Fast Networks and Slow Memories: A Mechanism for Mitigating Bandwidth Mismatches

Timo Schneider*†, James Dinan*, Mario Flajslik*, Keith D. Underwood*, and Torsten Hoefler†
Intel Corporation*, ETH Zurich†

Abstract—The advent of non-volatile memory (NVM) technologies has added an interesting nuance to the node level memory hierarchy. With modern 100 Gb/s networks, the NVM tier of storage can often be slower than the high performance network in the system; thus, a new challenge arises in the datacenter. Whereas prior efforts have studied the impacts of multiple sources targeting one node (i.e., incast) and have studied multiple flows causing congestion in inter-switch links, it is now possible for a single flow from a single source to overwhelm the bandwidth of a key portion of the memory hierarchy. This can subsequently spread to the switches and lead to congestion trees in a flow-controlled network or excessive packet drops without flow control. In this work we describe protocols which avoid overwhelming the receiver in the case of a source/sink rate mismatch. We design our protocols on top of Portals 4, which enables us to make use of network offload. Our protocol yields up to 4x higher throughput in a 5k node Dragonfly topology for a permutation traffic pattern in which only 1% of all nodes have a memory write-bandwidth limitation of 1/8th of the network bandwidth.

Non-Volatile Memory (NVM) provides a new tier in the traditional system memory and storage hierarchies. NVM provides higher density, lower cost, and lower power than conventional DRAM and it is dramatically higher in performance than traditional storage devices. For example, recently released Intel® Optane™ Solid State Drives (SSDs) that utilize 3D XPoint™ memory technology provide 5 GB/s read and 3 GB/s write bandwidth [1]. This makes NVM beneficial for a wide range of applications that process large data sets [2], [3], [4], as well as for more general usage models such as checkpointing [5] and IO staging [6].

Given the breadth of usage models, NVM is being deployed in a variety of environments ranging from conventional datacenters to high performance computing (HPC) systems. In many deployments, high performance datacenter networks and HPC fabrics have now reached 100 Gb/s speeds (12.5 GB/s). This includes 100 Gigabit Ethernet⋆, EDR InfiniBand⋆, and Intel Omni-Path™ Fabric [7]. Most of these technologies have options to enable remote direct memory access (RDMA) to user memory. At the same time, a popular usage model is to transparently map NVM into an application's address space [8], [9]. This leverages existing mechanisms within operating systems and the cache hierarchy to hide latencies.

The combination of the transparent mapping of NVM and the potential a bandwidth mismatch between the network and memory in a receiving node poses a challenge to communication middleware: two processes exchanging large messages cannot know the effective end-to-end bandwidth a priori. The sending process is likely to inject data at the network line rate: freshly generated data would be sent from an address range cached in DRAM. However, the receiver may sink the data into a buffer that is not cached in DRAM. Thus, the receiver cannot sink the data at the same rate. We investigate this problem in the context of HPC systems and middleware; however, the concepts are more broadly applicable.

Instead of dropping packets in such situations, which requires expensive retransmission protocols, HPC networks often utilize credits to pace senders and avoid packet loss. At each network hop, packets are only forwarded if there is sufficient buffer space at the receiver to store them. Thus, once the buffers residing in switches along the path of the transferred data are full, the sender will be back-pressured, since it is out of credits. While this mechanism prevents packet loss, it potentially impacts other messages utilizing the same network, as buffer space is occupied by the DRAM-to-NVM transfer and is not available for other transfers that might be able to progress at a much faster rate.

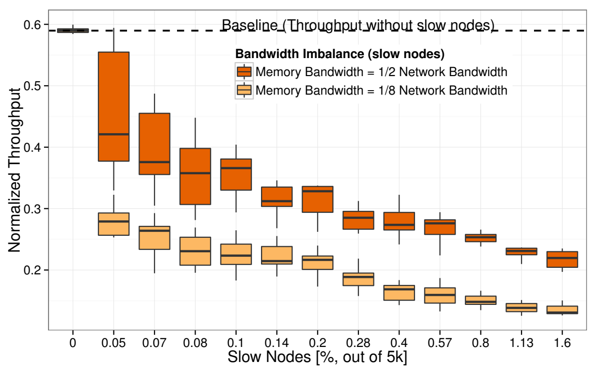

Fig. 1. Impact of slow memory nodes on average network throughput.

The magnitude of this problem — the overall system slowdown caused by slow nodes — depends on the network topology and routing mechanisms. Unfortunately, networks that are regarded as highly efficient in terms of cost per bandwidth, such as Dragonfly networks [10], are most susceptible to this problem. This is highlighted in Figure 1, which shows normalized average throughput in a 5,256 node balanced Dragonfly network with adaptive non-minimal routing for a permutation traffic pattern [11]. In the experiment, we vary the percentage of nodes that are bandwidth mismatched. With no slow nodes, a Dragonfly sustains nearly 60% of the injection rate. However, with as little as 0.1% of transfers into NVM (write bandwidth 2x and 8x less than network bandwidth), there is a dramatic decrease in average throughput. Variability shown in Figure 1 is generated by different random traffic permutations. With only one or two slow nodes in the system (i.e. 0.02% or 0.04% out of 5k), their impact depends on the random traffic permutation. With more slow nodes (e.g., 0.1%), overall throughput is repeatably poor for all permutations.

