

Correct, Fast Remote Persistence

Sanidhya Kashyap*
Georgia Institute of Technology

Dai Qin*
University of Toronto

Steve Byan Virendra J. Marathe
Oracle Labs

Sanketh Nalli
Oracle

Abstract

Persistence of updates to remote byte-addressable persistent memory (PM), using RDMA operations (RDMA updates), is a poorly understood subject. Visibility of RDMA updates on the remote server is not the same as persistence of those updates. The remote server’s configuration has significant implications on what it means for RDMA updates to be persistent on the remote server. This leads to significant implications on methods needed to correctly persist those updates. This paper presents a comprehensive taxonomy of system configurations and the corresponding methods to ensure correct remote persistence of RDMA updates. We show that the methods for correct, fast remote persistence vary dramatically, with corresponding performance trade offs, between different remote server configurations.

1 Introduction

Byte addressable persistent memory (PM) such as Intel and Micron’s 3D XPoint™ [1] promises to make local persistence faster than the state-of-the-art NAND flash by orders of magnitude. Furthermore, PM’s *byte addressability* promises to fundamentally change the way applications represent and manage persistent data. At the same time, modern *Remote Direct Memory Access* (RDMA) network fabrics (InfiniBand [15, 8], RoCE [7], and iWARP [9, 11, 30, 31, 35]) are bringing network access latencies down to singleton microseconds. They offer similar byte addressable remote memory access. The confluence of these two technologies in enterprise and distributed systems, where high availability is critically important, is inevitable.

Some early work [21, 42] has indeed demonstrated the synergistic performance benefits RDMA and PM can deliver to distributed, highly available applications. However, little is understood on how persistence of RDMA updates to remote PM can be correctly achieved. For instance, mere receipt of a completion notification of a RDMA WRITE to remote PM does not necessarily mean that the WRITE has persisted on the remote PM. In fact, persistence of remote PM updates, using RDMA operations, truly depends on the configuration of the remote server. Absent the recipe to correctly persist a remote update, serious consistency problems can emerge in distributed applications in the face of failures.

To the best of our knowledge, there is no comprehensive

analysis on system configurations and their implications on methods to correctly, and efficiently, persist updates using RDMA operations. Douglas [13] provides an enumeration of some remote server configurations and corresponding remote persistence recipes. However, it lacks comprehensiveness. Furthermore, certain system configurations assumed in Douglas’ categorization are not even relevant to today’s state-of-the-art system support for PM [32]. Recent effort by the InfiniBand Trade Association (IBTA) standards community proposes extensions to RDMA operations to enable remote persistence [10, 28]. However, we show that the proposed extensions cover correct, fast remote persistence for a somewhat narrow set of remote server configurations.

This paper presents a comprehensive taxonomy of remote server configurations that has significant implications on the methods required to correctly persist RDMA updates to the remote server’s PM. Our taxonomy breaks down configurations along three axes: (i) the notion of a *persistence domain* in a system – the portion of the memory hierarchy and RDMA-capable Network Interface Card (RNIC) buffers that are effectively persistent; (ii) enablement of optimizations to direct RDMA updates to the remote server’s processor cache [4, 5], also called *Data Direct I/O* (DDIO) by Intel [12]; and (iii) location of receive queue work request buffers (RQWRBs) of RDMA connection endpoints, called *Queue Pairs* (QPAIRS), on the remote server – either in the server’s DRAM or its PM.

We incorporate not only existing RDMA operations in our analysis (RDMA WRITE, RDMA WRITEIMM, and RDMA SEND), but also operations and extensions newly proposed by the IBTA standards community [10, 28] – RDMA FLUSH and *non-posted* RDMA WRITE.

Our thorough analysis of the server configuration space and RDMA operations has led us to 10 distinct methods for remote persistence of singleton RDMA updates. Our analysis leads us to some interesting, and surprising, methods of remote persistence that let clients treat the traditionally two-sided RDMA SEND operation as a one-sided RDMA operation.

We also examine correct persistence of *compound* updates. We try to address the following question: What is the way to correctly, and efficiently, persist causally dependent updates on the remote server’s PM? (Much of the IBTA standards community discussion has revolved around this question [10, 28].) We find 9 additional methods for correct and

*Work done when author was at Oracle Labs.

efficient remote persistence for compound updates. Through our analysis we precisely pin-point the correct and efficient method of remote persistence for each of the combined 72 different scenarios considered in our work. The programmer must carefully apply the correct remote persistence method for a given remote server configuration. Application of an incorrect persistence method may lead to worse performance, or even critical data inconsistencies in the face of failures.

We evaluate our methods of remote persistence using a ubiquitous workload that manifests in most distributed and highly available systems – *log replication* (REMOTELOG). REMOTELOG serves as the test bed for both singleton and compound RDMA update persistence. Our evaluation shows significant performance trade offs between the various methods of remote persistence, with a general indication that, for REMOTELOG, remote persistence via one-sided RDMA operations performs significantly better than a message passing based approach using RDMA SENDs.

We first present some background in §2 that describes pertinent system support for PM, the high level architecture of a system that participates in a RDMA network, various pertinent RDMA operations (current, and new ones proposed by the IBTA standards community [10, 28]), and recent technological advancements that have significant implications on remote persistence. In §3, we describe our remote persistence taxonomy and its implications on persisting RDMA updates to PM. We present a detailed qualitative analysis of the steps needed for remote persistence. Our taxonomy’s methods of remote persistence are based in part on recent RDMA extensions proposed by the IBTA standards community [10, 28]. We treat persistence of *singleton* and *compound* remote updates separately. Our preliminary evaluation in §4 indicates significant performance trade offs between the various methods of remote persistence.

2 Background

System Support for PM: PM supported systems are expected to become available imminently [27]. In these systems, PM will be available in the DIMM form factor, alongside traditional DRAM DIMMs as shown in Figure 1. Applications will be able to perform *load*, *store*, and other memory access instructions on PM that are typical of DRAM. Furthermore, processor vendors such as Intel [16] and ARM [6] are providing extensions to enable ordering of persistence of *stores* to PM – e.g. Intel’s *clflush-opt* and *clwb* instructions, and extensions to existing instructions with fence semantics to enforce completion of prior cache line flushes and writebacks (ARM has similar extensions [6]). Cache line flushes, writebacks and memory fences guarantee that updated cache lines are evicted or copied to at least the Integrated Memory Controller (IMC) buffers. IMC buffer entries are drained out to the PM DIMMs as scheduled by the memory controller. During a power failure, hardware features such as *Asynchronous DRAM Refresh (ADR)* [3, 32] are used to drain the IMC

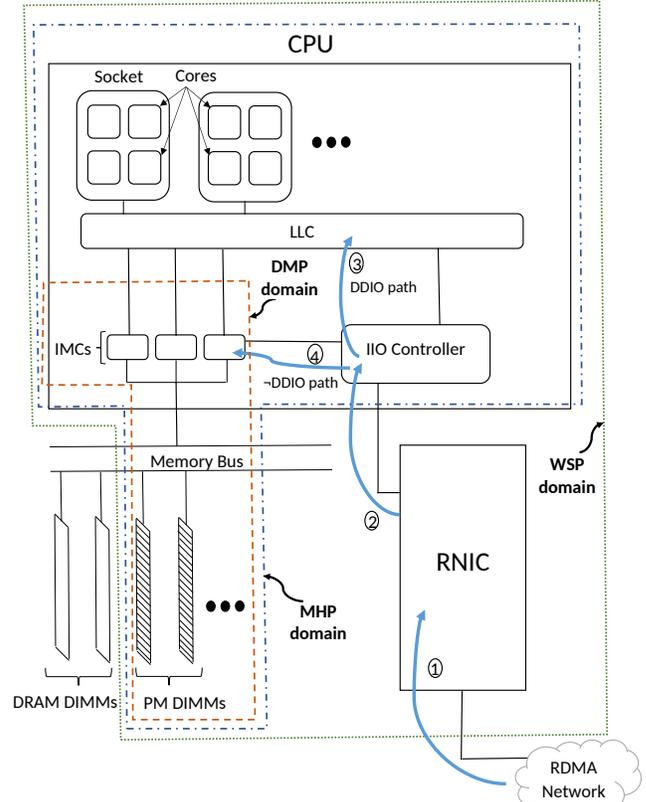


Figure 1: Block Diagram of a computer connected to an RDMA network fabric.

buffers to PM DIMMs. PM is already available in the industry in the form of NVDIMM-N type DIMMs [18] – DRAM backed by NAND flash using ultra capacitors [2, 24, 39].

