equivalent).

5.2 Remote NVMM Log with libpmemlog

The FileMR abstraction only requires that the backing "file system" to appropriately implement the `bind()` method, RNIC callbacks, and have access to raw NVMM. For instance, a FileMR can be created by an application having access to the raw NVMM device. In this section, we leverage this flexibility and build a remote NVMM log based on `libpmemlog`.

We modify the allocator of `libpmemlog` to use the necessary FileMR callbacks — that is, whenever memory is allocated or freed for the log, the RNIC's RangeMTT is updated. The client appends to the log with the new append verb. On the server side, when the FileMR size is within the mapped RangeMTT, the RNIC can perform the translation while bypassing the server application. If not, a range fault occurs and the library expands the region by allocating and mapping additional memory.

6 Evaluation

In this section, we evaluate the performance of the FileMR. First, we measure control plane metrics such as registration cost, memory utilization of the FileMR, as well as the efficiency of RangeMTT. Then we evaluate application-level data plane performance from our two case studies and compare FileMR-based applications with existing systems.

6.1 Experimental Setup

We run our FileMR on servers configured to emulate persistent memory with DRAM. Each node has two Intel Xeon (Broadwell) CPUs with 10 cores and 256 GB of DRAM, with 64 GB configured as an emulated NVMM device. We setup Soft-RoCE on an Intel X710 10GbE NIC connected to a switch.

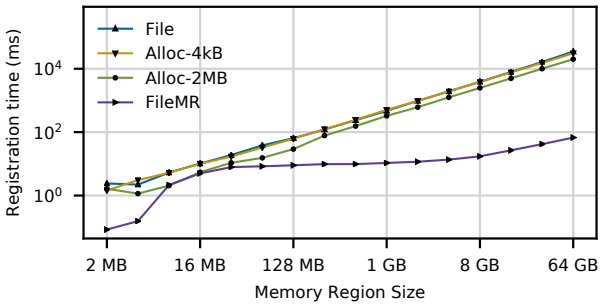

Figure 6: **FileMR registration time.** This figure shows the time consumed to register a fixed size memory region. The Y-axis is in log scale.

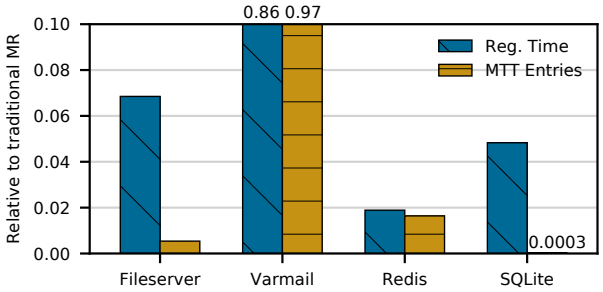

Figure 7: **FileMR on fragmented files.** Compared to traditional MRs, FileMR saves registration cost and RNIC translation entries.

Workload	# Files	Avg. Size	Description
Fileserver	7980	6.82 MB	File IOs
Varmail	4511	11.3 kB	Random IOs
Redis	2	561 MB	Write + Append
SQLite	1	109 MB	Write + Sync

Table 5: **Workload Characteristics.** Description of workloads to evaluate registration cost of FileMR and pin-down cache hit rate.

6.2 Registration Overhead

Allocated Regions We measure the time consumed in memory region registration using FileMRs versus conventional user-level memory regions backed by NOVA with 4 kB pages and anonymous buffers with 4 kB and 2 MB pages. This experiment demonstrates the use case when an application allocates and maps a file directly, without updating its metadata. For FileMR, we also include the time generating range entries from NOVA logs, which happens when an application opens the file for the first time.

As shown in Figure 6, registering a large size memory region consumes a non-trivial amount of time. It takes over 30 seconds to register a 64 GB persistent (**File**) and volatile (**Alloc-4K**) memory region with 4 kB pages. Using hugepages (**Alloc-2M**) reduces the registration cost to 20 seconds, while it only takes 67 ms for FileMR (three orders of magnitude lower). The FileMR registration time increases modestly as the region size grows mainly due to the internal fragmentation of the file system allocator.