The simplest approach to avoiding such congestion is for the receiver to notify the sender of the maximum bandwidth that it can sink for a given transfer. This is only applicable for transfers that are larger than the maximum transmission unit (MTU) of the network, since it is not useful to throttle the transmission rate of a single packet. The sender then self-throttles according to this information. We consider this approach impractical; it requires either that a rate negotiation step be performed prior to each transfer, or that the sender maintains a large amount of knowledge about the state of every receiver. In addition, a rate negotiation step can negatively impact performance for transfers not involving NVM and may also impact the ability to utilize existing offloaded transfer methods. Storing bandwidth information at the sender is also infeasible, because this information is changing rapidly because of caching and buffering. Mis-predicting even a small number of times can lead to large performance penalties, as illustrated in Figure 1.

Thus we focus our investigation on the communication middleware. First, we explore if known congestion control mechanisms, such as Forward and Backward Explicit Congestion Notification (FECN/BECN) can be used to solve this problem. Our results indicate that this congestion management strategy does not improve performance in most cases. The FECN/BECN mechanism is unable to distinguish between individual transfers. As a result, it often slows down transfers that do not actively contribute to congestion, but happen to share resources with transfers that do.

To discern individual transfers we move further up the protocol stack and focus on the middleware. While we evaluate our proposed enhancements in the context of the *de facto* standard for HPC — the Message Passing Interface [12] (MPI), we expect the concepts to be applicable to other middleware layers. We present a novel, receiver-driven protocol for the exchange of large messages that can leverage network offloading capabilities and automatically paces data transfers at the rate of the lowest bandwidth step in the flow. Instead of transferring the bulk of the message in a single transfer, the receiver uses Remote Direct Memory Access (RDMA) read operations to transfer the data in smaller chunks, whose size is chosen to avoid overwhelming network buffers, even in the presence of a bandwidth mismatch. We implement our solution using the Portals 4 [13] interface, which defines a networking layer with a rich set of primitives capable of hardware offload [14]. We take advantage of these primitives to maximize offload of our data transfer protocol and demonstrate that chunking incurs a minimal performance penalty for transfers that do not suffer from bandwidth mismatch.

In the course of this work, we identified a critical flaw in a previously published [15] MPI large message protocol implemented on top of Portals 4. We describe this error and further prove that completely offloading an MPI rendezvous protocol is impossible with the current Portals API. This impossibility result in turn provides hints towards possible changes in the Portals 4 specification that can improve its ability to better support this important communication protocol.

1 BACKGROUND

While our approach is broadly applicable, in this work we describe its implementation and evaluate it in the context of MPI middleware implemented using the Portals 4 networking interface. In this section, we provide background information on the relevant MPI and Portals semantics.

1.1 Message Passing Interface

The Message Passing Interface (MPI) [12] defines an application programming interface (API) for exchanging data between parallel processes, referred to as MPI ranks. MPI supports a number of communication models; however, its point-to-point messaging is its most popular feature. This interface is comprised of blocking and nonblocking send and receiver routines. When implementing low level protocols for these functions, it is crucial to know their exact semantics (including unusual use cases) as defined by the MPI standard.

The send and receive functions take a tag, an MPI communicator handle, and arguments describing the location, size, and layout of the buffer from or to which data is transferred. The tag argument is a user-selected integer that the application can use to identify the message. A send operation must specify a tag; however, receive operations may ignore the tag by supplying `MPI_ANY_TAG`. The communicator argument defines the communication environment that is used; in particular, messages sent on a given communicator can only be received by receive operations performed on the same communicator. A peer rank argument is also provided indicating the process to which data is sent, or from which data is received. A send operation must also specify a valid peer rank; however a receive operation may ignore the peer rank by supplying `MPI_ANY_SOURCE`. Nonblocking versions of these routines also take a request handle, which can be used to check for completion of the operation. Finally, receive operations return the size of the message, the tag, and the source rank through a status object that is supplied directly to blocking receive operations or retrieved after completion for nonblocking receive operations.

MPI messages are *matched* by the receiver. A message matches a given receive operation if the following three conditions are met:

1) The operations occurred on the same communicator
2) The sender rank is equal to the rank specified at the receiver or the receiver specified `MPI_ANY_SOURCE`
3) The sender's tag is equal to the tag specified at the receiver or the receiver specified `MPI_ANY_TAG`

MPI message matching is defined to be non-overtaking. Given two processes, A and B, a receive operation posted by B that can match multiple pending send operations performed by A must match A's oldest matching operation. Similarly, a send operation performed by process A that can match multiple pending receive operations performed by B must match B's oldest matching operation. Thus, even when `MPI_ANY_TAG` is used, MPI message matching is deterministic. However, nondeterministic matching can occur when `MPI_ANY_SOURCE` is used and messages from multiple senders match the same receive operation.

Note that the message size is not part of the matching criteria. If a message matches a receive and the receive buffer size is smaller than the message, it is treated as an error. If the message is smaller than the specified receive buffer, the receive buffer is filled partially. The programmer can use the status handle to check the size of the received message. As a consequence of these semantics, the receiver does not know the message size at the time the receive buffer is provided. Similarly, the receiver (i.e. MPI library) also does not know the tag argument if `MPI_ANY_TAG` was used, nor do they know the source rank of the message if `MPI_ANY_SOURCE` was used. This information gap has a significant impact on protocol design for MPI messaging.

The fact that messages from the same source to the same destination are not allowed to overtake each other might suggest that sequence numbers can be used to match sends with receives; however, as shown in Section 3.1, since MPI supports non-blocking messages the sender and receiver can perform their respective operations in different orders.

1.2 Portals 4

Portals offers two types of interfaces, matching and non-matching, that support different receiver-side message processing models. Both interfaces support RDMA put, get, and atomic operations and allow the user to specify the target offset within the destination memory region. Memory regions are exposed by appending list entries to portal table entries. In the non-matching interface, the first list entry on the given portal table entry is always selected, whereas on the matching interface, a list walker traverses the list and selects the first entry that matches the incoming message.

