2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00029

RVMA: Remote Virtual Memory Access

Ryan E. Grant, Michael J. Levenhagen, Matthew G.F. Dosanjh, Patrick M. Widener
Sandia National Laboratories*
Center for Computational Research
Email: {regrant,mjleven,mdosanj,pwidene}@sandia.gov

***Abstract*—Remote Direct Memory Access (RDMA) capabilities have been provided by high-end networks for many years, but the network environments surrounding RDMA are evolving. RDMA performance has historically relied on using strict ordering guarantees to determine when data transfers complete, but modern adaptively-routed networks no longer provide those guarantees. RDMA also exposes low-level details about memory buffers: either all clients are required to coordinate access using a single shared buffer, or exclusive resources must be allocatable per-client for an unbounded amount of time. This makes RDMA unattractive for use in many-to-one communication models such as those found in public internet client-server situations.**

Remote Virtual Memory Access (RVMA) is a novel approach to data transfer which adapts and builds upon RDMA to provide better usability, resource management, and fault tolerance. RVMA provides a lightweight completion notification mechanism which addresses RDMA performance penalties imposed by adaptively-routed networks, enabling high-performance data transfer regardless of message ordering. RVMA also provides receiver-side resource management, abstracting away previously-exposed details from the sender-side and removing the RDMA requirement for exclusive/coordinated resources. RVMA requires only small hardware modifications from current designs, provides performance comparable or superior to traditional RDMA networks, and offers many new features.

In this paper, we describe RVMA's receiver-managed resource approach and how it enables a variety of new data-transfer approaches on high-end networks. In particular, we demonstrate how an RVMA NIC could implement the first hardware-based fault tolerant RDMA-like solution. We present the design and validation of an RVMA simulation model in a popular simulation suite and use it to evaluate the advantages of RVMA at large scale. In addition to support for adaptive routing and easy programmability, RVMA can outperform RDMA on a 3D sweep application by 4.4X.

I. Introduction

Remote Direct Memory Access (RDMA) has been a feature of high-performance networks for many years. RDMA provides semantics that are similar to local host memory operations, but use a remote node's memory as the target. RDMA is commonly referred to as "one-sided" communication, requiring the target node to operate on incoming data at the Network Interface Controller (NIC) level. Avoiding host CPU involvement in data movement is a key benefit of RDMA. RDMA avoids this involvement by including the target memory addresses of RDMA operations in the network packets themselves. Including this information requires pre-registering the memory region on the target node and communicating the address and length of the registered region to the source. The source node stores this information and includes it when communicating with the target on the remote host.

*Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The semantics enforced by RDMA can make it difficult to use. Essentially, RDMA provides a remote memory subsystem without a coherency mechanism. The target node of RDMA operations does not understand what operations are expected to be performed and therefore cannot understand when operations are complete. It is not straightforward to determine when all of the operations on a memory region are complete and memory is again available for use on the target-side node. In addition, memory resources that have been allocated for remote operations remain dedicated to the corresponding remote nodes for an unspecified amount of time. Some special RDMA operations are capable of generating observable completion events on the target node (e.g., RDMA write/put with immediate). The data payloads sent by these commands can fit in a single packet, defining the completion for the target node. Unfortunately, those payloads are small — typically under 64 bytes in size. For all larger data transfer sizes, completion notification is normally a small additional packet that can deliver an event once all of the RDMA operations are complete.

Two hardware-level changes would greatly improve RDMA. The first is a lightweight notification mechanism for signaling the completion of operations on a buffer on the target-side node. Lightweight notification is possible on traditional statically routed networks, where byte-level write ordering can ensure that the last byte in a receive buffer was the last byte written. That last byte can then be polled as the basis of a flag notification mechanism. Static network routing is being replaced by adaptive techniques for increased network efficiency on advanced network topologies, making notifications based on byte-level ordering infeasible in future networks. Viable alternative notification mechanisms all rely on operation ordering, requiring additional messages for completion.

The second hardware change which would improve RDMA would shift responsibility for management of memory resources from the source of network operations to their target. To enable this, a new method will need to provide the ability to write directly into remote system memory without requiring a specific memory address. This eliminates the need to exchange the specific memory address on the remote node and to store it locally for initiating future transfers. In addition, a receiver must be able to manage its own resources. Without this, a system can be flooded by external requests that it cannot refuse, which can be detrimental to performance. In order for a receiver to ensure timely usage of its resources, it should not assign resources requested from remote sources without a guarantee that said resources will be used in a timely manner. Ideally receivers will manage their resources independent of control by remote systems.

Our proposal, Remote Virtual Memory Access (RVMA), is a result of significant hardware-software co-design that provides both of these key improvements. First, RVMA provides a virtual address abstraction as the external-facing interface. The virtual address in RVMA is not a physical memory address as in RDMA, but rather a mailbox address. This allows RVMA addresses to serve as incoming message buffer attachment points, which allows for traffic stream separation and basic message matching capabilities. Second, RVMA provides a lightweight completion mechanism that is significantly different from modern event/completion queue designs. It provides completion notification by writing a buffer address and length to a predefined memory address on a per-buffer basis. The update of this notification memory address is a signal that the associated data buffer is complete, and requires some additional information to be posted with buffers for hardware-level completion. In addition, there are methods by which the user can pre-empt hardware level completion, handing the buffer over to software; this is useful in cases where the amount of data is unknown, allowing for out-of-band completion notification.

Each RVMA mailbox address is mapped to a queue of buffers posted by the target node. Each buffer in the queue is associated with a *completion pointer* and a *threshold*. The completion pointer is written once the buffer is determined to be full and points to the head of that full data buffer. The NIC uses the threshold to determine whether the buffer is full. The threshold is a count of either bytes or operations, and once reached indicates that the buffer is complete and the completion pointer should be updated (which in turn notifies the application). This deceptively simple abstraction is incredibly powerful, and enables a host of features that are not currently possible with RDMA while improving the efficiency of existing RDMA features.

RVMA's lightweight notification method is broadly applicable to existing RDMA networks, as it requires a small amount of additional state on the NIC and an extension to the software interface. This enables the use of a Monitor/MWait mechanism (quick thread wakeup on memory location update) in addition to the traditional memory polling on completion event queues. Notification happens whenever a registered RVMA target buffer is filled and deregistered, allowing the user to determine how often notification may occur based on their sizing of RVMA window buffers. This notification approach is superior to the completion event queue model, as individual completions can use Monitor/MWait (as they have known notification addresses) instead of a shared completion queue. In this paper we will show how RVMA can enhance HPC networks although the technique is also able to support sockets-based applications.

The contributions of this paper are as follows:

- We propose Remote Virtual Memory Access (RVMA), a new method of performing zero-copy remote memory operations which provides lightweight hardware message completion notification and efficient target-side resource management.
- We illustrate how Remote Virtual Memory Access can solve the completion notification difficulties in traditional RDMA.
- We show how RVMA can be used to create the world's first hardware-level fault-tolerant remote memory access technology, allowing for "rewinding" of RVMA operations;
- We describe an RVMA simulator model developed in a major simulation software suite and use it to assess performance at scale. The models are validated against performance results from existing RDMA solutions with timing data representing the subset of RDMA primitives that would be needed for RVMA.