For small files, NOVA only creates one or two extents for the file, while conventional MRs still interacts with the virtual memory routines of the OS, causing overhead.

Data Fragmentation FileMR benefits from the contiguity of the file data. The internal fragmentation of a file system can happen for two reasons: file system aging [19, 45] and using POSIX IO that changes file layout frequently. To evaluate the FileMR performance in a fragmented file system, we first warmed up the file system using four IO intensive workloads that issued POSIX IO requests: varmail and fileserver workloads in filebench [53], Redis [41] and SQLite [46] (using MobiBench [18]). Once the file system was warmed up and fragmented, we created memory regions over all files in NVMM. Table 5 summarizes the workloads.

As shown in Figure 7, running FileMR over the fragmented file still shows dramatic improvement on region registration time and memory consumption for MTT entries. Fileserver demonstrates the case with many files, where FileMR only creates 0.5% of the entries of traditional memory regions, and requires only 6.8% of the registration time. For a metadata-heavy workload (Varmail), FileMR only reduces the number of entries by 3% (due to the heavy internal fragmentation and small file size), but it still saves 20% on registration time because it holds the inode lock, which has less contention. Redis is a key-value store that persists an append-only file on the IO path, and flushes the database asynchronously — little internal fragmentation means that it requires 2% of the space and time of traditional memory regions. Similarly, SQLite also uses logging, resulting in little fragmentation, and drastic space and time savings.

6.3 Translation Cache Effectiveness

The performance degradation of RDMA over large NVMM is mainly caused by the pin-down cache misses (Figure 2). Since Soft-RoCE encapsulates RDMA messages in UDP and accesses all RDMA state in DRAM, we cannot measure the effectiveness of the cache through end-to-end performance.

Instead, we measure the cache hit ratio of our emulated pin-down cache and range pin-down cache for FileMR. We collect the trace of POSIX IO system calls for workloads described in Table 5, and replay them with one-sided RDMA verbs to a remote host.

Figure 8 shows the evaluation result. Our emulated range-based pin-down cache is significantly more efficient (3.8× - 340×) than the page-based pin-down cache. For large allocated files with a few entries, the range-based pin-down cache shows near 100% hit rate (not shown in figure).

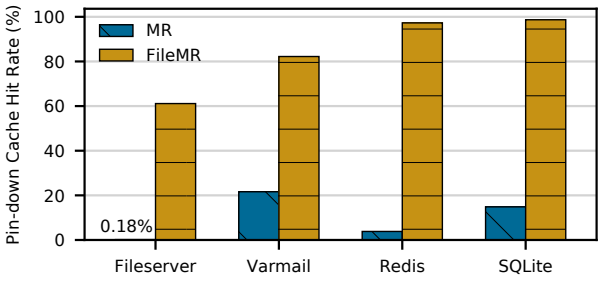

Figure 8: **Translation cache effectiveness.** FileMR significantly increases the effectiveness of the pin-down cache.

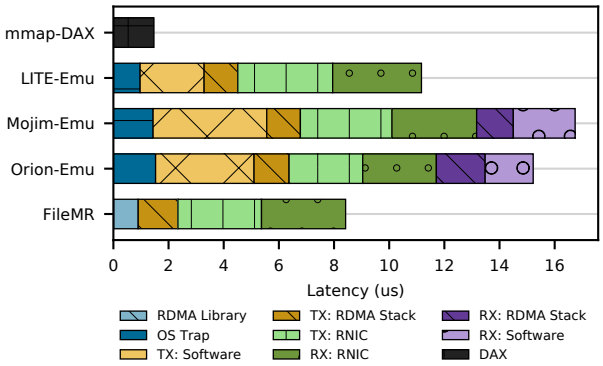

Figure 9: **Latency breakdown of accessing remote file.** FileMR can access remote file location without indirection on datapath.

6.4 Accessing Remote Files

To evaluate the datapath performance, we let a client access files on a remote server running `novad` (introduced in Section 5.1). The client issues random 1 kB writes using RDMA write verbs, and we measure the latency between the client application issuing the verb and the remote RNIC DMAs to the target memory address (`memcopy` for Soft-RoCE).