In the case of the matching interface, the initiator specifies a 64-bit *match bits* value that contains the user-supplied MPI tag and an integer context ID that identifies the communicator. Receive operations are implemented by appending match list entries to the priority list of a given portal table entry. Match list entries contain a peer ID (or `PTL_RANK_ANY`) and match bits similar to those used by the sender. In addition, the match list entry also specifies ignore bits, which can be used to ignore the tag portion of the match bits. Before processing an incoming put, get, or atomic operation, Portals must search the priority list to locate a matching list entry. If a match is not found in the priority list, an overflow list that is used for handling unexpected messages is searched. These lists are searched in list order using the matching rule, `((msg_bits^match_bits) & ~ign_bits) == 0`.

A match list entry can have a counter associated, which can be configured to count different types of events, such as successful transfers or transferred bytes. At the initiator, buffers used for put or get must be registered before their usage. A registered buffer is referenced by a memory descriptor. Similarly to a match list entry, a memory descriptor can have a counter associated with it that counts the number of communication operations or bytes read/written. Counting events are meant to be lightweight events; thus, they only carry a small amount of information. Alternatively, full events can be used, which carry additional information such as the size of the matched message, the match bits used by the peer, the peer rank, and additional details. Full events are stored in an event queue to be delivered to software and cannot be used to trigger other operations; however, one operation can be configured to deliver both a full event and a counting event.

One approach to offloading communications in Portals leverages the counters attached to memory descriptors and match list entries. Almost all of the Portals functions, such as put or get, have a triggered variant that takes two additional arguments: a counter handle and a threshold value. The triggered operation is executed once the specified counter reaches (or is greater than) the threshold. A high-performance implementation of the Portals 4 API can execute triggered operations without involvement of the host CPU, and thereby guarantee asynchronous progress. One important limitation of Portals 4 is that the arguments of a triggered operation cannot be changed after the operation has been posted.

A Portals match list entry can be configured to either match and truncate a message that is larger than the targeted buffer, or to mismatch and continue searching the match list. If a message is truncated and generates a full event, this full event will carry information about the total message length, as well as the amount of data actually received.

2 Related Work

Extensive research and implementation optimization has been done for large MPI messages. In this work we will only discuss RDMA based protocols. Rashti and Afsahi [16] propose a RDMA based protocol that allows either the sender or the receiver to initiate the transfer, depending on who arrives first. However, they do not have mechanisms that would allow bandwidth metering.

Congestion control in HPC systems is widely available. In InfiniBand (IB) [17], a switch detecting congestion sets a Forward Explicit Congestion Notification (FECN) bit [18]. Upon receiving this bit, the destination sends a backward ECN (BECN) bit to the source, which will respond by temporarily throttling injection. The problem with this approach is that it is reactive — it only springs into action once congestion occurred. In addition, it takes up to two round-trips to have any effect and it is probabilistic. Once congestion is detected, the switch cannot know which message transfer should be throttled, thus it randomly (based on a configurable probability) selects packets to mark with the FECN bit. Thus, in the presence of bandwidth imbalance it is likely that flows which did not cause the congestion (but happen to share a switch buffer with the flow at fault) are slowed down. The shortcomings of IB congestion control are demonstrated by Luo et al. [19], which shows that simply varying the number of concurrent flows can lead to a 30% throughput difference for large messages.

Barret et al. [15] proposed a fully offloaded rendezvous protocol implemented on top of Portals 4 triggered operations. They use a single get operation to retrieve the data, so it is not adaptive to bandwidth imbalance between the sender and receiver. In addition, it does not conform to MPI semantics. The authors partially acknowledge that — in the presence of `MPI_ANY_SOURCE` they fall back to a different protocol that is not fully offloaded. Their protocol attempts to deal with `MPI_ANY_TAG` by using sequence numbers (between each pair of ranks). The sequence number is subsequently used as match bits by the sender and the receiver. However, this implies that the i-th send from rank a to rank b has to match the i-th receive posted by rank b expecting a message from rank a. As discussed in Section 1.1, the MPI standard does not require processes to perform matching send and receive operations in the same order. Thus, the proposed protocol does not achieve the intended purpose across legal use models, and designing an efficient, fully offloaded MPI rendezvous protocol is still an open problem. Schneider et al. [20] proposed offloaded protocols as well, however, the proposed protocols are intended to implement collective operations, where `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be avoided altogether and all send and receive operations are known a priori.

MPI implementations such as Open MPI [21] and MPICH [22] support Portals 4 as a back-end and thus implement protocols for point-to-point messaging. The protocol [23] implemented by MPICH is similar to ours — it is get based for large messages and initiated by the sender. However, it does not split large messages into smaller transactions in order to account for bandwidth-mismatch between sender and receiver. The protocol [24] implemented by Open MPI is the same if one chooses to use the eager/rendezvous version. Open MPI can be configured to use an eager protocol for large messages as well [25].

3 Receiver-Driven Rendezvous Protocol

We introduce a receiver-driven rendezvous protocol (RDRP) for large message transfer that transparently throttles bandwidth mismatched transfers. The protocol is receiver-driven; that is, it utilizes RDMA read operations (`PtlGet` in Portals 4) to transfer most of the data. However, in contrast to previously described protocols, we do not use a single RDMA read to transfer the data. Instead, we break up the transfer into chunks and transfer each chunk individually. The transfer of each chunk is scheduled by the receiver to avoid overwhelming their available memory bandwidth. When using a traditional network interface, this approach may degrade the potential for communication/computation overlap or require additional service threads, as software intervention would be required to initiate chunk transfers as bandwidth becomes available. However, in Portals 4, we use triggered operations to offload starting subsequent chunks as the transfer of previous chunks is completed.