Networked Computer Architectures: RNICs are becoming increasingly common in modern data centers. Figure 1 depicts the high level architecture of today’s data center computers connected to each other via RDMA network fabrics. A computer communicates with another via the RNIC. The RNIC itself has internal buffers that can host data recently sent or accessed by a remote node. These buffers however are not coherent with the host processor’s cache [37].

We now discuss the data flow between different parts of the system caused by RDMA accesses (shown in Figure 1). We focus on RDMA WRITES here, but the same details apply to RDMA SENDs, READs, and ATOMICs. An incoming RDMA WRITE first lands in the RNIC buffers (step 1 in Figure 1). Thereafter, the RNIC posts (via DMA) the WRITE to the server’s memory. The data itself travels through the processor’s PCIe I/O controller. The I/O controller was traditionally a separate off-die chip between peripheral I/O devices and the processor. However, in recent years, processors have the *Integrated I/O (IIO)* feature that hosts the I/O controller on the processor’s die for a much more efficient data path between peripheral I/O devices (e.g. RNICs) and the processor. IIO is prevalent in modern processors, and has its private

independent buffers.

The RDMA WRITE’s data in practice moves from the RNIC buffers to the IIO controller buffers (step 2), after which it is moved to the target memory DIMMs via the pertinent memory controllers (step 4). The received data can then be accessed by the server’s processor. This necessarily leads to a high latency memory bus transaction in the server to fetch the recently received data. Recognizing these overheads, Intel introduced the notion of *Data Direct I/O* (DDIO) in its processors that provides a data path from the IIO directly to the server processor’s L3 cache, rather than placing the data in IMC buffers [12] (step 3). A similar feature exists in processors by other vendors such as ARM [4, 5].

With DDIO, all incoming RDMA WRITES are likely to directly reach the server processor’s L3 cache; in high traffic scenarios, writes may partially be written directly into the server’s memory DIMMs. DDIO is a standard feature in modern Intel processors, and can be switched off to avoid cache pollution if the application hosted on the server is not expected to access recently received data through its peripheral I/O devices. As we will show in §3, DDIO plays an important role in remote persistence.

RDMA Operations: The RDMA programming model [23, 31, 34] contains a QPAIR abstraction that is used to represent an endpoint of a communication channel (connection) between computers (nodes) over and RDMA network. A connection management system establishes a connection between QPAIRS of communicating nodes. This includes designating the type of the connection: Unreliable Datagram (UD), Unreliable Connection (UC), or Reliable Connection (RC). Since reliability (messages are not lost) is critical to remote persistence, we assume reliable connections in the rest of the paper unless mentioned otherwise. All RDMA data operations (detailed below) can be used in reliable connections. In the rest of this paper, we call the remote server that receives RDMA requests the *responder*, and the requesting server is called the *requester*.

RDMA operations contain a “two-sided” message passing operation – the RDMA SEND. The responder must receive and process the sent message and potentially provide a response to the requester via another RDMA SEND message directed to the requester. The responder’s processor performs these tasks, which may not be desirable for some applications [14]. The “one-sided” RDMA operations – RDMA READ and RDMA WRITE – do not require any participation from the responder’s CPU; these operations read and write remote memory respectively (the responder’s RNIC uses DMA to perform the reads and writes). A hybrid RDMA WRITE with *Immediate Data* (WRITEIMM) operation performs a WRITE on the responder’s memory and delivers a message to the responder with a 32-bit sized payload – the *immediate data*. RDMA WRITEIMM does require the responder’s CPU to process the message with immediate data, like RDMA SEND. RDMA also contains one-sided ATOMIC operations

(RDMA *Fetch-And-Add* (FAA) and RDMA *Compare-And-Swap* (CAS)) – 64-bit wide atomic read-modify-write operations that can be used for synchronization between remote requesters [34, 41], but can incur significant performance overheads [19].

All RDMA operations described above are asynchronous. The requester posts a RDMA *work request* at a QPAIR and, if the request was flagged to return a completion notification, can subsequently wait on a *completion notification* for the request. The notification is generated at the requester once the operation completes. All operations that semantically generate a response (RDMA READ, FAA, and CAS) necessarily generate a completion notification. Work requests for other operations (RDMA SEND, WRITE, and WRITEIMM) can optionally be flagged for completion notification. An application can use the RDMA ordering rules to infer completion of operations that are not flagged for completion notification.

More recently [10, 28], the IBTA standards community decided to add a new RDMA operation – the RDMA FLUSH. This is because simply receiving a completion notification, at the requester, for a RDMA WRITE/SEND/WRITEIMM does not guarantee that the operation has become *visible* (available in the memory hierarchy) at the responder’s end – the operation may still reside in the responder’s RNIC buffers, or in the case of iWARP [9, 11, 30, 31, 35], may not even have left the requester. As we will see in §3, a failure at the responder at such a juncture could lead to loss of data residing in the responder’s RNIC buffers in some system configurations. The new RDMA FLUSH operation guarantees that all prior RDMA update operations, issued on the same connection, become visible to the responder before the FLUSH’s completion notification is received at the requester.

RDMA Operation Ordering: The RDMA standard specifies interesting ordering rules for RDMA operations [8, 31, 34]. In particular, it separates RDMA operations into two categories: (i) operations that produce a return value that is consumed by the requester (RDMA READ, CAS, FAA, and now, FLUSH), informally referred to as “non-posted” operations in the community [10, 28]; and (ii) operations that do not require a return value (RDMA SEND, WRITE, and WRITEIMM), also referred to as “posted” operations.

The effects of non-posted operations are totally ordered with all prior operations at the responder – their effects are made visible at the responder in the order they were issued at the requester. Effects of posted operations are totally ordered with each other. However, posted operations can be ordered at the responder *before* prior non-posted operations issued by the requester.

Visibility of updates to PM on a local system does not imply persistence of those updates [20, 29, 38]. Similarly, visibility of RDMA updates does not imply persistence of those updates. As a result, though RDMA updates become visible in-order on a reliable connection, they may become persistent out-of-order. RDMA FLUSH was introduced by

IBTA precisely to enforce order of persistence. However, since RDMA FLUSH is a non-posted operation, a subsequent posted RDMA operation can be ordered *before* the FLUSH leading to the out-of-order persistence problem. To address this problem, IBTA decided to add an ATOMIC version of RDMA WRITE [10, 28] that guarantees atomicity semantics similar to the other ATOMIC operations (FAA and CAS). In effect, an ATOMIC WRITE ($\text{WRITE}_{\text{atomic}}$) acts as a non-posted operation – it is ordered, at the responder, *after* all preceding posted and non-posted RDMA operations issued on the same connection. Furthermore, $\text{WRITE}_{\text{atomic}}$ can be used to update up to 8-bytes atomically (all or nothing semantics).