We compared FileMR with both mmapped local accesses and other distributed systems that provide distributed storage access. We implemented datapath-only versions of Mojim-Emu [64], LITE-Emu [56] and Orion-Emu [62] for Soft-RoCE. All these systems avoid translation overhead by sending physical addresses on the wire. We will further discuss these systems in Section 8.

In Figure 9, we show the latency breakdown of these systems. Note that the latencies of all systems are higher than a typical RDMA NIC, because Soft-RoCE is less efficient than a real RNIC. Also, we omit the latencies of UDP packet encapsulation and delivery, which dominate the end-to-end latency. It only takes 1.5 μs to store and persist 4 kB data to local NVMM. FileMR has lower latency than other systems because it eliminates the need for any indirection layer (`msync()` system call for Mojim, shared memory write for LITE, and POSIX write for Orion).

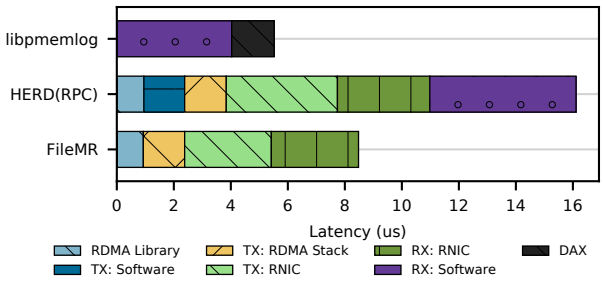

Figure 10: **Latency breakdown of accessing remote log.** With the append verb, Remote logging with FileMR achieves similar performance to local one.

6.5 Accessing Remote NVMM logs

Finally, we evaluate our introduction of the new append verb using our remote log implementation introduced in Section 5.2. We compare to a baseline `libpmemlog` on using local NVMM (bypassing the network), as well as with logging within the HERD RPC RDMA library [21, 22].

Figure 10 shows the latency breakdown of creating a 64 Byte log entry. It takes 5.5 μs to log locally with `libpmemlog`. FileMR adds 53% overhead for remote vs. local logging, while the HERD RPC-based solution adds 192% overhead.

7 Discussion

The current FileMR implementation relies on software-based RDMA protocols. In this section, we discuss the potential benefits and challenges of applying FileMR on hardware and other deeper changes to the RDMA protocol. We consider them to be the future work of this paper.

Data Persistence For local NVMM, a store instruction is persistent once data is evicted from CPU last-level cache (via cache flush instructions and memory fences). A mechanism called asynchronous DRAM refresh (ADR) ensures that the write queue on a memory controller is flushed to nonvolatile storage in the event of a power failure. There are no similar mechanisms in the RDMA world since ADR does not extend to PCIe devices. Making the task even more difficult, modern NICs are capable of placing data into CPU cache using direct cache access (DCA) [17], conceivably entirely bypassing NVMM.

The current workaround to ensure RDMA write persistency is to disable DCA and issue another RDMA read to the last byte of a pending write [9], forcing the write to complete and write to NVMM. Alternatively, the sender requests that

the receiving CPU purposefully flush data it received; either embedding the flush request in an extra send verb or the immediate field of a write verb.

A draft standards working document has proposed adding a `commit` [50] verb to the RDMA protocol to solve the write persistency problem. A `commit` verb lists memory locations that need to be flushed to persistence. When the remote RNIC receives a commit verb, it ensures the all listed locations are persistent before acknowledging completion of the verb.

With the introduction of FileMR, implementing data persistence is simplified since there is no longer need to track modified locations at the client: the RNIC already maintains information about the files. A commit verb can simply request that all updates to a file are persisted, which is analogous to an `fsync` system call to a local file, which is usually light-weight. Even better, since the commit needs little state, a commit flag can be embedded to the latest write verb, reducing communication overhead.

Connection Management Several NVMM-based storage systems [30,43,56,62] store data across nodes, or use a model similar to distributed shared memory. This model requires establishing N^2 connections for N servers with NVMM. For user-level applications, the reliable connection transport enforces the protection domain within the scope of a process. Thus a cluster with N servers running p processes will establish $N^2 \times p^2$ connections.