3.1 Impossibility of Full Rendezvous Offload in Portals

Ideally, the message transfer protocol should be fully offloaded; i.e., all calls into the Portals interface occur in the send and receive functions and asynchronous progress is thereafter provided by the Portals 4 implementation. However, because of limitations in the Portals 4 interface and the semantics of MPI messaging (even for the case where we exclude the use of `MPI_ANY_SOURCE`), full offload for MPI rendezvous protocols implemented on top of Portals is not possible. Instead, a software bootstrapping step is required, as described in Section 3.2.

We prove the impossibility of full offload in two steps. In the first step, we establish that a fully offloaded protocol would require the sender and receiver to choose the match bits used in each non-blocking send/receive without exchanging any additional information. In the second step, we show that no scheme exists which assigns match bits to send/receives based on local information only and adheres to MPI matching semantics.

Theorem 1. In a fully offloaded receiver-driven rendezvous protocol over Portals 4, the sender and receiver need to choose match bits based on local information only.

A receiver-driven rendezvous protocol satisfying MPI semantics must contain (at least) two messages. First, the sender must indicate that the data is available, and also specify the total size of the message, as well as the match bits used. The receiver cannot have this information a priori since MPI allows the receiver to specify a larger buffer as well as using `MPI_ANY_TAG`. We refer to this initial message as ready-to-send (RTS) in the following. After the receiver has performed the matching and has knowledge that the sender's data is available, it can issue a second message to perform an RDMA read operation to receive the data. Other protocols — where the sender transfers the whole message to the receiver, or protocols that poll for data at the sender — are not classified as rendezvous protocols.

When considering the expected message case, we can identify several crucial points in time: at t_1 the sender posts the send, at t_2 the receiver posts the matching receive. We assume this receive is non-blocking. At t_3 the receive finishes. The RTS arrives at the receiver at t_4, the get starts at t_5. We denote the point at which the get operation is offloaded at t_x. We show a graphical representation of this in Figure 2. We use the notation $a \rightarrow b$ to express a has to happen before b.

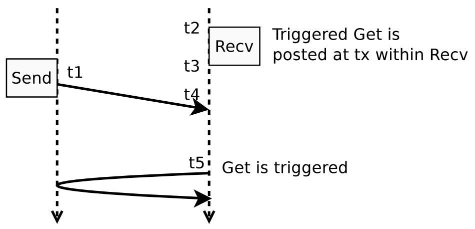

Fig. 2. Sequence of operations performed by sender and receiver during a rendezvous protocol message transfer.

We require $t_3 \rightarrow t_4$ because we restrict our discussion to expected messages and non-blocking receives. Thus the receive must be able to return independently of the sender. Also we require that $t_x \rightarrow t_3$, otherwise we do not have full offload and asynchronous progress; all calls into the Portals API (except checking for completion) must be made before t_3. If we obtain the match bits to use for the get from the RTS message $t_4 \rightarrow t_x$ which leads to a cycle. Since we cannot modify other edges constituting this cycle, we need to remove $t_4 \rightarrow t_x$, by choosing the match bits without involving the sender. ■

It was suggested previously [15] to utilize sequence numbers for this purpose. However:

Theorem 2. Any scheme in which the sender and receiver select match bits independently from each other, based on local information only, cannot implement MPI semantics.

The sender has no information about the ordering or tags of receives the receiver will post; thus, the sender's sequence of match list entries posted for RDMA read by the receiver is invariant of what the receiver does. Thus, for the purpose of analysis we can assume the sends are posted first without loss of generality. Similarly, the receiver also has no information about the order in which send operations are performed by the sender and in the case of `MPI_ANY_TAG`, also has no knowledge of the tag that will be used by the sender. This suggests that neither the order in which operations are performed, nor the tag can be used by sender and receiver to independently select match bits.

We construct an example with two sends, each using a different tag, $t1$ and $t2$, which will lead to two match list entries, each consisting of (potentially different) match bits and ignore bits. On the receiver side, there are now seven ways to post receives ($*$ represents `MPI_ANY_TAG`), as shown in Table 3.1. We indicate the matching behavior that is required by MPI next to each case, sx $\leftrightarrow$ ry means the x-th send matches the y-th receive. Note that there is one case where the second receive does not match anything, which can result in deadlock in the application.

For each of these receives, the match bits can be chosen according to the following criteria: tag $\in$ $\{t1, t2, *\}\times$ position $\in$ $\{1, 2\}\times$ tag-before $\in$ $\{none, t1, t2, *\}$, where tag-before denotes which tag was seen before the current recv. To reduce the size of the search space from the full 64-bit match bits available in Portals, we encode the matching criteria as a 9-bit match bit string using a bit position for each sub-state value, allowing ignore bits to mask any subspace. Since we now determined the match bits the

TABLE 1
Possible send and receive orderings for two messages transmitted with tags $t1$ and $t2$. Receive operations with tag $*$ can match any tag.

Sender	Receiver	Matching
	recv($t1$); recv($t2$)	s1↔r1; s2↔r2
	recv($t2$); recv($t1$)	s1↔r2; s2↔r1
	recv($*$); recv($*$)	s1↔r1; s2↔r2
send($t1$); send($t2$)	recv($*$); recv($t2$)	s1↔r1; s2↔r2
	recv($*$); recv($t1$)	s1↔r1; no match
	recv($t1$); recv($*$)	s1↔r1; s2↔r2
	recv($t2$); recv($*$)	s1↔r1; s2↔r2

receiver will use to fetch data during the RDMA read operation, we can search all possible combinations for the match and ignore bits in both sends for a combination that obeys MPI matching and non-overtaking semantics. A search of the resulting 36-bit search space yields no viable combination of match/ignore bits. ■

3.2 Chunked Receiver-Driven Rendezvous Protocol

We have established that a fully offloaded receiver driven protocol cannot be implemented over Portals 4 because current triggered operations semantics force sender and receiver to choose match bits independently. We now present the design of our protocol, which uses full events to retrieve match bits that are chosen by the sender prior to posting triggered get operations.