Ordering enforcement between posted and non-posted RDMA operations is not a new problem. The pre-existing solution in the RDMA standard is to tag posted RDMA operations with the *fence* flag. This flag ensures that the fenced operation blocks at the requester’s end until all prior non-posted operations on the same QPAIR complete – their result is received at the requester’s end. Such fenced posted operations can be used in conjunction with RDMA FLUSH to enforce ordering of persistence of two consecutive RDMA updates. However, it incurs the overhead of an extra round-trip for the RDMA FLUSH, that the second RDMA update must wait for. $\text{WRITE}_{\text{atomic}}$ eliminates that extra round-trip, and enables pipelining of the dependent updates (separated by an intervening RDMA FLUSH).

3 Taxonomy of Remote Persistence

3.1 Remote Server Configuration

Persistence on the responder’s end of RDMA based remote updates depends on several characteristics of the responder’s configuration. We first describe these characteristics that enable us to build an elaborate taxonomy of server configurations that have significant implications on methods to perform remote update persistence on the responder. We break down the responder’s configuration along three axes: (i) the notion of a *persistence domain* in a system [22, 33]; (ii) enablement of optimizations to direct RDMA updates to the remote server’s processor cache (e.g. DDIO in Intel’s processors); and (iii) location of work request buffers in the receive queue of RDMA QPAIRS on the responder – either on its DRAM or its PM.

3.1.1 Persistence Domains

Prior work [22, 33] has defined a persistence domain as the portion of the memory hierarchy that is effectively persistent – writes reaching this part of the memory hierarchy are guaranteed to persist across power failure and restart cycles. We extend that definition to include the RNIC’s buffers as well. We observe three distinct persistence domains based on characteristics of a system’s configuration.

1. *DIMM and Memory controller Persistence (DMP)*: This persistence domain includes the PM DIMMs as well

as the integrated memory controller (IMC) buffers. The IMC buffers are included in the persistence domain through hardware features such as Asynchronous DRAM Refresh (ADR) [32, 33] that flush the IMC buffers to the PM DIMMs during a power failure event. This is expected to be the dominant type of persistence domain in at least the near term.

2. *Memory Hierarchy Persistence (MHP)*: This persistence domain includes the entire memory hierarchy of the system [17, 26, 33] – all processor caches, store buffers, etc. Here the assumption is that the system has enough residual power to flush processor buffers and caches to the PM DIMMs on a power failure event. Custom system hardware and power supply equipment may be required to make this possible [33]. MHP has significant implications on the programming model for PM since now the visibility of `store` instructions implies persistence – there is no need for explicit cache line flushes and write-backs and corresponding persistence barriers [20, 29]. The RNIC buffers are however not included in the persistence domain. As a result, there is still a need to perform RDMA FLUSHes.
3. *Whole System Persistence (WSP)*: This persistence domain encompasses the entire system, including the RNIC buffers. Battery backed systems can serve WSP [25]. On a power failure, the caches of the processor as well as the buffers of all peripheral devices will be flushed to corresponding regions in the PM DIMMs. Since RNIC buffers become effectively persistent, WSP has interesting implications on the programming model for remote persistence in that it eliminates the need for explicit ordering barriers between posted RDMA operations. In WSP, the normal RDMA ordering semantics now apply to persistence, once the data has been received by the responder RNIC.

Figure 1 depicts DMP, MHP, and WSP. We do not include a persistence domain limited to just the PM DIMMs [13, 22] since such persistence domains are unlikely to be supported in real systems [32].

3.1.2 Implications of DDIO / Cache Stashing

DDIO [12] is the feature on Intel processors that enables delivery of incoming RDMA data from the responder’s RNIC directly into the L3 cache. A similar feature called *cache stashing* is provided by ARM processors [4, 5]. While it does provide the processor much more efficient access to incoming data, for a system that supports just DMP, DDIO ends up keeping the inbound data outside the persistence domain, that is, in the processor cache. Extra work needs to be done by the responder’s processor to flush/writeback the arrived data to the persistence domain. Alternately, if DDIO is turned off at the responder, the inbound data will move to the memory

controller, which is within the DMP domain. If the responder supports MHP, arrival of the inbound data in the responder’s L3 cache implicitly persists the data. For WSP, arrival of inbound data in the RNIC buffers itself ensures persistence.

3.1.3 RDMA Queues and Work Requests

Each QPAIR used for a RDMA connection internally contains two queues – the *send queue* and the *receive queue*. Each of these is a linked list of *work requests (WRs)* created by the enclosing application. Each work request itself points to a *work request buffer (WRBs)* created by the application. The work request buffer contains application specific data. For the send queue, one or more work request buffers can be associated with a work request. The more pertinent queue for persistence of RDMA updates is the receive queue. The responder must preallocate work requests (and their buffers) in a QPAIR’s receive queue in order to receive work requests from the requester. These work request buffers are completely under the control of the application, which can allocated them from either DRAM or PM. As we shall see later, these allocation choices have significant implications on remote persistence of RDMA SEND requests to the extent that they can be treated by the requester like one-sided RDMA updates.

We note that a QPAIR also is associated with a *Completion Queue (CQ)* that contains requester-side, RNIC generated, completion notifications for operations explicitly marked to generate a completion notification. For a posted operation, a completion notification is generated by the requester’s RNIC immediately after the responder’s RNIC receives the operation – this is triggered from the lower level RDMA transport layer, when the requester’s RNIC receives an acknowledgment from the responder’s RNIC for the receipt of the operation at the responder’s end. As noted earlier, for a non-posted operation, a completion notification is generated at the requester only after the return value of the non-posted operation is received from the responder.

Receipt of inbound two-sided RDMA operations (RDMA SEND and WRITEIMM) results in creation of a *receive completion* notification in the QPAIR’s CQ. This receive completion is however generated *after* the corresponding receive queue work request, and its corresponding work request buffer, is populated and made visible by the RNIC.

Our analysis above leads us to twelve distinct remote server configurations as shown in [Table 1](#).

3.2 Persisting Singleton RDMA Updates

We first consider the case of singleton remote updates comprising update of just one contiguous block of data in the responder’s PM. The block can range from $1 - 2^{31}$ bytes in size, the permissible size for a RDMA SEND, WRITE, and WRITEIMM on a reliable connection. [Table 2](#) depicts the full taxonomy of remote persistence of singleton remote updates for the twelve different responder configurations enumerated in [Table 1](#). We break down our discussion of remote persistence by the persistence domain configured on the responder.

Config	Explanation
DMP + DDIO + →	DMP, with DDIO turned on, and RQWRB placed in DRAM.
DRAM-RQWRB	
DMP + DDIO + →	DMP, with DDIO turned on, and RQWRB placed in PM.
PM-RQWRB	
DMP + -DDIO + →	DMP, with DDIO turned off, and RQWRB placed in DRAM.
DRAM-RQWRB	
DMP + -DDIO + →	DMP, with DDIO turned off, and RQWRB placed in PM.
PM-RQWRB	
MHP + DDIO + →	MHP, with DDIO turned on, and RQWRB placed in DRAM.
DRAM-RQWRB	
MHP + DDIO + →	MHP, with DDIO turned on, and RQWRB placed in PM.
PM-RQWRB	
MHP + -DDIO + →	MHP, with DDIO turned off, and RQWRB placed in DRAM.
DRAM-RQWRB	
MHP + -DDIO + →	MHP, with DDIO turned off, and RQWRB placed in PM.
PM-RQWRB	
WSP + DDIO + →	WSP, with DDIO turned on, and RQWRB placed in DRAM.
DRAM-RQWRB	
WSP + DDIO + →	WSP, with DDIO turned on, and RQWRB placed in PM.
PM-RQWRB	
WSP + -DDIO + →	WSP, with DDIO turned off, and RQWRB placed in DRAM.
DRAM-RQWRB	
WSP + -DDIO + →	WSP, with DDIO turned off, and RQWRB placed in PM.
PM-RQWRB	

Table 1: Remote server configurations. RQWRB is the Receive Queue Work Request Buffer.