Existing works [12, 12, 49] reduces this complexity by sharing queue pairs [49], multiplexing connections [27] or dynamically allocating connections [12] to reduce the RDMA states. These optimizations work well for MPI-based applications, but it is challenging to implement them for NVMM applications, especially for applications with fine-grained access control. In particular, a file system supports complex access control schemes, which may disallow sharing and multiplexing.

With the FileMR, the file permission is checked at the bind step, and so each server only requires a single connection to handle *all* file system requests, drastically reducing the amount of states required to store on the RNIC.

For NVMM, data replication is essential for reliability and availability. Existing RDMA-aware systems on distributed NVMM [6,33,43,62,64] transfer data multiple times to replicate NVMM because of the limitation (unreliable datagrams and two-sided verbs) of the existing RDMA protocol. The RDMA payload could be potentially multicasted by the current network infrastructure with FileMR, allowing a single RDMA verb to modify multiple copies of the same file.

Page Fault on NIC Some ethernet and RNICs support page fault or on-demand paging [28, 29] (ODP). When using ODP, instead of pinning memory pages, the IOMMU marks the page as not present in IO virtual addresses. The RNIC will raise an interrupt to operating system when attempting DMA to a non-present page. The IO page fault handler then fills the entry with the mapping.

With ODP, a page fault is very expensive. In our experiment, it takes 475 μs to fulfill an IO page fault and complete a 8-byte RDMA write on a Mellanox CX-4 RNIC. In contrast, it only takes 1.4 μs to complete when the mapping is cached in the RNIC. In general, prefetching is a common way to mitigate the cost of frequent page faults. An optimization for ODP introduces `ioctl(advise_mr)` to hint prefetching [44], and recent research uses `madvise` system call to help prefetching local NVMM [8].

The design of FileMR is orthogonal to ODP, though it leverages the `append` verb. Fortunately, the file system is situated to provide better locality by prefetching ranges based on the file access pattern.

8 Related Work

The FileMR abstraction sits at the intersection of work in address translation, RDMA, and NVMM systems.

8.1 Address Translation

Reducing the cost of address translations has been the focus of work spanning decades. We here describe some common, general approaches and how they can be applied to RDMA and NVMM.

Using Hugepages Using hugepages is the standard way to reduce address transaction overhead and TLB pressure. In Linux, applications can explicitly allocate buffers from `libhugetlbfs`, which manages and allocates from a predefined page pool. An alternative is to allow the kernel to manage hugepages transparently using transparent hugepages (THP) [3, 37] or page swap [61] in the kernel, where the OS tries to allocate hugepages and merge smaller pages into hugepages (via compaction or swapping) in the background.

There are three drawbacks of using hugepages with RDMA and NVMM considered. First of all, applications [21,30,36] that use libraries such as `libhugetlbfs` will manage memory directly and bypass the file system. Second, transparent hugepage will violate the consistency of the MTT entries on the RNIC. Finally, since the current memory region uses a flat namespace, only one type of memory region is supported, which causes fragmentation when using hugepages. By introducing range-based translation, FileMR reduces the number of translation table entries significantly, while retain the support for file system managing the layout of the files.

Access Indirection Several existing works addressed the issue of accessing flat memory space by introducing an indirection layer for accesses and optimizing the communication cost.

LITE [56] uses physical memory region and lets all requests go into the kernel via shared memory. Remote Region [2] also redirects requests to kernel but consists of a pseudo-file system and a user library. Hotpot [43] and Mojim [64] use customized interfaces over memory mapped regions with

support such as data replication and allocation. Storm [35] improves RDMA performance by introducing a new software stack that creates less RDMA states.

With the customized interface, these systems manage the data structures and RDMA states internally to reduce the state handled by RNICs. Programming with these interfaces is unintuitive, especially when working with an NVMM library that manages the data through memory-mapped files and handles allocations with its interface. Additionally, maintaining a physical memory region allows the remote server access arbitrary physical addresses, including DRAM.