The timing diagram in Figure 3 shows the flow of messages in the unexpected and expected cases. The sender sends the first chunk of the message to the receiver, this chunk includes header data (match bits, message size, and sender rank). Upon receiving the first message and matching it against local receives, once a matching receive is found, the receiver posts a series of gets of a limited size, ensuring there are no more than c outstanding gets. Doing so ensures that the network is not flooded with gets that cannot complete due to insufficient write bandwidth at the receiver (the chunksize and c need to be chosen according to the buffer space available in switches, we show their impact on performance in Section 4).

3.3 Implementation

In the following we describe the implementation of non-blocking send, receive and test functions that realize the protocol described above. Blocking versions can easily be implemented using wrapper functions that first execute the non-blocking functions, then wait until they complete before returning.

Non-Blocking Send: A send operation will perform a `PtlPut` on the matching interface using a concatenation of tag and communicator id as match bits. A sender-selected matching nonce can also be incorporated into the match bits to distinguish messages with identical matching criteria, allowing them to be retrieved concurrently. These *nonce bits* must uniquely distinguish a pending send operation from other pending sends with identical matching criteria; thus, a small number of bits is sufficient. A special nonce value can be reserved to inform the receiver to fall back to an in-order retrieval protocol when all nonce bit values are already in use. If nonce bits numbers are used, the corresponding bits must be masked by ignore bits at the receiver.

It is important that the send message includes the total size of the message in its header (this is done automatically by Portals). The sender performs the initial Put with the full message size, but posts a truncating match list entry (ME) at the receiver side to truncate to receive only the first chunk; a high-quality Portals implementation will ensure no excess data is transmitted. Before the put is sent, MEs for the subsequent gets (using the same match bits) need to be set up. The sender sets the ignore bits to zero for all MEs posted by the sender. If the message size is below the eager/rendezvous threshold, the sender does not post any MEs, since there will be no subsequent gets. In this case, the state of the request associated with the send is set to done once the put has completed. If the message size is larger than the eager threshold, the sender appends the ME with a counter attached configured to count successful gets. The state of the send request is set to `send_wait`; in a subsequent wait call the sender compares the counter value to the expected value and sets the request to done when they match. This comparison and local update can also be performed using a triggered operation that targets local memory.

Non-Blocking Receive: The implementation of the receive is more complex; upon initializing MPI each rank appends a number of truncating MEs to its overflow list. These MEs will receive the first chunk of unexpected messages into bounce buffers. Furthermore, if we attempt to append an ME into the priority list, Portals will first search the unexpected headers associated with this overflow list for an ME that has already matched a message with compatible match bits. Thus, when we enter receive, we construct an appropriate ME (using the ignore bits to mask out the part of the match bits corresponding to nonce bits and the tag if `MPI_ANY_TAG` is used, and setting peer to `PTL_RANK_ANY` in case of `MPI_ANY_SOURCE`).

Next, the receiver attempts to append the ME to the priority list. If the append matches an unexpected message, the receiver captures a full event, indicating the match bits, message size, and overflow list entry that was used to buffer the eager data chunk. Thus we copy the first chunk of the message from the bounce buffer into the receive buffer and post the get sequence. The state of the receive request is set to `wait_gets` until all chunk gets have completed. If appending the ME does not match an unexpected message, the receiver knows that it is in the expected message case. At some point an incoming message will match the posted ME, which will also generate a full event, containing the match bits, total message size, and local location of the data. We cannot wait for this event in the receive itself, thus we set the request state to `wait_match` and poll the event queue for the match event in the MPI progress engine (e.g., in a call to the `MPI_Test` function).

Progress Engine: In the MPI progress engine we first check the state of the tested request: if it is `send_wait`, we check if the successful number of gets has reached the expected value (based on the message size and chunk size). If it has, we set the request state to `done`. If the request is in state `wait_match`, we check if there is a put event in the event queue, if an event is present, we copy the first chunk of the message from the bounce buffer into the receive buffer, retrieve the match bits and message size from the event. If the message size is below the rendezvous threshold we set the request state to `done`, otherwise we post a sequence of gets and set the request state to `wait_gets`. A request in the `wait_gets` state is progressed by checking if all gets have completed (the memory descriptor used as a target for the get has a counter attached to it), if this is the case the request state is set to done.

Get Sequence: The receiver's chunk gets utilize triggered get operations that trigger on a counter associated with the memory