II. Background

Remote Direct Memory Access (RDMA) extends traditional on-node memory operations to remote nodes. RDMA operations are described slightly differently between different network

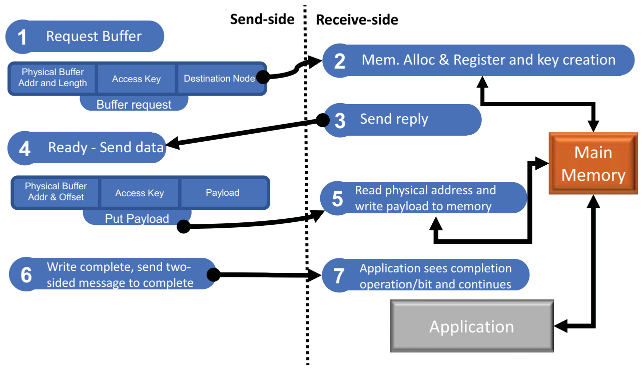
Send-side
Receive-side
1 Request Buffer
Physical Buffer Addr and Length
Access Key
Destination Node
Buffer request
2 Mem. Alloc & Register and key creation
3 Send reply
4 Ready - Send data
Physical Buffer Addr & Offset
Access Key
Payload
Put Payload
Main Memory
5 Read physical address and write payload to memory
6 Write complete, send two-sided message to complete
7 Application sees completion operation/bit and continues
Application

Fig. 1: RDMA Put/Write Operation Example

interfaces, but retain the basic functionality of writing the contents of a local buffer into remote memory or fetching remote data into a local buffer. InfiniBand uses "write" or "read" to describe remote operations, while many other interfaces such as SHMEM use "put" and "get" terminology to describe these basic functions.

The basic functions of RDMA are fundamentally an extension of the traditional on-node memory subsystem interface in modern RISC systems. Unlike the memory subsystem on-node, RDMA operations are not coherent between nodes. A write to a remote node is not immediately observable on the target node, unlike coherent caches in a single system. This is a significant difference. An on-node memory subsystem does not require applications to explicitly manage memory to ensure that all pending write operations are complete before reading data from memory. RDMA imposes this requirement on applications as there is no explicit completion method for RDMA transfers (with the exception of single packet commands which can have corresponding events).

RDMA bypasses the need for CPU processing of incoming packets. As memory operations are handled directly by a DMA engine on the NIC, the CPU does not need to be involved in determining the correct memory address for incoming data. This enables very low latency handling of incoming messages and low CPU overhead for network processing. However, this feature is part of the reason why RDMA cannot provide notifications when operations complete on the target side. RDMA NICs have no method of determining when all of the operations for a memory region are complete, as the source-side of the operations controls their types and timing.

To address this, RDMA requires that a source and target pre-negotiate a buffer before any operations occur between them. Figure 1 illustrates the steps in RDMA Put/Write operations. Initiators must contact the target node and request a memory region be allocated and the address and length of that region be returned to the initiator (step 1). The target must then allocate and register the buffer (step 2). This data must then be retained by the initiator and included in all RDMA operations involving that target (step 3). Data can be sent from the initiator using the allocated receive-side buffer (steps 4 & 5) until the sender/initiator is done at which time the sender must signal completion (steps 6 & 7). Traditional RDMA in this way places all control of operations and completion in the hands of the initiator, and the target has no knowledge that would allow it to determine when operations are complete.

RDMA is said to be "one-sided" in that source nodes "own" memory regions on remote nodes. A target-side memory region reserved by remote nodes is dedicated for that purpose until the initiator indicates that no further operations will happen on the memory region (step 6 in Figure 1). There are no guarantees that initiators will use the memory registered for RDMA operations in a timely manner. While a target node can shut off a remote memory region to reclaim it, the ability to manage resources is binary, in that the memory region is either accessible remotely or not.

Completion of RDMA operations requires a secondary mechanism with strict ordering to understand when buffers are safe to read on the target. For write or put operations, it is not obvious when incoming RDMA operations on a buffer are finished. Any given write/put operation could complete, but the number of allowed operations to a remote memory region is not bounded. Operations can even overlap areas of memory; it is valid to have multiple write/put operations to a single memory location, overwriting previously written values. The only way to determine when a memory region is safe to read on the target node is through a notification from the initiator, which alone has the knowledge that all operations have completed (step 6 in Figure 1).

Low-latency notification can be accomplished in several ways. Two basic techniques exist: polling on a completion data structure, or allowing hardware to sleep on modification of host memory (such as the semantics for Monitor/MWait operations). Polling is the most common method and takes two forms. Polling on the final bit/byte of a RDMA buffer has been used for low-latency notification. However, this requires byte-level write ordering and complete packet level ordering to work correctly. In a non-ordered network like an adaptively routed network, the last packet may arrive before other payload packets and prematurely trigger "completion", leading to corrupted data buffers. Alternatively, polling on a completion queue can be implemented by using an operation performed after the RDMA Write/Put has completed to indicate to the receiver that the RDMA operation is also complete. RDMA operations do not typically create completion queue entries with a small number of exceptions that have very limited payloads. Monitor/Mwait is a set of operations that allow a hardware thread to be put to sleep and awoken in a very short time period (one to several clock cycles) when the memory associated with a cache line is modified. In practice Monitor/MWait is not extensively used due to difference between hardware implementations and practicalities of waking on changes to a shared completion queue.

Many software interfaces support one-sided communication, which are typically aligned with common RDMA hardware functions. InfiniBand has supported RDMA operations for decades, and the IB Verbs API uses write/read operations over queue pairs to accomplish this. All other major HPC networks also support RDMA operations; for example, Aries network supports `puts` and `gets` while Portals-based [1] networks provide `PtlPut` and `PtlGet` calls.

One-sided communication can have alternative matching/lookup approaches versus a direct memory address included in an RDMA command header. Fundamentally, all of the data for programming a DMA engine can be carried in the message itself. However, local management of resources is possible with rich matching support in hardware like that provided in a Portals [1] network. Portals networks provide rich matching based on matching elements that have source network addresses and a special matching tag bit for each posted buffer. This allows steering of put and get operations to buffers determined by the target. This means that for a Portals network, there is no requirement that memory address information be exchanged prior to communication.

Hardware lookup methods vary in cost and speed depending on the required size and features of the lookup engine. Lookup engines can be simple LUT devices or much more complex alternative like ternary content addressable memory (TCAM). TCAMs can allow for quick lookup and support wild-card lookups as well. However, the capabilities of a TCAM come at the expense of capacity. The largest TCAMs in common networking usage are in switches with capacities on the order of a few thousand elements. Simple lookup tables can provide more capacity, and may be designed to provide similar performance to a TCAM, but without support for wildcards and potentially fewer components that can be matched.