DMP: The DMP domain comprises just the PM DIMMs and the host processor’s IMC buffers (see [Figure 1](#)). As a result, if DDIO is turned on at the responder, incoming RDMA updates may very well go into its processor cache, which is not a part of its DMP. As a result, the requester must send a message to the responder to flush those updates to DMP. It is clear that remote persistence is not reliably possible with just one-sided RDMA operations (RDMA WRITE and RDMA FLUSH). The alternative shown in [Table 2](#) is for the requester to perform an RDMA WRITE followed by a RDMA SEND that informs the responder that a WRITE has just happened, and needs to be flushed out of the responder’s processor cache. After performing the flush, the responder sends back a response to the requester informing completion of remote persistence. Remote persistence is guaranteed to have happened from the requester’s perspective when it receives the response.

With RDMA SEND we assume that the sent message contains the target location to update in the responder’s PM and the value that needs to be written to that location. We thus use the standard message passing idiom to enforce persistence of remote updates. It however leads to a copy of the payload on the responder’s side – from the RQWRB of the responder to the target memory location of the responder (done using local copy followed by local flush shown in [Table 2](#)).

RDMA WRITEIMM is much more elegant in that it eliminates the copying overhead (we assume that the write in WRITEIMM is directed to a location in PM). However it does require the responder’s processor to perform local flushes of the updated cache lines. Furthermore, RDMA WRITEIMM has the limitation that the target address must be identifiable using just the 32-bit immediate data embedded in the message delivered to the responder, which may not be possible in some application contexts.

Remote persistence gets more interesting when DDIO is

	DMP			MHP			WSP		
	Write	WriteImm	Send	Write	WriteImm	Send	Write	WriteImm	Send
DDIO + DRAM-RQWRB	Rq Write(a) Rq Send(&a) Rsp Receive(&a) Rsp flush(&a) Rsp Send(ack) Rq Receive(ack)	Rq WriteImm(a) Rsp Receive(&a) Rsp flush(&a) Rsp Send(ack) Rq Receive(ack)	Rq Send(a) Rsp Receive(a) Rsp copy(a) + flush(&a) Rsp Send(ack) Rq Receive(ack)	Rq Write(a) Rq Flush Rq Comp ^{Flush}	Rq WriteImm(a) Rq Flush Rq Comp ^{Flush}	Rq Send(a) Rsp Receive(a) Rsp copy(a) Rsp Send(ack) Rq Receive(ack)	Rq Write(a) Rq Comp ^{Write(a)}	Rq WriteImm(a) Rq Comp ^{WriteImm(a)}	Rq Send(a) Rsp Receive(a) Rsp copy(a) Rsp Send(ack) Rq Receive(ack)
DDIO + PM-RQWRB	As above	As above	As above	As above	As above	Rq Send(a) Rq Flush Rq Comp ^{Flush}	As above	As above	Rq Send(a) Rq Comp ^{Send(a)}
-DDIO + DRAM-RQWRB	Rq Write(a) Rq Flush* Rq Comp ^{Flush}	Rq WriteImm(a) Rq Flush Rq Comp ^{Flush}	As above	As above	As above	Rq Send(a) Rsp Receive(a) Rsp copy(a) Rsp Send(ack) Rq Receive(ack)	As above	As above	Rq Send(a) Rsp Receive(a) Rsp copy(a) Rsp Send(a) Rq Receive(a)
-DDIO + PM-RQWRB	As above	As above	Rq Send(a) Rq Flush Rq Comp ^{Flush}	As above	As above	Rq Send(a) Rq Flush Rq Comp ^{Flush}	As above	As above	Rq Send(a) Rq Comp ^{Send(a)}

RQWRB = Receive Queue Work Request Buffer
copy = local memcopy at the Responder
Rq Completion = Receipt of completion notification at requester
*On some systems, a message exchange, using RDMA SEND, can possibly be more efficient than a RDMA FLUSH.

Rq = Requester
flush = local cache line flush at the Responder
Receive = Requester/Responder receives message

Rsp = Responder

Table 2: Taxonomy for Singleton Updates (value a) using RDMA operations to location $\&a$ in the responder’s PM. Each row in the table corresponds to a remote server (responder) configuration with DDIO turned on or off and the RQWRB resident in DRAM or PM. Each column represents the primary operation used to implement the remote update (RDMA WRITE, RDMA WRITEIMM, or RDMA SEND), further grouped by the persistence domain configured on the responder – DMP, MHP, WSP. Value a is written by the requester at the responder; $\&a$ represents the address of the target PM memory block at the responder.

turned off. First, remote persistence using one-sided operations – RDMA WRITE, RDMA WRITEIMM, and RDMA FLUSH – becomes possible (we assume that RDMA WRITE or RDMA WRITEIMM updates a location in the responder’s PM). The requester must wait for the completion notification for the RDMA FLUSH, which is received only after its FLUSH has taken effect on the responder. Prior work [13, 37] has described this particular case. We note however that in some RDMA network and RNIC implementations, an RDMA FLUSH could possibly have higher latency than a two-sided message exchange using RDMA SENDS.

With DDIO turned off, if the RQWRB resides in DRAM, the method of remote persistence with RDMA SEND follows the traditional messaging passing idiom. However, if the RQWRB resides in PM an interesting possibility emerges: Since the RQWRB resides in PM, a message received in the responder’s DMP domain is effectively persistent. This message can survive power failure cycles, and the enclosing application’s recovery subsystem can be designed to access these messages and persist their effects. As a result, the requester needs to simply persist the data from RDMA SEND operations on the responder, which is done by issuing a RDMA FLUSH. In applications where processing of these persistent messages is not possible during recovery (because of additional missing context that was hosted in DRAM), the standard two-sided message exchange idiom would be the only viable alternative.

MHP: MHP refers to persistence of the whole memory hierarchy. Since the RNIC buffers are not a part of the MHP domain, RDMA FLUSH is required for remote persistence

using just one-sided operations. That is the case not only with RDMA WRITE, but also with RDMA WRITEIMM. In the latter case, we assume that application correctness is not compromised even if the immediate data delivered with the RDMA WRITEIMM is lost due to a power failure or application crash. (If synchronous immediate data delivery is required for application correctness, a different method for remote persistence, most likely based on RDMA SEND, must be used by the application.) The requester can infer persistence of remote updates when it receives a completion notification for the RDMA FLUSH.

The RDMA SEND based method for persistence of remote updates varies based on whether the RQWRB resides in DRAM or PM. If in DRAM, the method is the classic message passing based idiom. Note however that the responder’s processor does not need to issue local cache line flushes since completing local stores itself ensures persistence. If the RQWRB resides in PM, persisting just the RDMA SEND on the responder may be sufficient for the requester to infer persistence of the remote update. Again, if the application requires a synchronous handshake between the requester and responder, the classic message passing idiom may be a better match, but may come with higher latency for remote persistence.

WSP: When even the RNIC buffers become effectively persistent, persistence of remote updates may become quite simple. For the InfiniBand and RoCE RDMA transports, receipt of just the completion notification of RDMA WRITE and RDMA WRITEIMM at the requester is sufficient to infer persistence of a remote update. The same method of remote

persistence applies to the RDMA SEND based approach if the responder’s RQWRB resides in PM. If not, the classic message exchange idiom is required for remote persistence.