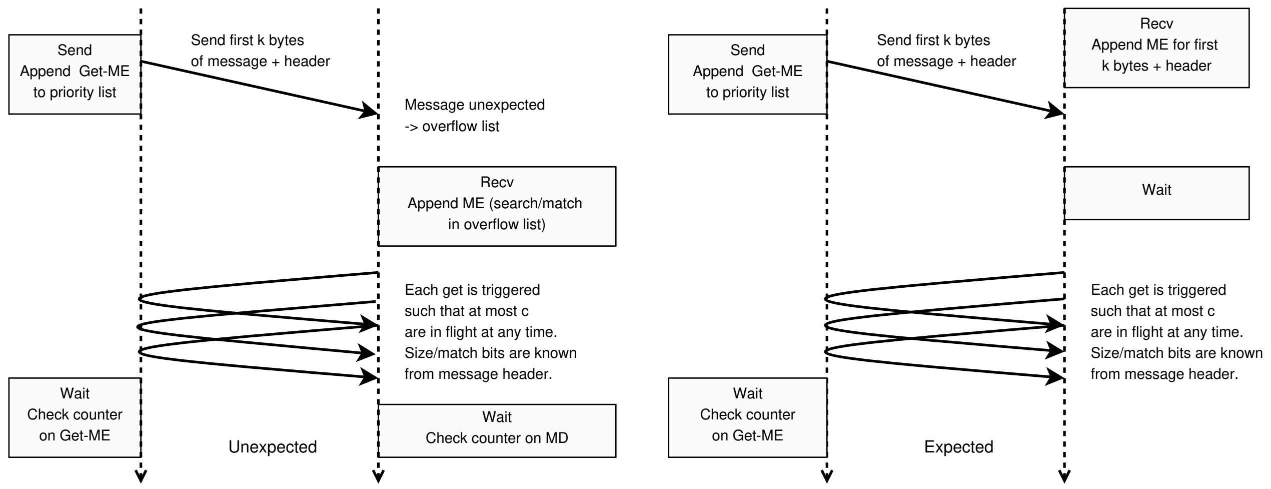

Fig. 3. Timing diagram of the proposed protocol for large MPI messages, for (left) unexpected and (right) expected message cases.

descriptor of the destination buffer. The first c gets are configured to trigger on a value of zero. Each subsequent get $c+i$ is triggered by a counter value of i or more. The value c is thus equivalent to a credit, and there can be no more than c outstanding gets. A c value greater than 1 allows multiple chunk transfers to pipeline and hides the latency of initiating each chunk transfer.

4 Evaluation

Since we do not have access to a hardware implementation of Portals 4, or a large machine with a fat network and NVRAM, we used simulation and emulation to evaluate our protocol. We used Booksim [26] to simulate a cluster with a varying number of slow receivers, and we also implemented the protocol described in Section 3 on top of the Portals 4 reference implementation, running on a dedicated CPU core to simulate hardware offload, similarly to other works evaluating Portals based protocols [20].

4.1 Simulations

We simulate a Dragonfly topology with 5,256 nodes, configured as 73 groups of 72 nodes in Booksim. The simulations use an input queued router model with 2.4x internal speedup and buffer sizes of 256 flits per virtual channel. Latency of a local link is set to 2 cycles, and global link latency is 10 cycles. We use Progressive Adaptive Routing [10] with minimal bias of 2x and threshold of 30 flits. The size of a packet is limited to 16 flits, and larger messages are split into 16 flit MTUs. Relative ratios of buffer size, link latency and packet size are representative of real networks, the chosen absolute values are small to keep simulation times manageable. The simulations run until convergence, i.e., until variation in throughput between successive iterations is less than 5%.

The simulated traffic pattern is a static permutation pattern, where node A sends all of its traffic to a randomly chosen node B, which sends all of its traffic back to A (cf. Effective Bisection Bandwidth metric used in [27], [28]). This permutation pattern is representative of the large message workload (e.g., check-pointing) that is particularly susceptible to the slow node vulnerability.

Results in Figure 1 show the baseline throughput of 0.59 without any slow nodes. However, even with a very small number of nodes sinking traffic at a slower rate (2x slower, or 8x slower), the average network throughput drops dramatically. The drop is the result of the adaptive non-minimal routing algorithm spreading the congestion tree across the entire network in the attempt to route around the congestion caused by the slow nodes. Relying on adaptive routing in this case only makes the situation worse, and instead we need to throttle the injection of the traffic targeting the slow nodes.

A known mechanism that detects and throttles the congestion causing traffic is the explicit congestion notification mechanism, also known as FECN/BECN mechanism [29]. In this mechanism, when congestion is detected in a switch, the packet is marked with the FECN (Forward Explicit Congestion Notification) bit with some probability, depending on the configured marking rate and the level of congestion. When the target node notices a FECN bit set on a packet from node A, it sets the BECN (Backward Explicit Congestion Notification) bit on a packet that it sends back to the same node A (e.g., on the acknowledgment packet). The original node A receives this packet with a BECN bit set, and as a result it throttles its injection. The intent is to throttle the nodes that contribute to congestion in the switches in order to alleviate that congestion.

We implemented the FECN/BECN mechanism in Booksim. Occupancy of the output link's remote queue is used as the congestion metric. Packets can only be marked with the FECN bit if the remote queue is more than 50% full. The probability of marking a packet linearly increases from 0 to 1, as the remote queue occupancy increases from 50% to 100% full. We do not send acknowledgment packets, but instead for every received packet with a FECN bit set, we set the BECN bit on one of the packets that is sent back. This works due to the pairwise nature of our permutation traffic. When a node receives a packet with the BECN bit set, it increments a counter by a value of 8. If the received packet does not have the BECN bit set, the same counter is decremented by a value of 1. This BECN counter is limited to be between 0 and 20, and it is used to determine if, and by how much, to throttle the traffic injection. Injection is throttled linearly with the BECN counter, with no throttling when the counter is at 0, and maximum throttling when the counter is at its maximum value. To avoid converging into a state where nobody sends anything, the BECN counter is also automatically decremented by one every 4 cycles. The counter change values

