# Light-Weight Protocols for Wire-Speed Ordering

Hans Eberle
*NVIDIA Research*
Santa Clara, CA
heberle@nvidia.com

Larry Dennison
*NVIDIA Research*
Westford, MA
ldennison@nvidia.com

***Abstract*—We describe light-weight protocols for selective packet ordering in out-of-order networks that carry memory traffic. The protocols are designed for heterogeneous high-performance systems, in particular, accelerated systems with end-points that have few resources available for interfacing the network.**

**The protocols preserve the semantics of a relaxed memory ordering model as adopted by highly-threaded many-core processors and accelerators.**

**The protocols achieve link-rate performance through the following techniques: (1) Speculative connection setup avoids round-trip delays found in protocols with little knowledge about endpoint resources, (2) target-side ordering avoids round-trip delays found in source-side ordering mechanisms, (3) fine-grained ordering removes dependencies unwarranted by program code avoiding cumulative ordering dependencies caused by coarse-grained ordering, (4) ordering relaxations and optimizations for producer/consumer communication patterns.**

**We describe two ordering protocols that provide (1) strict sequential ordering and (2) relaxed ordering for multi-packet transfers. The protocols impose no restrictions on routing, including multipath routing.**

***Index Terms*—protocols, memory interconnects, ordering, out-of-order networks**

## I. Introduction

We consider the problem of providing selective packet ordering for an interconnection network that carries memory traffic (loads and stores). We describe ordering protocols that preserve the semantics of a relaxed memory ordering model as used in many-core processors. Memory interconnects have been used, for example, in Cray machines from the T3D [1] through Cascade [2]. Prior packet ordering solutions include forcing deterministic routing for ordered packets (employed by the Cray interconnects and Gen-Z) and using sliding window protocols (as in TCP).

Our protocols target a small-footprint implementation. Specifically, we are interested in accelerated systems that interconnect end nodes that are not equipped with the storage and compute resources needed to run heavy-weight protocols. Accelerators such as GPUs are becoming commonplace in HPC systems [3] and data centers [4], [5]. Other types of accelerators such as ASICS and FPGAs are also increasingly deployed in data centers [6], [7]. Enabling accelerators to communicate with each other directly, without reliance on CPUs, can significantly mitigate communication overheads and bottlenecks [8], [9] when scaling out applications.

In this paper, we focus on an out-of-order network with memory semantics for large-scale accelerated compute systems. We argue that the applicability of memory models and their relaxations has to be expanded such that they cover not only local memory accesses, but also remote memory accesses transported over a network. Applications should specify the ordering rules that apply to the issued memory operations and the underlying transport protocols should expose the different ordering modes. This way, by closely matching application characteristics and hardware capabilities, optimal performance can be achieved[1].

Out-of-order packet delivery is inherent to scalable high-performance networks that employ advanced techniques such as multi-path routing and congestion management. Networks such as Dragonflies, Fat Trees, and Tori rely on adaptive, multi-path routing to balance network load and reduce average forwarding latency. This may lead to packet reordering. It is important to note that adaptive routing is not only critical for low-diameter networks such as Dragonflies but for a wide array of network topologies. In a fat tree topology, for example, there is a choice of links between switches. Furthermore, routing choices even exist at the link level if a subset of lanes can be selected for forwarding packets.

Network congestion management can also lead to reordering. Protocols such as the Last-Hop Reservation Protocol (LHRP) [10] will discard packets to relieve congestion and reschedule their transfer for a later time, thereby delivering packets in an order that is different from the original injection order.

Many considerations influence the design of an ordering mechanism including the frequency of out-of-order packets, throughput and latency goals for ordered traffic, as well as targeted implementation costs. Simple, often-implemented mechanisms include source-side ordering and deterministic routing. Source-side ordering is typically done by serializing packet transfers in that the source waits for a request packet to be acknowledged before the next request packet is injected into the network. This approach limits the transfer rate to at most one packet every network round-trip time (RTT). Another technique often employed restricts ordered streams to a single deterministic path between source and target. Though different streams can still be routed over multiple paths, deterministic routing can have significant performance impacts, in particular, since networks often lack the ability to provide fine-grained

[1]It is beyond the scope of this paper to discuss APIs that closely match the capabilities of the presented protocols.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 

ordering and, with it, are unable to exploit the available path diversity.