Virtual Memory Contiguity As the size of the physical memory keeps increasing while the TLB size has grown slowly, several previous works discuss the contiguity of virtual memory address space. There are proposals on architecture support for coalesced [38], range [23] or segment [5] based address management, or accessing physical addresses with necessary permission checks [15]. In this paper, we assume the NIC hardware is capable of handling range-based entries using one of these mechanisms. Using software-based transparent page management [1, 61] can also increase the contiguity of virtual memory.

8.2 RDMA and NVMM

As discussed in the introduction, building systems that leverage the direct memory access of RDMA and NVMM is appealing. Significant work has already been completed.

Large RDMA Regions Creating memory regions over large memory with a flat namespace has become a popular choice for building RDMA-aware systems, even for those without a persistent memory component. Several systems use this strategy, including key-value stores [21, 34, 36], distributed memory allocators [2, 36, 55, 56], transactional systems, RPC protocols [20, 22, 47], and file systems [43, 62]. To better facilitate this use case, optimizations to the RDMA protocol such as on-demand paging [28, 29], dynamically connected transport [12], multi-path RDMA [31] have been purposed.

NIC Design PASTE [16] is a customized NIC designed for NVMM. It tightly couples the traditional networking stack with the NVMM file system. It provides a holistic design which performs naming and persistence in the networking stack. FileMR is designed for RDMA networking with NVMM file system, and supports general purpose verbs. FlexNIC [24] is a NIC that supports offloading software routines, such as key-value interface and packet classification down to the NIC. Floem [39] provides a programmable abstraction that describes the offload scheme. In contrary, FileMR focuses on a specific use case and can be further extended to support rich semantics.

Distributed File Systems Building distributed file systems by providing remote access at the file system level should provide remote access without interface changes. Orion [62] and Octopus [30] are two distributed NVMM-aware file systems. Orion is a kernel level file system that incorporates RDMA functionalities, and uses physical addresses for RDMA accesses. Octopus is a user level file system using FUSE [26] interface with hugepages to reduce page table entries. When using these file systems, all POSIX file accesses are intercepted and transferred with the file system routines.

There are two major issues in building distributed functionalities in the file system layer: the overhead of calling file systems routines and the granularity of access. In Linux, issuing system calls are expensive and involve multiple memory copies. In a DAX file system, the kernel still copies data from user buffers for security purposes. For mmapped data, the kernel supports "flushes" only in page granularity. These operations are expensive on large memories because the kernel needs to identify the dirty pages and persist them via memory flushes.

Large Memory and Persistent Connection This paper focuses on a particular way of utilizing RDMA networking: create memory regions over large, persistent memory with a flat namespace and maintain them for remote accesses during the application lifecycle. Alternative mechanisms for managing accessible RDMA, including using bounce buffers [52], registering a transient memory region for every access [51], and using tricks (DMA MR [62], Fast MR [58], FastReg MR [13]) require physical addresses in kernel space. We do not consider these solutions because they violate the file system's ability to manage the physical address space.

9 Conclusion

The conflicting systems on metadata management between NVMM and RDMA causes expensive translation overhead and prevents the file system from changing its layout. This work introduces two modifications to the existing RDMA protocol: the FileMR and range-based translation, thereby providing an abstraction that combines memory regions and files. It improves the performance of RDMA-accessible NVMMs by eliminating extraneous translations, while conferring other benefits to RDMA including more efficient access permissions and simpler connection management.

Acknowledgments

We thank our shepherd, Amar Phanishayee and the reviewers for their comments and suggestions. We would like to thank Tom Tapley for his insights and feedback, and Jiawei Gao for suggestions on improving the writing. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

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

[3] Andrea Arcangeli. Transparent hugepage support. In *KVM forum*, volume 9, 2010.

[4] InfiniBand Trade Association. SOFT-RoCE RDMA Transport in a Software Implementation, 2015.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[6] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. Rdmc: A reliable rdma multicast for large objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82. IEEE, 2018.

[7] Dave Chinner. xfs: updates for 4.2-rc1. https://patchwork.kernel.org/patch/10723687/, 2015. Accessed 2020-1-1.