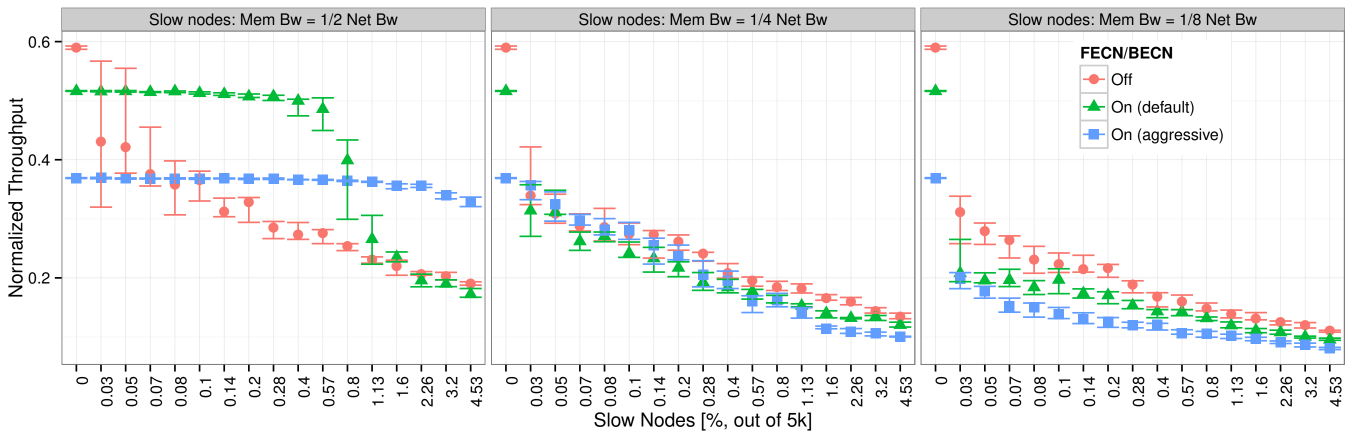

Fig. 4. Throughput in the presence of a small fraction of slow receivers (2x, 4x, and 8x slower than link bandwidth) with different FECN/BECN configurations.

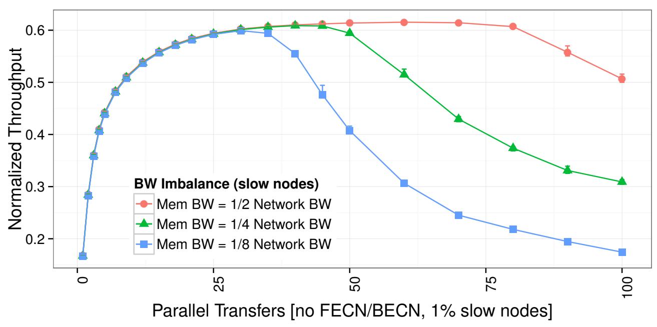

Fig. 5. Throughput of our RDRP protocol in the presence of of 1% slow receivers and a varying number of parallel transfers.

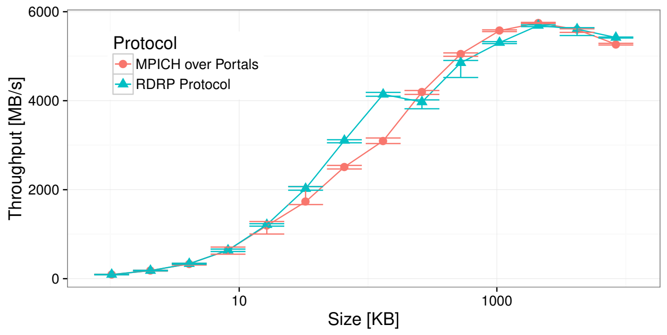

Fig. 6. A comparison between our RDRP protocol and the Portals 4 back-end in MPICH.

were tuned by trial and error, and while our parameters might not be optimal, they are reasonable for our network configuration. We also present simulation results for more aggressive FECN/BECN parameters with 2x higher FECN marking rate probability, and auto decrementing the BECN counter only every 50 cycles.

The results of the FECN/BECN simulation are shown in Figure 4. We show normalized throughput of a permutation traffic pattern, depending on the number of slow receivers. Throughput is shown for the case of slow nodes sinking traffic at 2x (left), 4x (middle), and 8x (right) lower rate than injection bandwidth. We compare the baseline with two sets of parameters for the FECN/BECN mechanism, the default one and a more aggressive version with higher FECN marking rate. The FECN/BECN mechanism is capable of improving the baseline throughput only in the 2x slowdown case, and only for a small number of slow receivers. However, even that limited improvement comes with a penalty in the well-behaved case with no slow receivers, particularly with the aggressive parameter set where the throughput is reduced by 40% in the well-behaved scenario. With higher slowdown of slow receivers (e.g., 8x slowdown), the FECN/BECN mechanism is not only ineffective, but it additionally reduces throughput by up to 50%.

Our RDRP protocol was also implemented in Booksim. In our protocol, each node can only have a limited number of outstanding get requests. To simplify the simulation, instead of generating get requests, we use the original static permutation traffic pattern but we limit the number of outstanding packets for each node. Each node maintains a number of credits, where each credit corresponds to one outstanding packet. As the packets reach their destination, the credits are returned to the sender explicitly by piggy-backing them onto regular traffic. This approach simplifies the simulation, while maintaining the functionality and the round-trip latency nature of get requests.

The results of simulations with our RDRP protocol are shown in Figure 5. We plot normalized throughput versus the maximum number of parallel transfers (i.e. maximum number of concurrent get requests for each node). Each get request transfers 16 flits of data. If the number of parallel transfers is too small, bandwidth is wasted waiting for get-requests/acks, if too many transfers are allowed in parallel, congestion occurs. For the case with 8x slower receivers, 30 outstanding transfers gives the best performance. Using this value will give a performance improvement over the best possible configuration using FECN/BECN and a single big transfer of 1.7x, 3.3x, and 4.3x for 2x, 4x, and 8x slower nodes.