Typical use cases that require ordered packet delivery are: (1) Sequential consistency per memory location (SC-LOC), (2) producer/consumer communication patterns including remote queues and Halo exchanges where some independent data transfers are followed by a synchronization operation that indicates the availability of the data, (3) ordered I/O transfers.

## II. Use Cases

Network ordering mechanisms have to be designed with an eye on the endpoints' memory models, their coherence and consistency rules, and the resulting communication patterns. Coherence and consistency models specify the rules that apply to ordering memory operations, both in relation to the processes issuing the operations and the memory locations the operations are accessing. A comprehensive overview of coherence and consistency models with all their intricate nuances is beyond the scope of this paper [11].

Compute systems and, in particular, accelerators such as GPUs are becoming massively parallel with tens of thousands of threads operating in parallel. To exploit the available computational parallelism, ordering relaxations have been introduced to memory models. To optimize overall performance when executing distributed parallel applications, it is critical to be able to pass on ordering relaxations from the endpoint to the underlying network and to perform the, hopefully few, ordered operations at wire speed.

We briefly describe the typical use cases that guided us when devising the ordering protocols.

*Sequential Consistency per Memory Location (SC-LOC):* Highly-threaded many-core processors such as GPUs adopt a relaxed memory model to improve parallelism and, with it, overall performance. Their programming models and compilers typically assume at least sequential consistency per memory location (SC-LOC), that is, strict ordering for operations that access the same memory location.

*Producer/Consumer Communication:* Parallel applications often exhibit producer/consumer communication patterns. A prominent example is a remote queue with an endpoint enqueuing data (i.e., producing data) by writing into remote memory and another endpoint dequeuing data (i.e., consuming data) by reading from local memory[2]. Synchronization between producer and consumer is typically done with the help of a flag that indicates the availability of the associated queue entry. This communication pattern does not require strict ordering for all operations: the synchronization operation has to be ordered relative to the data writes; the data writes, however, can occur in any order. In the following, we will describe a protocol optimized for this use case.

*Ordered IO:* Another important use case is I/O traffic. Some I/O standards and, in particular, some legacy I/O interfaces impose strict ordering requirements. An example is certain classes of PCIe traffic that require in-order delivery. Networks often bridge I/O traffic and thus have to match the corresponding ordering guarantees.

Also, we find I/O interfaces that assume ordered accesses to memory locations or registers that are not explicitly addressed. For example, a FIFO memory might "hide" behind a single address. Similarly, a command "port" might pose as a portal to several hidden registers that cannot be individually accessed but rather have to be written by several sequential writes to the same address.

[2]Alternatively, the queue is located in memory local to the producer and remote to the consumer.

## III. Protocol Description

Before we describe the protocols in detail, we outline the techniques that we applied to accelerating ordering and the assumptions that underlie the design of the protocols.

### A. Acceleration Techniques

To achieve wire-speed operation, our protocols employ several techniques that accelerate packet ordering. In the following, we introduce these techniques as they play an integral part in the design of the protocol.

*Target-Side Ordering:* Ordering can be enforced at the source or at the target. Our protocol orders packets at the target (with the help of a reorder buffer) allowing for overlapped packet transfers at the full link rate. The alternative is to order packets at the source by serializing packet transfers in that one packet at a time is transferred. Source-side ordering is often employed in the absence of ordering hardware by having software serialize packet transmission. When software assists ordering, overheads will likely greatly exceed network delays. Throughput of source-side ordering is less than 1/RTT.

Fig. 1 illustrates the two choices. Ordering at the source (a) serializes transfers in that request $R_n$ can only be sent after acknowledgment $A_{n-1}$ was received. The example shows three requests that need to be delivered in the order $R_0$, $R_1$, $R_2$. Ordering at the target (b) allows for overlapping transfers.

*Fine-Grained Ordering:* Our protocol allows for fine-grained ordered packet streams that can be independently scheduled. By isolating ordering domains and individually mapping them to separate ordered streams, unnecessary ordering dependencies are avoided. This is illustrated in Fig. 2. The sequence diagrams show two ordered packet sequences (green/solid and red/dashed) where the only requirements are that $R_2$ must follow $R_0$ (green/solid) and $R_3$ must follow $R_1$ (red/dashed). With coarse-grained ordering (a), where packets are transferred as part of a single ordered stream, unnecessary dependencies are created ($R_2$ waiting for $R_1$). With fine-grained ordering (b), only real dependencies cause delays ($R_3$ waiting for $R_1$).