We note a key difference in the semantics of completion notifications between iWARP [9, 11, 30, 31, 35] and InfiniBand/RoCE [7, 8, 15]. InfiniBand and RoCE guarantee that the RDMA operation is received at least at the responder’s RNIC *before* the corresponding completion notification is generated at the requester. iWARP, however, makes a “weaker” guarantee in that a completion notification is created as soon as the operation reaches the requester’s reliable transport layer (TCP, or SCTP [9]). As a result, the completion notification may be received by the application on the requester’s side even before the operation is sent to the responder. The implication the iWARP semantics have on remote persistence for responders supporting WSP is that RDMA FLUSH becomes necessary for correct remote persistence. The methods for remote persistence for WSP essentially mimic the corresponding methods for remote persistence for MHP on iWARP.

3.3 Persisting Compound RDMA Updates

Ordering of consecutive updates is foundational to achieve data consistency. RDMA based updates are no different. We now focus on the methods programmers can use to enforce correct order of persistence of consecutive updates using RDMA operations. We want to ensure that if a requester is posting two *strictly ordered* updates, a followed by b , to the responder’s PM, those updates are persisted at the responder in the same order. *Log append* is a canonical example of such dependent updates – the log record at the remote log’s tail must first be updated and persisted, before advancing the log’s tail pointer and persisting it. Table 3 shows the recipes to enforce the correct order of persistence on the responder.

DMP: With DDIO turned on, dependent RDMA WRITES must be separated by a message exchange between the requester and responder. The responder must also flush the affected cache lines for the first RDMA WRITE to ensure its persistence before sending back an acknowledgment to the requester. Furthermore, the second RDMA WRITE must also be followed by another identical message exchange to inform the requester about persistence of the second RDMA WRITE. We observe similar set of operations required for remote persistence of consecutive RDMA WRITEIMMs; the WRITEIMM itself ends up performing a write followed by delivery of a message to the RQWRB of the responder.

RDMA SEND is perhaps the more effective way of performing remote updates since both updates (a and b) can be packaged in a single message. The responder must first write and flush a before writing and flushing b . A receipt of an acknowledgment from the responder informs the requester that the two updates have persisted on the responder’s PM. The RDMA SEND based approach does however lead to two copies of the updates at the responder’s end – in the RQWRB and the final target. As a result, the above two approaches

may be more efficient for coarse grained updates. The methods for remote persistence of dependent updates remains the same as above even when the RQWRB resides in PM since DDIO can push the remote updates to the responder’s processor cache, which the responder must flush locally to its DMP domain.

We observe interesting implications if DDIO is turned off on the responder. When the RDMA WRITE operation is used to perform the remote updates, there are two possible means of ordering the persistence of the two updates a and b . Table 3 shows just one alternative that aligns with the canonical log append example mentioned above – the second update is a fine-grain write, to the log’s tail, that atomically updates at most 8 bytes. The RDMA FLUSH after the first RDMA WRITE ensures that the write will be flushed to the responder’s PM. The use of RDMA WRITE_{atomic} for the second write ensures that, at the responder, it is ordered *after* the preceding RDMA FLUSH. The second subsequent RDMA FLUSH makes sure that the WRITE_{atomic} persists before the completion notification for the second FLUSH is received at the requester.

If the second update is more than 8 bytes long, a RDMA WRITE_{atomic} will not work. In that case, the requester must wait for the completion notification of the first RDMA FLUSH before issuing the second RDMA WRITE. As mentioned earlier, in some RDMA fabric and RNIC implementations, a message exchange based notification of completion of remote persistence (similar to the approach taken above in the case where DDIO is turned on) may perform better than the use of RDMA FLUSH.

With DDIO turned off, RDMA WRITEIMM can be used as somewhat of a one-sided operation in that the responder does not need to send back an acknowledgment message to the requester after either of the two RDMA WRITEIMMs. (We again assume that application correctness is not compromised even if the immediate data delivered with the RDMA WRITEIMM is lost due to a power failure or application crash.) However, since there is no ATOMIC version of WRITEIMM, the requester must wait for the completion notification of its first RDMA FLUSH before performing the second (dependent) RDMA WRITEIMM. Thereafter the second RDMA FLUSH and its corresponding completion notification informs the requester that the second update has also remotely persisted.

For RDMA SEND based remote updates, the method described above for the DDIO case will work correctly if the RQWRB resides in DRAM. However, if the RQWRB resides in PM, the RDMA SEND can be used like a one-sided operation. This is because using RDMA FLUSH ensures that the sent message resides in the PM location of the corresponding RQWRB at the responder. This implicitly persists the compound update, which can survive an immediate power failure at the responder. The application’s recovery subsystem can be used to find and apply the sent message to the correct

	DMP			MHP			WSP		
	Write	WriteImm	Send	Write	WriteImm	Send	Write	WriteImm	Send
DDIO + DRAM-RQWRB	Rq Write(a) Rq Send(&a) Rsp Receive(&a) Rsp flush(&a) Rsp Send(ack) Rq Receive(ack) Rq Write(b) Rq Send(&b) Rsp Receive(&b) Rsp flush(&b) Rq Send(ack) Rq Receive(ack)	Rq WriteImm(a) Rsp Receive(&a) Rsp flush(&a) Rsp Send(ack) Rq Receive(ack) Rq WriteImm(b) Rsp Receive(&b) Rsp flush(&b) Rsp Send(ack) Rq Receive(ack)	Rq Send(a,b) Rsp Receive(a,b) Rsp copy + flush(a,b) Rsp Send(ack) Rq Receive(ack)	Rq Write(a) Rq Write(b) Rq Flush Rq Comp _{Flush}	Rq WriteImm(a) Rq WriteImm(b) Rq Flush Rq Comp _{Flush}	Rq Send(a,b) Rsp Receive(a,b) Rsp copy(a,b) Rsp Send(ack) Rq Receive(ack)	Rq Write(a) Rq Write(b) Rq Comp _{Write(b)}	Rq WriteImm(a) Rq WriteImm(b) Rq Comp _{WriteImm(b)}	Rq Send(a,b) Rsp Receive(a,b) Rsp copy(a,b) Rsp Send(ack) Rq Receive(ack)
DDIO + PM-RQWRB	As above	As above	As above	As above	As above	Rq Send(a,b) Rq Flush Rq Comp _{Flush}	As above	As above	Rq Send(a,b) Rq Comp _{Send(a,b)}
-DDIO + DRAM-RQWRB	Rq Write(a) Rq Flush* Rq Write _{atomic} (b) Rq Flush Rq Comp _{Flush}	Rq WriteImm(a) Rq Flush Rq Comp _{Flush} Rq WriteImm(b) Rq Flush Rq Comp _{Flush}	As above	As above	As above	Rq Send(a,b) Rq Flush Rsp Receive(a,b) Rsp copy(a,b) Rsp Send(ack) Rq Receive(ack)	As above	As above	Rq Send(a,b) Rsp Receive(a,b) Rsp copy(a,b) Rsp Send(ack) Rq Receive(ack)
-DDIO + PM-RQWRB	As above	As above	Rq Send(a,b) Rq Flush Rq Comp _{Flush}	As above	As above	Rq Send(a,b) Rq Flush Rq Comp _{Flush}	As above	As above	Rq Send(a,b) Rq Comp _{Send(a,b)}

RQWRB = Receive Queue Buffer
WRITE_{atomic} = RDMA Atomic WRITE
Rq Comp = Receipt of completion notification at requester
Rq = Requester
copy = local memcopy at the Responder
Receive = Requester/Responder receives message
Rsp = Responder
flush = local cache line flush
*On some systems, a message exchange, using RDMA SEND, can possibly be more efficient than a RDMA FLUSH.

Table 3: Taxonomy for Compound Updates using RDMA operations. The above taxonomy orders remote persistence of two updates – a followed by b .

locations in the responder’s PM. The requester can infer that the compound update has persisted on receipt of completion notification for the RDMA FLUSH.