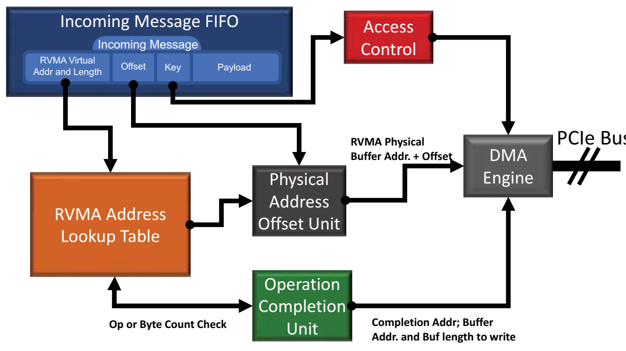
Incoming Message FIFO
Incoming Message
RVMA Virtual Addr and Length
Offset
Key
Payload
Access Control
RVMA Physical Buffer Addr. + Offset
PCIe Bus
DMA Engine
Physical Address Offset Unit
RVMA Address Lookup Table
Operation Completion Unit
Op or Byte Count Check
Completion Addr; Buffer Addr. and Buf length to write

Fig. 2: RVMA Architectural Diagram

III. Architecting and Implementing RVMA

In this section, we discuss details of a potential RVMA implementation. First, we detail a high-level architectural description of RVMA. Next, we describe how a traditional RDMA put operation would change in RVMA and present our proposed RVMA API. The reasoning behind the architectural decisions and benefits thereof will be discussed in detail in Section IV.

A. RVMA Architecture

There are two key concepts to the Remote Virtual Memory Access (RVMA) architecture. The first is abstraction of the data buffer addresses, the benefits of which are discussed in Section IV-A. This happens both on the initiator and on the target. The initiator observes only an abstract virtual memory address on the target representing a mailbox. Instead of receiving the low-level details of the target's physical memory addresses, it only uses a basic virtual mailbox address and offset into a buffer registered to that mailbox at the target. The target host CPU also sees an abstraction of a buffer and interacts with the remote buffer through a pointer to the head of the last completed buffer in memory. For RVMA, support for abstracted "virtual" addresses requires the NIC to handle an extra level of redirection via established memory pointers with which incoming data is steered. While the idea of hardware message matching is not new [1], RVMA takes a simplified approach using a simple lookup table (shown in the RVMA architecture diagram in Figure 2). Unlike other hardware message matching approaches meant for MPI support, RVMA does not allow wildcards in the lookup, meaning that it always has a single-lookup response (item found or no item found). The benefits of this are discussed in Sections IV-B and IV-D.

The second key concept is the novel completion mechanism for RVMA. An RVMA NIC recieves information included when receive buffers are posted that allows the NIC to provide completion notification; either a number of operations to be performed on a buffer or a number of bytes to be written. RVMA requires two addresses for any posted buffer: the address of the head of the buffer in memory and the address of a "completion pointer". The completion notification pointer is a cache line aligned memory address stored at a memory location agreed upon by the NIC and the host. When an RVMA buffer is complete, the completion unit, shown in Figure 2, recognizes this and the NIC writes the head of the buffer's address to the location indicated by the completion

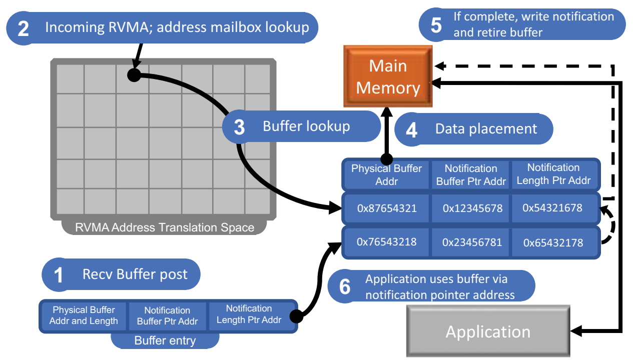
2 Incoming RVMA; address mailbox lookup
5 If complete, write notification and retire buffer
Main Memory
3 Buffer lookup
4 Data placement
Physical Buffer Addr
Notification Buffer Ptr Addr
Notification Length Ptr Addr
0x87654321
0x12345678
0x54321678
0x76543218
0x23456781
0x65432178
RVMA Address Translation Space
1 Recv Buffer post
6 Application uses buffer via notification pointer address
Physical Buffer Addr and Length
Notification Buffer Ptr Addr
Notification Length Ptr Addr
Buffer entry
Application

Fig. 3: RVMA Message Receive Flow Diagram

pointer. It also deactivates the buffer entry in the RVMA lookup table (Figure 2). In particular, the application or communication middleware needs only to check the memory location indicated by the completion pointer. This has several advantages that will be discussed in detail in Section IV-C. While RVMA NICs can use traditional event queues for notification, the existence of the buffer abstractions on the target side allow for new notification mechanisms with new benefits (see Sections IV-D, IV-E, and IV-F).

B. Applying RVMA to a Put Operation

To illustrate how the key concepts of RVMA are applied, we first walk through the steps of an RVMA put command (Figure 3). Before an RVMA operation can occur, a receive buffer must be posted to an RVMA mailbox on the target. When a buffer is posted, it must include the RVMA mailbox address to which the buffer will be attached, the physical memory address, and length of the buffer. In addition, it must include the threshold value (the number of operations or bytes expected for completion), the address of the completion pointer, and an address where the length of the completed buffer will be written. Typically these two completion addresses will be consecutive and be aligned to a single cache line for optimal completion notification efficiency.

Once receive buffers have been posted (step 1 in Figure 3), an RVMA command can be initiated by a remote host. No handshaking to discover a remote address is required as in traditional RDMA, since RVMA addresses can be organized by applications without a discovery step. This is possible because an RVMA address, unlike traditional RDMA, is not interpreted as an actual memory address of the buffer on the target side, but an identifier to a virtual RDMA buffer. Therefore, the target side NIC must keep a translation table from the virtual addresses used by sources to the actual physical locations of memory on the target. This also allows multiple buffers to be attached to a single RVMA mailbox address.

When an RVMA command has been received, address translation is performed (step 2), making the data payload ready to be written to memory (steps 3 and 4 in Figure 3). At this point a completion check is performed, incrementing a counter associated with the virtual address and checking it against the completion threshold that was defined when the buffer was posted. If the buffer is complete (determined by bytes or operations), a memory write occurs to the completion pointer address with the memory address of the head of the data buffer (illustrated in step 5 in Figure 3). The length of the buffer is also written, so applications may safely use the data buffer in cases where the length of the transfer was not known when the receive command was posted. An application is now free to read the data buffer (step 6).

Enabling completion checks like those in our example requires a small amount of state on the NIC and a counter for each virtual memory address that has an active buffer posted to it. In practice, the RVMA translation table will be very sparse, meaning that a limited number of counters are required for typical operation. In cases where the NIC counters are fully occupied, host memory can be used, albeit with a potentially significant performance penalty. For PCIe Gen 6+ this performance penalty is minimal due to bus latency improvments (10 of ns vs 200 today). This penalty can be avoided entirely by having a number of counters that is equivalent to current RDMA hardware for counting on Queue Pairs which is suffiient for avoiding such overflow in current generation systems.

Completion criteria is definable for most codes, as the operations on a buffer are predictable. In cases where such knowledge is not available, buffers can be designated complete by the target NIC at any time. This is useful for applications where the amount of data being received is not known. In some sense, this mode of RVMA falls back to RDMA-like completion, in that it would typically use special control messages to indicate that the buffer is complete. RVMA still benefits from the buffer address abstraction on the target and, for networks without rich matching semantics (e.g. Portals4 networks [1]), the virtual addressing.