*Target-Side Synchronization:* Many producer/consumer communication patterns use release semantics in that a synchronization operation follows several data transfers (as discussed in Section II). Noticing that the data transfers can be unordered and that the only ordering requirement is that the synchronization operation is executed after all data transfers have finished, our protocol places accounting logic at the

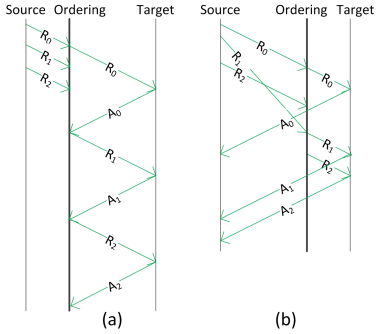



Fig. 1: Source-Side (a) vs. Target-Side Ordering (b).

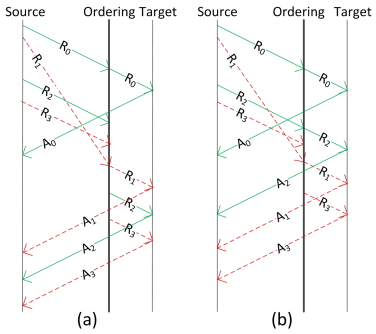



Fig. 2: Coarse-grained (a) vs. Fine-grained Ordering (b).

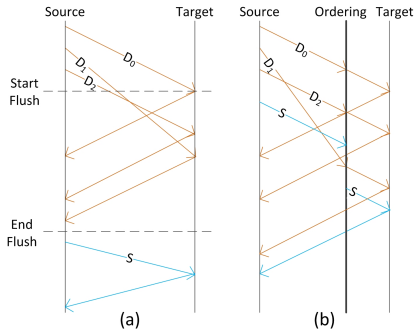



Fig. 3: Source-Side (a) vs. Target-Side Synchronization (b).

target to hold back the synchronization operation until all associated data operations have become visible. An alternative employed by some interconnects is to use a flush operation to bring outstanding data transfers to a conclusion before the synchronization operation is injected.

Fig. 3 illustrates source-side and target-side synchronization. The sequence diagrams show a packet stream consisting of three data transfers $D_0$, $D_1$, $D_2$ followed by a synchronization operation S. While $D_{0..2}$ can be delivered in any order, S must be delivered only after $D_{0..2}$ have become visible. With source-side synchronization, the source releases S after flushing the data transfers $D_{0..2}$ (a). With target-side synchronization, S is held at the target and released immediately after all data transfers $D_{0..2}$ have been accounted for (b).

### B. Design Assumptions

We have designed two separate protocols that we refer to as *ordered transfers* and *synchronized transfers*. While ordered transfers impose strict sequential ordering on packet delivery, synchronized transfers are intended for multi-data packet transfers that are followed by a synchronization operation where the only ordering requirement is that the synchronization operation is delivered after the transfer of the data packets has completed, allowing the delivery of the data packets to be unordered.

The design of the protocols is based on the following assumptions.

*Load/Store Model:* The network implements a load/store model where remote memory can be accessed like local memory with the same load/store semantics. Network packets thus transport memory operations. Ordering requirements apply similarly to memory accesses locally and remotely.

*No Response Ordering:* Ordering is concerned with ordering packets delivered to the target endpoint only. No ordering has to be applied to response packets returned to the source as we assume that the source already has mechanisms in place to reorder responses if needed[3].

[3]Processors are equipped with logic to deal with replies from the memory subsystem and can reuse this logic for out-of-order replies received over the network. Typically, the source keeps track of outstanding requests with the help of unique tags associated with the requests and corresponding replies. The tags are used to match replies with requests and are further used to order replies if needed.

*Unreliable Transport Layer:* No reliable transport service is provided. Thus, ordering has to be capable of recovering from packet loss.

Though unordered data transfers are not the topic of this paper, a corresponding transfer protocol, similarly, has to provide end-to-end reliability, for example, with a simple request/acknowledge protocol.

*Fixed Packet Lifetime and Failure Model:* Packets in the network have a maximum lifetime. If a packet is not ejected to an endpoint within its lifetime, it is lost and will never emerge from the network. Packet lifetime is enforced by a maximum hop count and a maximum switch and link-traversal time. We assume a non-byzantine failure model where packets in the network can be lost but never duplicated.