4.2 Implementation in Portals

We implemented our protocol using the Portals 4 reference implementation. We compared our implementation to MPICH 3.2, which also has a Portals 4 transport. Figure 6 shows the result of a simple ping-pong benchmark, which was repeated 25 times for each reported data size, the error bars show the 25th and 75th percentile of the data, the median is shown as well. The test system used consists of Intel E5-2699v3 Xeons, clocked at 2.30GHz. All nodes are connected to a single switch, using Mellanox Technologies MT27500 InfiniBand cards. This system does not contain NVRAM, we perform this experiment only to verify that using multiple small gets (we use 128k sized gets in this experiment) does not degrade performance excessively for transfers that do not target slower elements of the memory hierarchy. We can see that we almost achieve MPICHs point-to-point performance for large messages, we are slightly faster for small messages, since our prototype implementation has fewer overhead, i.e., only supports one transport layer, no argument checking, etc.

5 Conclusions

We have shown that a small number of nodes sinking traffic into a slow tier of the receiver's memory hierarchy can lead to congestion and widespread network performance impacts. We proposed a receiver-driven, bandwidth oblivious rendezvous protocol that transparently adapts to bandwidth mismatch and demonstrated that this approach can successfully prevent congestion caused by bandwidth mismatch. We demonstrate our approach in the context of MPI, the *de facto* standard for HPC messaging. MPI presents significant challenges because of its strict message ordering and matching semantics; thus, while our evaluation has focused on the HPC domain, we are confident that our approach can also benefit other communication models. We compared our high-level solution to different configurations of the FECN/BECN congestion management mechanism and demonstrate much better performance. We show that the overhead (in the absence of slow receivers) of our adaptive protocol are negligible compared to the non-adaptive protocol implemented in MPICH.

References

[1] Intel Corporation, "Intel® solid-state drive DC P3608 series product specification," Order Number: 333055-001US, 2015.

[2] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of nvram in data-intensive architectures: an evaluation," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 703–714.

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 217–228, 2009.

[4] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–11.

[5] A. Malinowski, P. Czarnul, M. Maciejewski, and P. Skowron, "A fail-safe nvram based mechanism for efficient creation and recovery of data copies in parallel mpi applications," in *Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology–ISAT 2016–Part II*. Springer, 2017, pp. 137–147.

[6] S. Kannan, A. Gavrilovska, K. Schwan, D. Milojicic, and V. Talwar, "Using active nvram for i/o staging," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*. ACM, 2011, pp. 15–22.

[7] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® Omni-path Architecture: Enabling scalable, high performance fabrics," in *IEEE 23rd Annual Symposium on High-Performance Interconnects*, ser. HotI 23, Aug 2015, pp. 1–9.

[8] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[9] A. Badam and V. S. Pai, "Ssdalloc: hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011, pp. 211–224.

[10] N. Jiang, J. Kim, and W. J. Dally, "Indirect adaptive routing on large scale interconnection networks," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 220–231. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555783

[11] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage switches are not crossbars: Effects of static routing in high-performance networks," in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 116–125.

[12] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," available at: http://www.mpi-forum.org.

[13] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "The Portals 4.1 network programming interface," Sandia National Laboratories, Tech. Rep. SAND2017-3825, Apr. 2017.

[14] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos, "The BXI interconnect architecture," in *Proc. IEEE 23rd Annual Symposium on High-Performance Interconnects*, ser. HOTI '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 18–25. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2015.15

[15] B. W. Barrett, R. Brightwell, K. S. Hemmert, K. B. Wheeler, and K. D. Underwood, "Using triggered operations to offload rendezvous messages," in *European MPI Users' Group Meeting*. Springer, 2011, pp. 120–129.

[16] M. J. Rashti and A. Afsahi, "Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects," in *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*. IEEE, 2008, pp. 95–101.

[17] G. F. Pfister, "An introduction to the infiniband architecture," *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.

[18] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato, "Congestion control in infiniband networks," in *High performance interconnects, 2005. proceedings. 13th symposium on*. IEEE, 2005, pp. 158–159.

[19] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu, "Congestion avoidance on manycore high performance computing systems," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 121–132.

[20] T. Schneider, T. Hoefler, R. E. Grant, B. W. Barrett, and R. Brightwell, "Protocols for fully offloaded collective operations on accelerated network adapters," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 593–602.

[21] R. L. Graham, B. W. Barrett, G. M. Shipman, T. S. Woodall, and G. Bosilca, "Open mpi: A high performance, flexible implementation of mpi point-to-point communications," *Parallel Processing Letters*, vol. 17, no. 01, pp. 79–88, 2007.

[22] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[23] K. Raffenetti, A. J. Pena, and P. Balaji, "Toward implementing robust support for portals 4 networks in mpich," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 1173–1176.

[24] A. Hassani, A. Skjellum, R. Brightwell, and B. W. Barrett, "Design, implementation, and performance evaluation of mpi 3.0 on portals 4.0," in *Proceedings of the 20th European MPI Users' Group Meeting*. ACM, 2013, pp. 55–60.

[25] R. Brightwell and K. Underwood, "Evaluation of an eager protocol optimization for MPI," in *Proceedings of EuroPVM/MPI*, September 2003.

[26] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, April 2013, pp. 86–96.

[27] T. Hoefler, T. Schneider, and A. Lumsdaine, "Optimized routing for large-scale infiniband networks," in *2009 17th IEEE Symposium on High Performance Interconnects*. IEEE, 2009, pp. 103–111.

[28] ——, "Multistage switches are not crossbars: Effects of static routing in high-performance networks," in *2008 IEEE International Conference on Cluster Computing*. IEEE, 2008, pp. 116–125.

[29] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato, "Congestion control in infiniband networks," in *High performance interconnects, 2005. proceedings. 13th symposium on*. IEEE, 2005, pp. 158–159.