When a buffer is completed, the RVMA NIC must change the physical address that the RVMA mailbox “virtual address” references by switching to the next buffer registered for that RVMA address. The virtual address itself does not change, allowing the window to be reused without requiring the initiators of transfers to perform a handshake for a new physical buffer address on the target. Traditional RDMA requires that any new buffer must exchange data such that the initiator can properly address commands to that buffer.

An RVMA mailbox address can be thought of as an identifier of a “bucket” of buffers, or a mailbox including multiple possible receive buffers. This addressing scheme allows RVMA to avoid the costly initialization and setup of remote buffers in RDMA. It also decouples receiver-side operations from any need for byte-level ordering in placement, as the buffers are only completed when so designated by the receiver. The user/application has no visibility into the buffer until the RVMA receive is complete in hardware. Techniques like polling on the last data buffer bit (that were widely used in statically routed RDMA) are not needed, and byte-level ordering requirements are consequently relaxed.

Unlike RDMA, RVMA uses “virtual addresses” of mailboxes of data buffers. This prevents the use of certain manipulations of physical addresses common in RDMA. Offsets supplied with an RVMA command are interpreted as offsets into the currently active abstracted (head) buffer corresponding with the RVMA address. RVMA operations only operate on the head buffer in the “bucket” for a given RVMA address (mailbox). Because of this, RVMA commands cannot be coordinated with different base RVMA addresses to assemble a contiguous payload in remote memory. For example, a put of 32 bytes to RVMA mailbox address 0x11FF0011 followed by a put of 32 bytes to RVMA address 0x11FF0031 does not result in a (overwritten) message of 32 bytes starting at address 0x11FF0011, but two different 32 byte messages written to different buckets/mailboxes. Similarly, sending two messages to the same RVMA address (mailbox) where each message triggers the completion threshold will result in the application receiving two separate buffers out of the “bucket” of buffers attached to the RVMA mailbox address.

To accomplish a contiguous 64 byte message send with RVMA, two messages can be sent to a given mailbox address (0x11FF0011) but with offsets 0 and 32 or message numbers of 1 and 2. Note that for message separation, one can use distinct RVMA addresses (mailboxes) to avoid interleaving data from multiple messages. A sufficiently large RVMA mailbox address space must be provided; we assume a 64-bit address space in this paper, which is sufficient for approximately 4.3 billion mailboxes. In practice this address size could be reduced, but 64 bits is equivalent to the size of addresses required by RDMA. The translation from an RVMA address to its associated buffers is transparent to the node that initiated the transfer as the translation happens on the target node.

C. Application Programming Interface

Here we detail the main API calls required for an RVMA implementation. For the sake of brevity, we have only included the smallest subset of calls required. Other functionality, such as an RVMA get/read call or options to create catch-all mailboxes for incoming messages that correspond to mailboxes/addresses that do not have buffers posted, would be included in a comprehensive RVMA specification document.

`Init_window` calls specify the virtual address mailbox to which a buffer will be attached and require the memory address of the completion notification pointer (which will be updated with the completed buffer's memory address when the buffer has been filled). This `notification_ptr` address is a pointer to the buffer that was completed in the last epoch (the last consistent buffer space, not the currently in progress communication/epoch). By this method, the currently progressing communication epoch is hidden from the user and only exposed when the hardware has determined that the transfer for that epoch has completed. `epoch_threshold` is interpreted as a total number of operations or the total number of bytes that need to be written on a window to complete the epoch. The interpretation of `epoch_threshold` is defined by the `epoch_type` that can be set to `EPOCH_BYTES` or `EPOCH_OPS`. Although it is not required that the user prevent overlapping RDMA puts, it is recommended that non-overlapping be enforced such that the epoch threshold for byte counted windows can be equal to the size of the window.

```
RVMA_win RVMA_Init_window(void*
virtual_addr, key_t* key, int_64t epoch_-
threshold, epoch_type type)
```

Posting a buffer with `RVMA_Post_buffer` adds a buffer to the list for a RVMA mailbox address.

```
RVMA_Status RVMA_Post_buffer(void *
buffer, int_64t size, void** notification_-
ptr, RVMA_Win)
```

Closing a window with `RVMA_Close_win` prevents further operations to that virtual address. Operations that are attempted on a closed virtual address (mailbox) are automatically discarded and *may* result in a NACK notification to the initiator. NACKs may be disabled to handle DoS attacks and manage overhead for systems that have such concerns.

```
RVMA_Status RVMA_Close_Win(RVMA_Win)
```

The `RVMA_Win_inc_epoch` call allows for returning control of a posted RVMA buffer to the user prior to the completion requirements being met (not having reached the threshold of number of operations or bytes). This call is useful for stream-like semantics where it is desirable to process all messages that have arrived so far or cases where the size of the message is not known in advance. This can also be useful for scientific applications in recovering from errors where a partial buffer may be of use.

```
RVMA_Status RVMA_Win_inc_epoch(RVMA_Win)
```

It is useful to understand what epoch a given virtual address/mailbox is currently operating in. Calling `RVMA_Win_get_epoch` allows for an understanding of the state of attached buffers to a virtual address. For example, system software may want to guarantee that a constant number of buffers are always posted for a given virtual address. Checking the current epoch allows the system software to make sure that buffers are posted in a lightweight manner.

```
int64_t RVMA_Win_get_epoch(RVMA_Win)
```

In order to determine when RVMA buffers are complete, a pointer to the data buffer is deposited into the notification address given when the buffer was first posted to the NIC. Notification addresses are already known to the process that posted the buffer. The `RVMA_Win_get_buf_ptrs` call allows retrieval of notification pointers in an array of notification pointers that is passed into the call, which must be of size `count`. The return value of the `get_buf_ptrs` call is the number of valid notification pointers that were returned; in most cases this will be equal to the requested count.

```
int RVMA_Win_get_buf_ptrs(RVMA_Win,
void* notification_ptrs[], int count)
```

1) Initiator-side API Specific: `RVMA_Put` calls initiate transfers to remote virtual memory addresses/buffers. Data must be sent to a physical or logical network address for a node and a virtual address (mailbox) on said node. Physical and/or logical addresses may include a network ID (NID) and process ID (PID) pair, if remote process space targeting is desirable.

```
RVMA_Status RVMA_Put(void* send_buffer,
int_64t size, struct addr_in* dest_addr,
void* virtual_addr)
```

IV. Novel RVMA-enabled Features

RVMA was designed to address several major issues with RDMA that have become apparent over the last two decades of RDMA use. To improve upon RDMA we found the following features to be the most important.

- *Abstraction of low-level details of the data buffers* that are needed for RDMA, including physical memory addresses, frees senders from reserving resources at the target and storing low-level info about remote buffers. Reserving remote resources complicates RDMA usage in environments where processes could be easily migrated or hosts could change, including failure scenarios.
- *Receiver management of resources* allows targets to have control of their buffers and add flexibility. This also helps with the process migration problem as receivers can steer their own data transparently to the memory spaces required.
- RDMA completion is difficult to manage and lacks many features. RVMA needed a *better completion mechanism* that supports wake-on-complete communication semantics and efficient multi-threaded completion mechanisms. Having a registered handle for a given completion notification (completion pointer) means that completion can be handled in a fine-grained manner as a process only waits on a chosen set of communication requests instead of sharing queues. This also prevents any issues that may arise from running out of completion queue contexts.
- Receiver managed resources and the completion counting features of RVMA provide *efficient support for adaptively routed networks*, completing operations correctly regardless of packet delivery order.