[8] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[9] Chet Douglas. RDMA with PMEM, Software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf. Accessed 2020-01-01.

[10] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE micro*, (2):66–76, 1998.

[11] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, May 2016.

[12] Richard Graham. Dynamically Connected Transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf. Accessed 2020-1-1.

[13] Sagi Grimberg. Introduce fast memory registration model (FRWR). https://patchwork.kernel.org/patch/2829652/. Accessed 2020-01-01.

[14] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[15] Swapnil Haria, Mark D Hill, and Michael M Swift. Devirtualizing memory in heterogeneous systems. In *ACM SIGPLAN Notices*, volume 53, pages 637–650. ACM, 2018.

[16] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. Paste: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 17–33, 2018.

[17] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 50–59. IEEE, 2005.

[18] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android Storage Performance Analysis Tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[19] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–704, Boston, MA, July 2018. USENIX Association.

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

[21] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[22] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[23] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

[24] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 67–81. ACM, 2016.

[25] Gurkirat Kaur and Manju Bala. Rdma over converged

ethernet: A review. *International Journal of Advances in Engineering & Technology*, 6(4):1890, 2013.

[26] http://fuse.sourceforge.net/.

[27] Matthew J Koop, Jaidev K Sridhar, and Dhabaleswar K Panda. Scalable mpi design over infiniband using extended reliable connection. In *2008 IEEE International Conference on Cluster Computing*, pages 203–212. IEEE, 2008.

[28] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. *ACM SIGOPS Operating Systems Review*, 51(2):449–466, 2017.

[29] Liran Liss. On demand paging for user-level networking. https://openfabrics.org/images/eventpresos/workshops2013/2013_Workshop_Tues_0930_liss_odp.pdf. Accessed 2020-01-01.

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

[31] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for rdma in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, 2018.

[32] Mellanox. Physical Address Memory Region. https://community.mellanox.com/s/article/physical-address-memory-region/. Accessed 2020-01-01.

[33] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[34] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

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

[36] Tayo Oguntebi, Sungpack Hong, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. FARM: A prototyping environment for tightly-coupled, heterogeneous architectures. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 221–228, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 679–692. ACM, 2018.

[38] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE Computer Society, 2012.

[39] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[40] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[41] redislabs. Redis, 2017. https://redis.io.

[42] Arthur Sainio. NVDIMM: Changes are Here So What's Next. *In-Memory Computing Summit*, 2016.

[43] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337. ACM, 2017.

[44] Moni Shoua. Add support to advise_mr. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html, 2018. Accessed 2020-1-1.

[45] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–213, 1997.

[46] SQLite. SQLite, 2017. https://www.sqlite.org.

[47] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[48] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15. ACM, 2017.

[49] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K Panda. Shared receive queue based scalable mpi design for infiniband clusters. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[50] Talpey and Pinkerton. RDMA Durable Write Commit. https://tools.ietf.org/html/draft-talpey-rdma-commit-00. Accessed 2020-01-01.

[51] T Talpey and G Kamer. High Performance File Serving With SMB3 and RDMA via SMB Direct. In *Storage Developers Conference*, 2012.

[52] Haodong Tang, Jian Zhang, and Fred Zhang. Accelerating Ceph with RDMA and NVMeoF. In *14th Annual OpenFabrics Alliance (OFA) Workshop*, 2018.

[53] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[54] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 308–314. IEEE, 1998.

[55] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R Gross. Rstore: A direct-access DRAM-based data store. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 674–685. IEEE, 2015.

[56] Shin-Yeh Tsai and Yiying Zhang. LITE: Kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324. ACM, 2017.

[57] Matthew Wilcox. Add support for NV-DIMMs to ext4, 2014. https://lwn.net/Articles/613384/.

[58] Bob Woodruff, Sean Hefty, Roland Dreier, and Hal Rosenstock. Introduction to the InfiniBand core software. In *Linux symposium*, volume 2, pages 271–282, 2005.

[59] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[60] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[61] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 698–710, New York, NY, USA, 2019. ACM.

[62] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, 2019.

[63] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST '20. USENIX Association, 2020.

[64] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.