MHP: For MHP, visibility of RDMA updates at the responder is equivalent to persistence, as long as the updates are directed to the remote PM. Existing RDMA ordering semantics [8, 31, 34] guarantee in-order visibility of consecutive posted updates. As a result, two dependent updates can be pipelined back-to-back as RDMA WRITES. However, since the writes need to be flushed from the responder’s RNIC buffers to its memory hierarchy, a RDMA FLUSH is needed. The requester can conclude ordered remote persistence of the two WRITES upon receipt of the completion notification for the FLUSH. RDMA WRITEIMM can be treated as a one-sided operation by the requester and used in a way similar to RDMA WRITE for MHP.

For RDMA SEND, the requester can send a compound message containing both the dependent updates, which are applied (using local stores) by the responder in the expected order. No local cache line flushes are required at the responder because of MHP. However, the responder must send an acknowledgment to the requester informing the latter of persistence of the compound update. If the RQWRB resides in PM, RDMA SEND can be treated as a one-sided operation by the requester. As a result, a subsequent RDMA FLUSH followed by receipt of completion of the FLUSH is all the requester needs to infer that the compound update has persisted on the responder’s end. However, note that the requester can-

not immediately try to read the responder’s affected memory without additional coordination with the responder. If no coordination takes place, the requester might end up reading a stale value from the responder if the preceding RDMA SEND was not applied at the responder.

WSP: The RDMA reliable connection guarantees ordered delivery of update requests (RDMA WRITE, WRITEIMM, and SEND) at the responder’s RNIC. As a result, with RDMA WRITE based updates, the requester can simply post the WRITE requests in the expected order. The requester can assume remote persistence of the two updates on receipt of a completion notification of the second RDMA WRITE. RDMA WRITEIMM can be treated like a one-sided operation for WSP, allowing simple back-to-back issuance of WRITEIMMs for ordered remote updates. Again the requester simply needs to wait for the completion notification for the second WRITEIMM to infer remote persistence. Similar operation sequence can be used with RDMA SEND if the RQWRB resides in PM. However, both the dependent updates can be packaged in a single SEND message. (As noted earlier in the singleton update case, RDMA FLUSH will be required for the iWARP transport protocol.) If the responder’s RQWRB resides in DRAM, the typical message passing idiom is needed to ensure remote persistence of the two updates.

3.4 Discussion

Our analysis above is intended to provide guidance to application developers for correct remote persistence. It is clear that

the method for remote persistence using RDMA operations varies significantly between the twelve different configurations detailed above. We make a few interesting observations based on our analysis of remote persistence methods for these various remote server configurations.

First, the DDIO feature was originally introduced to improve performance of applications that used the RDMA networking fabric. We however find that the DDIO optimization gets in the way of performing remote persistence using just the one-sided operations – RDMA WRITE and RDMA FLUSH – in remote servers configured with DMP, which will likely be a substantial portion, perhaps a majority, of systems supporting PM in the near future. Second, for MHP and WSP, placing the RQWRB in PM, enables treatment of RDMA SEND messages as one-sided operations, which will likely lead to lower latency communication between the requester and responder using these operations. Third, with WSP, the new RDMA FLUSH operation that is being discussed in the IBTA standards community [10, 28] becomes unnecessary for InfiniBand and RoCE network fabrics, although it is still required for iWARP. Fourth, for compound RDMA updates, we find that the new WRITE_{atomic} operation applies to a narrow set of configurations in the whole taxonomy. Lastly, while we described different methods of remote persistence for different system configurations, methods such as RDMA SEND based message passing for remote persistence are *universal* in that they can be used in *all* system configurations. However, as we will see in §4, they come with a performance penalty compared to remote persistence with just one-sided RDMA operations.

We note that RDMA FLUSH and non-posted RDMA WRITE are not supported in today’s RDMA protocol. However, RDMA FLUSH can be correctly emulated using RDMA READ [13]. This is because RDMA READ flushes the responder’s RNIC’s buffers the IIO per the RDMA ordering rules and then triggers a PCIe READ at the responder’s RNIC, which in turn flushes the IIO buffers to memory [36]. Non-posted RDMA WRITE cannot be correctly emulated by any existing RDMA operations at present. Fenced RDMA WRITE can be used for similar ordering behavior, however, it adds an extra round-trip between the requester and responder before the fenced WRITE can be sent by the requester.

Torn writes are always a data consistency concern for persistence. The concern is no different in RDMA based remote persistence. The application must ensure robustness against torn writes via algorithmic techniques such as *checksums* and strictly ordered writes [10, 22, 40], all of which are very well understood in the literature.

4 Evaluation

Our taxonomy clearly demonstrates that the method to correctly ensure persistence of RDMA updates varies significantly between the various system configurations. We however would like to understand the performance trade offs

between these methods. There are several key questions we want to answer using our evaluation. (i) Do the different methods for remote persistence perform differently? (ii) Is there a significant enough performance gap between the universal message passing based remote persistence and one-sided remote persistence? (iii) How much performance impact do the various persistence domains have on remote persistence? (iv) Does DDIO affect remote persistence performance? (v) Does placement of RQWRB in DRAM or PM matter to performance?

To answer these questions, we use a workload that is ubiquitous to distributed systems that perform replication for high availability – *log replication*. Log replication is arguably the dominant method used to perform replication of updates to remote nodes in a distributed system.

4.1 Log Replication with REMOTELOG

Our benchmark, called REMOTELOG, sets up a contiguous log at the server end that is accessible to the client over a RDMA connection. REMOTELOG’s client repeatedly appends 10 million log records to the log. Each append is made the RDMA based remote update and persistence methods discussed in this paper. We perform the experiment for all the 72 configurations from Table 2 and Table 3. We report average log append latency at the end of each experiment.

REMOTELOG’s append operation provides a test bed for both singleton RDMA updates and compound RDMA updates. Log appends happen at the tail of the log. This can be done using singleton RDMA updates by encoding the log record with a checksum. This checksum is used to detect the tail of the log at the server – the server detects the log tail when its checksum fails. Checksums are also an effective way to detect data corruption. Thus checksummed log records can enable a way to do log appends using singleton RDMA updates.

Another dominant means of maintaining a log is by explicitly managing the server’s log tail pointer from the client’s end. The client needs two RDMA updates to perform an append – first to write (and persist) a new log record at the log tail, and second to write (and persist) the tail pointer reflecting the new tail of the log. This compound update provides the compound RDMA update use case in our experiments.

In both cases, the server asynchronously garbage collects log records that have been applied at the server end.

4.2 Experimental Setup

Our experiments were conducted in a single client and single server setting, where both systems hosted a dual-socket Intel® Xeon® E5-2600 processors with 8 hyperthreaded cores per socket with a total of 48 GB of memory. The systems run the Fedora 25 distribution of Linux. We emulate PM with DRAM. Each system contains a Mellanox ConnectX-4 100 Gb/s InfiniBand RNIC that is used to communicate over the RDMA network fabric. The client and server communicate with each other via a Mellanox SB7700 36 port 100

Gb/s InfiniBand switch. Our underlying RDMA framework used busy-waiting for completions rather than sleeping while awaiting a completion event.

In our experiments, we emulated a RDMA FLUSH with a RDMA READ. We cannot correctly emulate a non-posted RDMA WRITE. However, for performance estimation of a RDMA FLUSH followed by a non-posted RDMA WRITE, we can use a RDMA READ followed by pipelined RDMA WRITE and a second RDMA READ. The RDMA WRITE can be ordered before the first RDMA READ at the server. However, the second RDMA READ will not be reordered with the first RDMA READ, and we believe will give a reasonable estimate for the overhead of a non-posted RDMA WRITE, although it does not enforce the correct ordering semantics.