- Explicit support for MPI's *RMA epochs* provides useful control intervals while avoiding the need for explicit high-overhead polling of the NIC to determine when events have completed.
- Finally, our bucket of buffers approach for each RVMA address/mailbox solves the long-standing *RDMA failure recovery* issue.

This section comprehensively discusses how RVMA provides each of these features and their resulting improvements upon RDMA.

A. Buffer Abstraction

The *virtual* in RVMA refers to the abstraction of the addresses used for the buffers. The "address" used in the user level interface is the address of a mailbox to which buffers are registered. RVMA redirects traffic to different mailboxes corresponding to physical buffers. RVMA also provides independent target side completion to logically separate communication regions, or epochs. These logical epochs were always a part of RDMA itself, but were not tied to buffer management on the target side in any way. In RDMA, target side buffers typically remain entirely static, as any buffer changes required a complete handshake and setup on both sides of the communication. In addition, there is no information on which a target can determine when a given buffer is ready to be accessed without explicit notification from the initiator. This is inefficient, especially for short transfers.

RVMA's buffer management and completion notification mechanisms were designed to be lightweight. There is a lookup required to translate the RVMA message address to the target physical buffer address. This requires a sparse table in the NIC that has all of the mailbox addresses in use and a corresponding list of buffers for each mailbox. These requirements are a subset of those of other HPC NIC designs, such as those based on Portals. Portals requires a list of all posted "match" elements. This is similar to buffers registered to a mailbox if one interprets Portals-style match bits and sources to be approximate analogues to RVMA addresses. The key difference between these approaches is that RVMA mailbox lookups can only resolve to a single element. Portals style matching follows Message Passing Interface (MPI) matching semantics that allows wildcards, mask bits for matching tags and then resolves multiple potential matches to a single message by the order in which the potential matches were posted to the NIC. This involves significantly more complex message matching hardware than a known single lookup resolution in RVMA. As the more complicated matching units in Portals NICs are deployed in real systems, such as Bull-Atos BXI supercomputers, the overheads of RVMA can be expected to meet the requirements of world-class high-performance networks.

The memory requirements on the NIC for storing the lookup tables (LUT) also imply low overhead. A LUT on an RVMA NIC would need to store the RVMA mailbox address, the buffer head address, and the completion pointer address. If a 64-bit RVMA address is used, the design space is very large but sparse. For example, dividing the 64-bit address space in half (with a 32-bit source network address and a 32-bit mailbox buffer space, using IP/port-style addressing) would allow an RVMA NIC to address buffers for any IPv4 address with the equivalent of $\tilde{2}.1$ billion "ports". Each entry requires 24 bytes of space but entries can be expected to be very sparse: application are very unlikely to register buffers for all possible network source addresses with billions of ports per address.

RVMA also requires a concept of tracking epochs. The hardware must therefore be able to count either the number of operations performed on a virtual address or the number of bytes written. This is only required for RVMA addresses that have buffers posted to them. Compared to SR-IOV support, these requirements are very small in terms of die area on the NIC.

B. Receiver Steered vs. Managed RVMA

Our example of RVMA operation assumed that all operations must include an offset into the buffer associated with a given RVMA address. It is possible to design a network that also counts received bytes and places incoming packets for a given buffer and places incoming packets consecutively in memory. RVMA was designed to support this alternative mode to match the semantics of socket network interfaces. This allows RVMA to efficiently support sockets-based network code with very minimal middleware support, unlike contemporary sockets-to-RDMA libraries. The discussion of this alternative mode, which we call Receiver-Managed RVMA, is complex in its own right, and so for the remainder of this paper we will concentrate on Receiver-Steered RVMA for HPC networks.

C. Lightweight Completion Notification

Desirable qualities of an RDMA notification mechanism include very low overhead and a very quick response time when an operation completes. RVMA provides this through the completion pointer. The location in host memory that this pointer occupies is only modified by the NIC when the operation is complete. This means that it can be waited on using existing hardware mechanisms that allow for core activation upon memory location modification. This behavior is typically referred to as Monitor/MWait after the set of operations that support this behavior in the x86 ISA. Using this method, threads of execution can be activated in as little as one clock cycle upon the RDMA operation completion. For architectures that do not support such lightweight wake on write operations, the memory location can be polled for change; this provides a similarly low latency but expends more energy. It should be noted that in either case, this is faster than waiting for a subsequent network operation to complete. In the case of byte-ordered networks, this may also be more performant as the underlying network can avoid the overhead required to provide such ordering; this can be significant, particularly when errors occur on the network or packet ordering is poor.

D. Relaxing Byte Write Ordering Requirements

Byte ordering requirements on some networks provide excellent methods of "cheating" typical RDMA rules by polling on the final byte of an RDMA window to determine when completion occurs. This is a common optimization for InfiniBand networks, but violates the InfiniBand specification. The InfiniBand specification states that no RDMA operation can be considered complete until a later send/recv operation has finished. This requires operation ordering on the network, but not byte level ordering. However, many InfiniBand hardware vendors provide complete byte ordering when using static routing, making the tactic of polling on the final byte in a window a useful one. This technique is popular because it significantly increases the performance of RDMA operations. Unfortunately, such ordering is not provided on adaptively-routed networks that are used in state-of-the-art network topologies.

There are several reasons why it is desirable to *not* have byte-level ordering on a network. First of all, it serializes many transfers and makes it much more likely that traffic will stall when packets get dropped or corrupted during transfer. In addition, it makes adaptive routing much more difficult as the network must ensure that the packets don't arrive so far out of order that they cannot be buffered at the target in order to ensure correct write order in host memory. This is why traditionally byte-ordered networks such as Mellanox's InfiniBand implementation cannot use byte-ordered RDMA when moving to new topologies that require adaptive routing. This means that adaptively-routed RDMA traffic needs additional completion mechanisms that do not rely on byte ordering going forward. Currently, the solution is to perform a full send/recv operation after the RDMA operation to ensure completion, as operation ordering is enforced.

RVMA avoids byte-level ordering requirements entirely by using the virtualization address in combination with offsets. This means that as packets arrive they can be placed into host memory irrespective of logical order as they are simply placed in physical memory based on their offsets when they arrive. This simplifies adaptive-routing implementations as there is no temporal requirement that packets arrive in any given order. With RVMA, an RDMA window could be written in reverse order with no performance impact. Combined with RVMA's receiver-side completion ability, RVMA can provide fast completion notification based on operation or byte count irrespective of byte-level ordering on the network. This enables RVMA networks to take full advantage of adaptively routed networks while preserving the efficiency of completion notification of statically-routed network-techniques in current RDMA networks.

E. Multi-Epoch RDMA

MPI's RMA specification uses the concept of an RMA epoch. This epoch defines a time period where a buffer can be remotely accessed and modified. RVMA fundamentally includes the concept of a RMA epoch. Traditional RDMA implicitly supports this idea, but there is no support for determining if a given operation is complete other than completing other network operations that provide explicit event notification or writing flags. RVMA fundamentally uses epochs as a mechanism for changing physical buffer addresses for a given virtual address upon completion. This built-in epoch response allows a user to more easily utilize remote memory regions on a host. RVMA purposely avoids using event queues and notification to avoid high overhead polling requirements which can create large amounts of requests to the NIC. Instead the epoch notification can be relatively transparent to an application, as the pointer to the remote data is updated after data has arrived for an epoch. This also has the benefit of allowing for multiple epochs of communication buffers to be kept to allow for communication rollback to previous epochs. Such rollback is not typically possible with traditional RDMA buffers as the same buffer is reused between epochs.