4.3 Singleton RDMA Updates

Figure 2 (a), (b), and (c) show the latencies of REMOTELOG append operations for the various server configurations. The various methods for remote persistence indeed have a significant impact on latency of appends. The more general trend is toward a sizable difference between one-sided and two-sided (classic message passing) operations, where the former outperforms the latter by up to 50%.

The persistence domains also have a significant impact on append latencies: When transitioning from DMP to MHP, the observed difference largely reflects the difference in method of remote persistence. For instance, for the DDIO_DRAM_RQWRB_WRITE bars, MHP performs significantly better than DMP since the former uses one-sided RDMA operations compared to the two-sided operations used by the latter, which lead to a ping-pong of messages between the client (requester) and server (responder) – a full round trip with additional CPU processing on the server’s end to flush cache target lines. For both, MHP and WSP, the foundational difference between performance boils down to whether the RDMA update is performed (and persisted) using one-sided operations or message passing based ping-pong between the client and the server, with the latter incurring the round-trip overhead for messages sent back and forth between the client and server. We also note that RDMA FLUSH has a significant impact on latency. In WSP, omission of RDMA FLUSH in a one-sided RDMA update drops its latency to 1.6 microseconds (a 25% reduction in latency from the one-sided RDMA updates in MHP). Overall, as expected, WSP enables the best latency for remote persistence using RDMA operations.

DDIO appears to selectively have a negative impact on performance of some configurations in DMP, particularly for RDMA WRITE and RDMA WRITEIMM, when the RQWRB is placed in PM. But this effect is indirect in that it forces a two-sided operation to ensure that the remotely updated cache lines are flushed to the DMP domain. Placement of RQWRB has a significant performance indirectly as well in that if the RQWRB is placed in PM, RDMA SEND

can be treated by the client as a purely one-sided operation, and hence gain the performance advantage of one-sided operations. However, care must be taken by programmers on balancing consumption of receive queue buffers at the server end with the rate of RDMA SEND and RDMA WRITEIMM operations coming from the client. Each such operation consumes a receive queue buffer on the server’s end, and the server must quickly recycle these buffers in order to continue receiving messages from the client. If the server is too slow, resource availability timeouts may be triggered on the client’s end leading to performance jitter.

4.4 Compound RDMA Updates

Compound update latency results appear in Figure 2 (d), (e), and (f). As in the case of singleton updates, the server configuration and resulting method of remote persistence has a significant impact on latencies of remote persistence. The universal message passing based approach, which had a negative performance impact for singleton updates, appears to have a significant advantage in servers supporting the DMP domain. The advantage is that the two updates – the log tail record and the tail pointer – can be packaged in a single message by the client, which keeps the operation to a single round trip. In contrast, use of RDMA WRITE and WRITEIMM with message passing leads to two round trips leading to more than 2X latency in DMP when DDIO is turned on. However, MHP unlocks the capability of doing one-sided compound RDMA updates lead to significant latency improvements in RDMA WRITE and WRITEIMM based methods, which end up performing with a latency up to 20% better than the latency of message passing. This gain is more pronounced to 30% for WSP.

Similar to singleton updates, DDIO appears to have an indirect negative performance impact on DMP configurations for RDMA updates done using RDMA WRITE and RDMA WRITEIMM, in that it forces additional message passing (and cache line flushing at the server) overheads for remote persistence. The non-posted RDMA WRITE based method is enabled when DDIO is turned off, and appears to deliver a big performance improvement. In general, turning DDIO off enables compound updates using RDMA WRITE and WRITEIMM to be done using just one-sided RDMA operations, where the big performance manifests. Notice however, that the latency of RDMA WRITEIMM does not drop as much. This is because non-posted writes enable pipelining of updates and RDMA FLUSH. However since there is no non-posted version of RDMA WRITEIMM available, the RDMA WRITEIMM based method, incurs overhead of completion of the first RDMA FLUSH before issuing the second RDMA WRITEIMM. As expected, DDIO has no effect on MHP and WSP configurations.

Placement of RQWRB in PM enables a big optimization in the RDMA SEND based method in that, provided the persistence domain is either MHP or WSP, or DDIO is turned

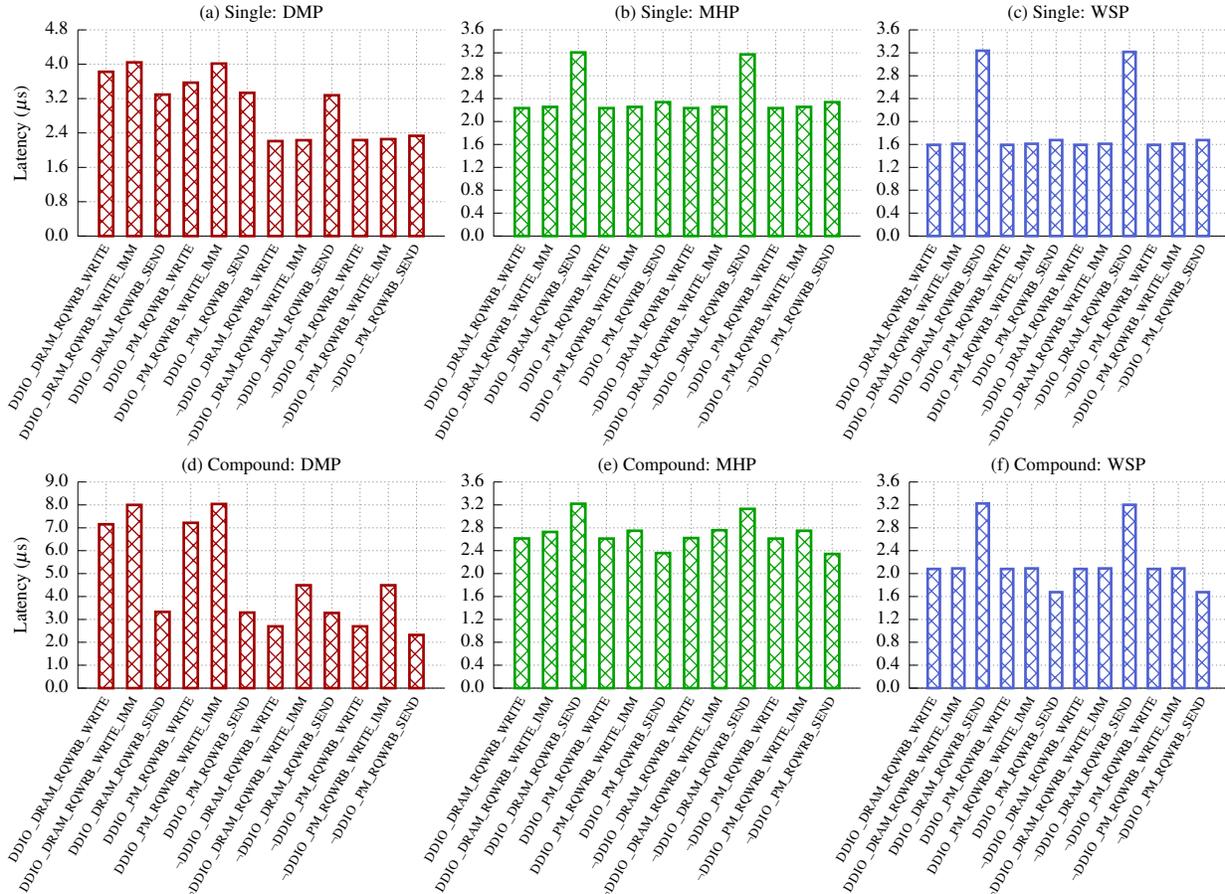


Figure 2: Latencies of remote persistence of REMOTELOG appends for singleton and compound RDMA update based implementations. In all cases, the client appends a 64-byte wide log record.

off, RDMA SEND can be treated by the client as a one-sided RDMA operation. As in the case of singleton updates, absence of RDMA FLUSHes in servers configured to support WSP boosts latency of remote persistence by close to 20%.

5 Conclusions

We presented the first comprehensive taxonomy of methods for persistence of RDMA updates to remote persistent memory. Our taxonomy spans server configurations along three different axes: (i) the persistent domain of the system, (ii) use of DDIO, and (iii) placement of RQWRBs in DRAM or PM. We showed how these configurations affect the methods to correctly and efficiently enforce persistence of RDMA updates. We also included some of the recent advances in the IBTA standards community [10, 28] in our analysis. Our detailed analysis covered persistence of singleton RDMA updates as well as compound RDMA updates that need to be persisted in the order they were issued from the requester. We find that the methods to correctly, and efficiently, persist RDMA updates vary significantly based on the underlying system’s configuration parameters enumerated above. Programmers must be extremely careful in applying these methods – application of the wrong method can lead to significant performance overheads, and even critical data inconsistencies

in the face of system failures.

Our evaluation demonstrated several interesting performance trade offs between available methods for persistence of RDMA updates. In particular, we find that remote persistence done using one-sided RDMA operations (RDMA WRITE, RDMA WRITE_IMM, RDMA FLUSH, and even RDMA SEND in cases where the RQWRB resides in PM) is generally more efficient than remote persistence enforced using RDMA SEND based message passing. In the end, we believe the workloads requirements may determine the best choices. A client may need to perform a complex set of non-contiguous updates at the server, which would be better served by a single RDMA SEND based remote procedure call (RPC) [19].

The newly proposed non-posted RDMA WRITE based method is also quite effective in delivering better performance. However, this RDMA extension seems useful only in a small part of the space of system configurations we explored in the paper. Perhaps, that small part itself could represent the dominant system configurations used in the industry. It remains to be seen what configurations will be used widely in the future.

Given the wide range of choices of remote persistence, it may be reasonable to build a single RDMA library that

transparently applies the correct method of remote persistence for a given system and application. There may be interesting subtleties that may lead to sub-optimal performance, and even correctness issues in the face of failure. However, we leave the exploration for future work. Another interesting aspect that remains to be explored is implications of these choices for remote persistence on memory persistency models [20, 29].

References

- [1] 3D XPoint Technology Revolutionizes Storage Memory., 2015.
- [2] AgigaRAM NVDIMMs. <http://agigatech.com/>.
- [3] AGIGA TECH. NVDIMM Messaging and FAQ. *SNIA Solid State Storage Initiative*. <https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf>, 2014.
- [4] Cache Stashing. <https://developer.arm.com/docs/100453/latest/part-a-functional-description/cache/cache-stashing>.
- [5] Arm® DynamIQ™ Shared Unit. Revision r3p0. Technical Reference Manual. https://static.docs.arm.com/100453/0300/dsu_trm_100453_0300_01_en.pdf.
- [6] Armv8-A architecture evolution. <https://community.arm.com/processors/blog/posts/armv8-a-architecture-evolution>.
- [7] ASSOCIATION, I. T. Supplement to InfiniBand™ Architecture Specification, Volume 1, Release 1.2.1, Annex A17: RoCEv2, 2014.
- [8] ASSOCIATION, I. T. InfiniBand™ Architecture Specification, Volume 1, Release 1.3, 2015.
- [9] BESTLER, C., AND STEWART, R. Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation. IETF RFC 5043. <https://tools.ietf.org/html/rfc5043>, 2007.
- [10] BURSTEIN, I. RDMA Memory Placement Extensions for PMEM. In *2018 Flash Memory Summit* (2018).
- [11] CULLEY, P., ELZUR, U., RECIO, R., BAILEY, S., AND CARRIER, J. Marker PDU Aligned Framing for TCP Specification. IETF RFC 5044. <https://tools.ietf.org/html/rfc5044>, 2007.
- [12] Intel Data Direct I/O Technology. <https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html>.
- [13] DOUGLAS, C. RDMA with PM: Software Mechanisms for Enabling Persistent Memory Replication. In *2015 Storage Developer Conference* (2015).
- [14] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), pp. 401–414.
- [15] InfiniBand™ Trade Association. <https://www.infinibandta.org/>.
- [16] Intel® 64 and IA-32 Architectures Software Developer’s Manual. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>, 2015.
- [17] IZRAELEVITZ, J., KELLY, T., AND KOLLI, A. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), pp. 427–442.
- [18] JEDEC STANDARDS COMMITTEE. DDR4 NVDIMM-N Design Specification. <https://www.jedec.org/standards-documents/docs/jesd248>, 2018.
- [19] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference* (2016), pp. 437–450.
- [20] KOLLI, A., GOGTE, V., SAIDI, A. G., DIESTELHORST, S., CHEN, P. M., NARAYANASAMY, S., AND WENISCH, T. F. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), pp. 481–493.
- [21] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference* (2017), pp. 773–785.
- [22] MARATHE, V. J., MISHRA, A., TRIVEDI, A., HUANG, Y., ZAGHLOUL, F., KASHYAP, S., SELTZER, M., HARRIS, T., BYAN, S., BRIDGE, B., AND DICE, D. Persistent Memory Transactions. <https://arxiv.org/abs/1804.00701>, 2018.
- [23] MELLANOX TECHNOLOGIES. RDMA Aware Networks Programming User Manual, Rev 1.7.

- [24] Micron Technology. <http://www.micron.com/products/dram-modules/nvdimn>.
- [25] NARAYANAN, D., AND HODSON, O. Whole System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012).
- [26] NAWAB, F., CHAKRABARTI, D. R., KELLY, T., AND III, C. B. M. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015* (2015), pp. 689–694.
- [27] Intel Optane DC Persistent Memory readies for widespread deployment. <https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment/>, 2018.
- [28] PAUL GRUN, STEPHEN BATES, R. D. Persistent Memory over Fabrics (PMoF). In *Persistent Memory Summit* (2018).
- [29] PELLELY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014* (2014), pp. 265–276.
- [30] PINKERTON, J., AND DELEGANES, E. Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMA) Security. IETF RFC 5042. <https://tools.ietf.org/html/rfc5042>, 2007.
- [31] RECIO, R., METZLER, B., CULLEY, P., HILLAND, J., AND GARCIA, D. A Remote Direct Memory Access Protocol Specification. IETF RFC 5040. <https://tools.ietf.org/html/rfc5040>, 2007.
- [32] RUDOFF, A. Deprecating the PCOMMIT Instruction. <https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction>, 2016.
- [33] RUDOFF, A. Programming Persistent Memory. *login*: 42, 2 (2017).
- [34] SHAH, H., MARTI, F., NOUREDDINE, W., EIRIKSSON, A., AND SHARP, R. Remote Direct Memory Access (RDMA) Protocol Extensions. IETF RFC 7306. <https://tools.ietf.org/html/rfc7306>, 2014.
- [35] SHAH, H., PINKERTON, J., RECIO, R., AND CULLEY, P. Direct Data Placement over Reliable Transports. IETF RFC 5041. <https://tools.ietf.org/html/rfc5041>, 2007.
- [36] THE PCI-SIG. PCI Express Base Specification Revision 3.0, 2010.
- [37] THE SNIA NVM PROGRAMMING TECHNICAL WORKING GROUP. NVM PM Remote Access for High Availability, 2013.
- [38] THE SNIA NVM PROGRAMMING TECHNICAL WORKING GROUP. NVM Programming Model (Version 1.0.0 Revision 10), Working Draft. http://snia.org/sites/default/files/NVMProgrammingModel_v1r10DRAFT.pdf, 2013.
- [39] Viking Technology. <http://www.vikingtechnology.com/nvdimn-technology>.
- [40] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 91–104.
- [41] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 87–104.
- [42] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. 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* (2015), pp. 3–18.