Existing work has shown that it is possible, although somewhat complex, to detect RDMA epochs in software [2]. While this is possible in current RDMA hardware, RVMA makes this task trivial, as the concept of an epoch is captured in the lightweight notification mechanism. Epochs can help in understanding RVMA communication and providing resilience for RVMA networks.

F. Fault Tolerant RDMA

Building on the capabilities of Multi-Epoch RDMA that RVMA provides, we can easily develop a fault-tolerance scheme for RVMA networks. Fault tolerance is difficult with RDMA as any failure in a given communication epoch leaves the RDMA buffer in an undefined state as, for example, the status of writes is unknown. As the hardware is responsible for these operations and the completion at the hardware level is not recorded, there is no way to recover a half-written buffer to the original contents if an error occurs. RVMA can accomplish this task in hardware through its ring of buffers. For a given RVMA mailbox, multiple buffers can be attached. Consequently, the address of buffers used in previous communication epochs could be retrieved from an RVMA NIC after a failure of a node.

A hardware command to retrieve the buffer address of previous epoch communications is easy to implement as it could simply read information from a hardware list for a mailbox that had been marked as complete. This buffer can then be passed to the application and "rollback" the communication to a previous known state in the computation. Techniques to rollback computation could then be applied for local data. The caveat to this approach is the application must not write new data over communication buffers, or that new data will be retrieved with the requested rollback buffer address from the NIC. This may be desirable (preserving the computation) but the application's fault tolerance recovery scheme must account for retired buffers with local modifications.

An example of this fault recovery mechanism on a modern communication library like MPI would be a `MPIX_Rewind(MPI_Win window)` call that could return a MPI RMA window to a previously well known state. For a traditional timestep simulation application, this would most likely be the last completed timestep.

V. Experimental Results

In this section, we present results using two evaluation methods. First, we demonstrate the performance benefits of RVMA by instrumenting low-level networking operations on existing RDMA networks with fine-grained timers. This allows us to observe accurate timing from the network and also time individual elements of required network operations for RDMA. RVMA removes some of the required operations from the RDMA communication, namely communication setup negotiation and the requirement for additional synchronization messages on adaptively-routed networks. Using this framework we can evaluate RVMA based on only the operations necessary, using timings from existing RDMA networks. For this evaluation we use two different systems with InfiniBand networks using native Verbs and the new UCX interface.

The second evaluation method is through simulation of RDMA and RVMA networks. We use a popular architectural simulator, SST, for large scale system simulation. This simulator model lets us study the impacts of RVMA throughout a large supercomputer with common application communication behaviors through application "motifs". Motifs approximate the behavior of applications classes at large scale and have been used in simulator frameworks, like SST, by academics and industry for large scale simulations.

A. Real World Testing

In this section we investigate RVMA performance using current RDMA networks via high precision timing of RDMA operations and removal of overhead required on RDMA networks that would not be present with RVMA. We use InfiniBand networks with two different low-level interfaces, Verbs and UCX for evaluation.

1) Verbs Interface: When using traditional RDMA, the fastest method on contemporary networks is for the receiver to poll on the last byte of the message that is to arrive. This relies on in-order network message delivery (where bytes are written in order to the destination buffer). With adaptively routed networks, ordering is typically not provided. This allows applications to leverage the performance benefits of multiple routes through the network. Using adaptive routing, RDMA standards like InfiniBand mandate that a non-RDMA message (send/recv) is sent to the receiver after the RDMA traffic to indicate the RDMA message is complete. Figure 4 shows the latency penalty of RDMA versus RVMA when used according to the InfiniBand specification with the required subsequent send/recv. To measure this, we've modified a commonly used InfiniBand benchmark, OFED perftest, by adding a 1-byte send/recv after the RDMA put operation to signal completion on adaptively routed networks. These results in represent the average of 10 runs each performing 1,000 iterations. Both RVMA and RDMA results use the low-level Verbs interface for this test with Intel OmniPath InfiniBand 100Gbps with Intel Skylake Processors (platinum 8160 Xeon). RVMA performance is clearly superior to that of RDMA when it needs to use the extra send/recv for correct operation on dynamically-routed networks. We observe up to 65.8% reduction in latency by leveraging RVMA's completion semantics. In contrast, RVMA provides performance comparable to current statically-routed RDMA latency regardless of network routing (static or adaptive). It is worth noting that these tests do not account for the performance benefit that may be realized by using multiple paths through the network via adaptive routing.

2) UCX Interface: UCX is a newer interface for InfiniBand networks and promises higher performance for some operations. In order to assess if this performance gap is significant enough to impact results for RVMA we developed a UCX test with timing like that of the Verbs test in Section V-A1. These tests use UCP calls (UCX's protocol layer) to transfer data. This test, like the Verbs test, uses RDMA InfiniBand specification compliant data exchange for operations with send/recv completion. This is achieved by adding a send/recv call after the put operation to signal completion in the RDMA case, while RVMA's completion mechanism does not require the extra send/recv. UCX results were run on a ARM ThunderX2 CN9975 powered node with Mellanox ConnectX-5 EDR InfiniBand NICs. The UCX version used was 1.9.0 revision 945a9a8. The results of these RDMA vs. RVMA tests are shown in Figure 5. These results represent the average of 10 runs, each performing 100,000 iterations. Error bars, represent standard deviation between the runs. Again, observe that RVMA performs significantly better under these circumstances than RDMA. In this case, we observe a 45.8% reduction in latency by leveraging RVMA's completion semantics.

To measure the other benefits of RVMA, we implemented a test that measures the setup overhead of RDMA buffers. Setup overhead consists of an exchange of registered buffer address and length when an RDMA buffer is shared. These tests show RVMA vs. RDMA for statically-routed networks for a current state of the practice comparison. As with many applications, these microbenchmarks reuse buffers which leads to amortization of the set-up costs. In order to study this we analyzed the overheads of buffer setup to determine the number of data exchanges needed to amortize setup cost to within the margin of error for our latency tests (3%). Figure 6 shows that a large number of exchanges are needed to amortize away setup costs. This figure shows the results for both current static-routing approaches and adaptive-routing approaches. This large number of exchanges is common in microbenchmarks, but less so in typical applications. This highlights another benefit of RVMA, allowing the data transfer to begin without the initial buffer coordination required by RDMA.

Both the verbs and UCX results highlight a performance issue with RDMA's notification scheme that RVMA solves. While the main benefit of RVMA is the capabilities that it enables, this demonstrates it can also out perform RDMA on adaptively routed networks.

B. Large Scale Simulation with SST

Real world testing has limitations with regards to scale, network topology, and routing. Therefore, to investigate the benefits of RVMA at data-center scale, we have developed an RVMA network model in the Structural Simulation Toolkit (SST) environment. The RVMA model was built in SST to accommodate the new communication characteristics required to meet the specification of RVMA. A new RDMA model was also built that allows for comparison between RVMA and RDMA at key points. This work was done in collaboration with the SST team including the original authors of the networking simulation layers. The networking layers above RVMA in SST have been validated with real hardware and SST is used by several major hardware vendors to simulate next generation hardware designs and scale out simulations on current generation hardware. While it shouldn't impact the results, these simulations were run on a Cray XC30 cluster, the same cluster architecture used in the Cori and Trinity supercomputers.

The new RVMA and RDMA models for SST both use the identical timing for non-RDMA related traffic considerations. Both models use a PCIe latency of 150ns, meant to balance bus latencies between PCIe Gen 4 and Gen 5. With PCIe Gen 6 set to have much lower latencies (tens of nanoseconds roundtrip times) to the local NIC may no longer be significant compared to the time to travel over the network cables themselves. RVMA reduces required traffic over network links for coordination between sender and receiver and PCIe latency is modeled as a major contributor in our simulations. The results for current PCIe generations are therefore a conservative modeling of RVMA's future impact.

1) Motifs: In addition to studying the latency improvements when using RVMA in our previous simulations and real-world testing, we can also consider large scale simulations with SST motifs. Motifs are behavioral representations of common computation and communication patterns in HPC applications. Each motif was run for 262,144 cores split up as 8,192 nodes with 32 cores per node. All motifs were sized appropriately for such a system and use minimal compute to compare the impact of communication with medium to large size messages. Each motif was run for the default number of iterations using multiple real nodes for simulation performance and memory space requirements. SST and its motifs are used by many vendors for simulation of their large systems and we have kept simulation parameters for non-RVMA components at their suggested values.

The first motif that we will look at is Sweep3D. The Sweep3D motif is a 3D communication sweep exchange representing a "wave" of communication happening over all of the processes. This motif includes calculations on the data throughout the sweep and is mostly latency sensitive. Figure 7 compares RVMA and RDMA over a variety of different networks topologies and routing strategies. We can observe that the main differences in performance come from the use of the RVMA technique versus RDMA, with most routing techniques remaining similar in performance. The performance benefits come mainly from the receiver managing data buffers in RVMA such that the sender does not need to tightly coordinate data transfers; it simply sends the data when it is available and does not need to notify the receiver of completion as the number of expected incoming operations is known *a priori*. There are no performance overheads incurred by using RVMA versus RDMA. There is a small additional amount of memory needed on the RVMA NIC to support the address lookup. A reasonable address lookup capacity would add less than 0.1% to the manufacturing cost over a RDMA NIC.

Overall, we find that regardless of network topology, RVMA on adaptively routed networks is the clear winner. RVMA outperforms network speed for contemporary networks by at least 2X and for future link speeds like 2Tbps, RVMA outperforms by 4.4X in an adaptively routed dragonfly topology. The average speed up across all topologies and network speeds we tested is 3.56x.

For each of the bandwidths included in Figure 7, the corresponding switch crossbar bandwidths have been scaled as well. Crossbar bandwidth is always 50% greater than link bandwidth. Host bus bandwidth is always sufficient to keep the NIC/link supplied with data at line rate. As SST is a discrete

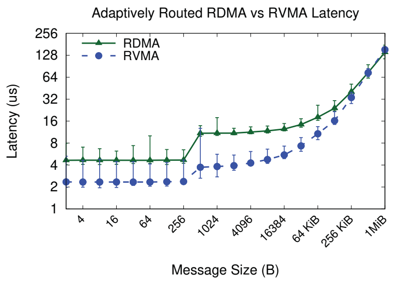

Fig. 4: RVMA vs. RDMA Latency (Verbs)

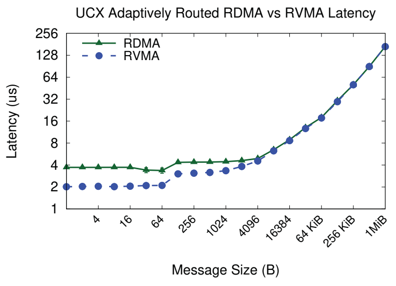

Fig. 5: RVMA vs. RDMA Latency (UCX)

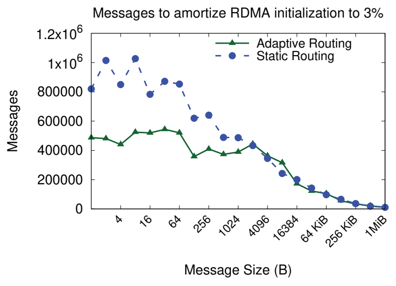

Fig. 6: UCX Amortization Analysis

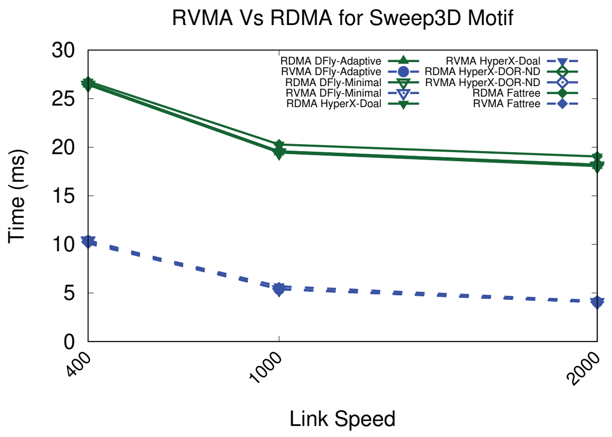

Fig. 7: RVMA vs RDMA using Sweep3D

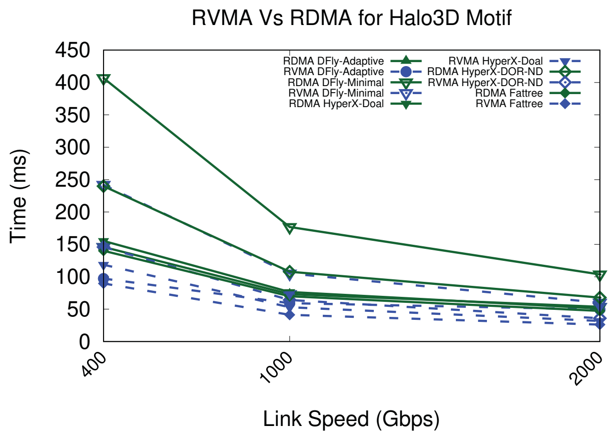

Fig. 8: RVMA vs RDMA using Halo3D

time event simulator, we chose a high update frequency for network packets to ensure high fidelity of data arrivals/processing times (5 billion updates per simulated second). We ensured that full queue stalls were not a constraining factor in the simulation by providing ample queue depths on the simulated NIC.

The Halo3D motif simulates a very common scientific computation and communication pattern for a 3D simulation. Each node in the simulated system is assigned a 3D block of a larger simulation space. Neighboring nodes exchange data on the faces of their blocks that are needed for calculations in adjacent simulation blocks. Halo3D communication exchanges benefit from high bandwidth and a low number of network hops to reach neighbor nodes. For Halo3D, network topology is much more important for performance as seen in Figure 8. For comparable network topologies, RVMA outperforms RDMA consistently with an average speedup of 1.57x across all of the topologies and network speeds we tested. The best case scenario is for HyperX Dimension Order Routing topology, where for 400Gbps networks RVMA has a 1.64X speedup over RDMA and for future 2Tbps networks RVMA sees a 1.89X speedup.

VI. Related Work

RDMA has been extensively applied in previous work including communication libraries [3] and middleware like memcached [4] and HDFS [5]. In addition to these examples, a vast number of research papers apply RDMA to problems to increase performance. This large body of work is a by-product of how hard RDMA is to use, as it justifies research publications in the simple application of the technique. Several major companies provide libraries to abstract away RDMA details from their developers, trading some performance for usability [6], [7]. An alternative to native RDMA use is programming for another higher level standard interface and building libraries to translate between the higher level and RDMA. Such solutions have been proposed such as Rsockets [8] which uses traditional RDMA and libraries that take advantage of proposed RDMA features to provide higher performance [9]. Other approaches have been proposed to provide RDMA performance while providing two-sided type communication semantics like those for MPI [10] where it also reduces message processing [11] and approaches for GPUs [12]. All of these approaches illustrate that RDMA requires very high levels of expertise to use correctly.

The lack of a target-side completion mechanism has been a open challenge in RDMA research for over a decade. Several techniques have been proposed including RDMA Write-record [9], [13] that creates a log of successful RDMA operations at the target. Notification schemes have also been proposed to solve this problem such as Put with notify [14] that provide events when puts complete. Some solutions for notifications exist in modern network implementations like InfiniBand's write with immediate operation, but they only support very small payload sizes. Audit [15] is a software solution for notification of put/get operations by tracking put/get operation counts on a source and target basis attached to registered memory regions. This differs from previous attempts at RDMA notification as it tracks completions on a memory region basis and uses a hierarchical tree to organize memory regions and sub-regions. Using RDMA in virtualized environments is a challenging problem that has seen recent work in containers [16].

Using a counter to determine if operations on an RDMA buffer are complete has been proposed in several APIs. IBM's LAPI [17] interface supported count variables which were incremented when an event occurred, and could be interpreted to determine when a message was received. The Portals API [1] has supported counters for triggered operations as early as version 4.0 [18]. sPIN [19] uses Portals message matching to launch small compute kernels on incoming packets. An advanced general compute proposal built off of Portals, INCA [20] uses counters and triggered operations for general compute and added functionality to counters to enable compute loops and jumps.

RDMA performance studies like the results for our microbenchmarks have also been done [21], [22], including performance studies using RDMA with sockets like the Sockets Direct Protocol [23]. Other performance studies look at network vendor proprietary sockets to network primitives libraries [24]. Solutions like Sockets Direct Protocol were early efforts to translate sockets code to RDMA [25].

Networks like the Cray Aries [26] can perform small RDMA transfers to unspecified buffers at a target. Portals networks like BXI [27] also support direct memory access writes to buffers without pre-negotiation using complex message matching and steering mechanisms. Such solutions use traditional event queue notification or other traditional RDMA completion semantics.

VII. Conclusion

In this paper we presented Remote Virtual Memory Access (RVMA), a new remote memory data transfer technique for networks. RVMA provides significantly improved performance over RDMA for adaptively routed networks with simulated application motif performance up to 4.4X better than RDMA while providing several new useful features. RVMA does not require byte-level ordering requirements like RDMA for efficient use and provides a lightweight completion mechanism for efficient host-side communication notification. It can provide a fault-tolerance mechanism via its multi-epoch support that provides hardware-level fault recovery for communication buffers, solving a long standing problem for RDMA networks. Finally, RVMA provides a programming interface more similar to popular two-sided communication libraries than a traditional remote memory network interface would provide, making it easier to program for.

References

[1] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, T. Hoefler, A. B. Maccabe, and T. Hudson, *The Portals 4.2 Network Programming Interface*, Sandia National Laboratories, November 2018, technical Report SAND2018-12790.

[2] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi, "Nonblocking epochs in MPI one-sided communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 475–486.

[3] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[4] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance RDMA capable interconnects," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 743–752.

[5] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance RDMA-based design of HDFS over InfiniBand," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[6] T. Ma, T. Ma, Z. Song, J. Li, H. Chang, K. Chen, H. Jiang, and Y. Wu, "X-RDMA: Effective RDMA middleware in large-scale production environments," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[7] Z. Li, N. Liu, and J. Wu, "Toward a production-ready general-purpose RDMA-enabled RPC," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019, pp. 27–29.

[8] S. Hefty, "Rsockets," in *2012 OpenFabris International Workshop, Monterey, CA, USA*, 2012.

[9] R. E. Grant, M. J. Rashti, A. Afsahi, and P. Balaji, "RDMA capable iWARP over datagrams," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 628–639.

[10] R. E. Grant, M. G. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned multithreaded MPI communication," in *International Conference on High Performance Computing*. Springer, 2019, pp. 330–350.

[11] W. Schonbein, M. G. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring multithreaded message matching misery," in *European Conference on Parallel Processing*. Springer, 2018, pp. 480–491.

[12] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized broadcast for deep learning workloads on dense-GPU InfiniBand clusters: MPI or NCCL?" in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018, pp. 1–9.

[13] M. J. Rashti, R. E. Grant, A. Afsahi, and P. Balaji, "iWARP redefined: Scalable connectionless communication over high-speed ethernet," in *High Performance Computing (HiPC), 2010 International Conference on*. IEEE, 2010, pp. 1–10.

[14] R. Belli and T. Hoefler, "Notified access: Extending remote memory access programming models for producer-consumer synchronization," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 871–881.

[15] A. Hori, J. Lee, and M. Sato, "Audit: A new synchronization api for the get/put protocol," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1464–1470, 2012.

[16] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual RDMA networking for containerized clouds," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 113–126.

[17] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and experience with lapi-a new high-performance communication library for the ibm rs/6000 sp," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE, 1998, pp. 260–266.

[18] Barrett, B.W. and Brightwell, R. and Hemmert, S. and Pedretti, K. and Wheeler K. and Underwood, K.D. and Reisen, R. and Maccabe, A.B., and Hudson, T., *The Portals 4.0 network programming interface*, Sandia National Laboratories, November 2012, technical Report SAND2012-10087.

[19] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming processing in the network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–16.

[20] W. Schonbein, R. E. Grant, M. G. Dosanjh, and D. Arnold, "INCA: in-network compute assistance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–13.

[21] R. E. Grant, P. Balaji, and A. Afsahi, "A study of hardware assisted IP over InfiniBand and its impact on enterprise data center performance," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 144–153.

[22] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 437–450.

[23] R. E. Grant, M. J. Rashti, and A. Afsahi, "An analysis of QoS provisioning for Sockets Direct Protocol vs. IPoIB over modern InfiniBand networks," in *Parallel Processing-Workshops, 2008. ICPP-08. International Conference on*. IEEE, 2008, pp. 79–86.

[24] "Myricom datagram bypass layer documents," https://www.ariacybersecurity.com/support/, (Accessed on 10/14/2020).

[25] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets Direct Protocol over InfiniBand in clusters: is it beneficial?" in *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004*. IEEE, 2004, pp. 28–35.

[26] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[27] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos, "The BXI interconnect architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 18–25.