### C. Protocol Highlights

Our protocols are characterized by the following major features:

*Slow and Fast Modes:* The ordered transfer protocol supports both source-side ordering and target-side ordering as introduced in Section III-A. Given the performance characteristics of the two modes, we refer to the former as *slow mode* and the latter as *fast mode*. There are two use cases for slow mode. Firstly, ordered sequences of packets injected by the source at a low rate can be transmitted in slow mode thereby avoiding the allocation of connection resources needed by fast mode. Secondly, when connection resources are exhausted, the protocol falls back to slow mode so that progress can still be made.

*Light-Weight Connections:* Connections are light-weight in that they are quickly set up and remain active only as long as there are outstanding requests. Connections are speculatively set up in that the source begins to burst packets into the network as soon as they become available assuming that connection state can be allocated at the target[4]. No explicit handshake between source and target is needed to set up a connection so that packets can be forwarded from the source to the target without delay.

Connections are typically short-lived and torn down as soon as there are no more outstanding requests. By only having

[4]Connection resources are required at both the source and the target.

active connections for packets that are in flight, resource needs depend on the bandwidth-delay product of the network only and are independent of the number of endpoints. This is an important feature as it provides for a scalable design.

*Small Reorder Buffers:* A small reorder buffer suffices that is capable of absorbing a network skew worth of traffic. The buffer capacity neither depends on the number of network nodes nor the number of connections allowing for a scalable design. Modest requirements are particularly critical when targeting resource-constrained NIC implementations for coprocessors and accelerators.

*No Timeouts:* In the absence of errors, our protocols do not rely on any timeouts or network delays[5,6]. Specifically, connections are explicitly closed to make resources available as soon as they are no longer needed. An explicit handshake is used to deallocate resources at the source and the target to avoid lengthy timeouts that prevent resources from being freed.

*Exactly-Once Delivery:* Packet duplicates can be created as the result of packet retransmissions. Duplicates can lead to faulty program behavior if packets deliver non-idempotent operations such as atomic operations or synchronization operations. Thus, an option is needed to detect duplicates[7]. We call this option *exactly-once delivery*. To determine that a packet is a duplicate, state is needed that registers the original copy of the packet. We leverage connections for this purpose. In addition, a resource in the form of replay buffers is needed to replay responses when a duplicate request is detected.

TABLE I: Nomenclature for Sequence Diagrams

Packet types (the packets in the sequence diagrams are labeled with indices to help the reader follow the flow of packets):

- *REQ*: Request packet.
- *ACK*: Acknowledgment packet. Acknowledges receipt of REQ.
- *NACK*: Negative acknowledgment packet. Indicates that REQ was rejected and instructs the source to retransmit the corresponding REQ.
- *FIN*: Finalize packet. Instructs the target to close the connection.
- *FIN-ACK*: Finalize acknowledgment packet. Signals to the source that the target has closed the connection.

Packet flags:

- *SYN*: Synchronize sequence number counter. Indicates to the target that the value of the SEQ field contained in the packet is to be used as the starting sequence number.
- *EOD*: Exactly-once delivery.
- *CON*: Connection has been established.

Other packet parameters:

- *CID*: Connection identifier.
- *SEQ*: Sequence number.
- *CNT*: Count of all request packets in a synchronized transfer.

[5]Timeouts are used for detecting erroneous or lost packets.

[6]We assume that packets "dropped" by congestion management are NACKed and thus do not time out.

[7]We consider duplicate packets received by the target only as duplicate packets received by the source do not lead to faulty program behavior.

### D. Ordered Transfer Protocol

The ordered transfer protocol is intended for use cases such as SC-LOC and ordered IO transfers.

We use sequence diagrams to describe the protocols[8]. Table I provides the nomenclature used in the diagrams.

*1) Ordered Transfer, Slow Mode:* Slow-mode ordered transfers serialize transfers in that there can be only one outstanding REQ packet and a REQ packet can only be sent after the ACK for the previous REQ was received. We call the resulting packet exchanges *one-at-a-time transfers*. Fig. 4 shows an example. The first transfer completes successfully when the source receives $ACK_1$ in return for $REQ_1$. The second transfer, however, does not complete as $ACK_2$ is lost. After $REQ_2$ times out, it is resent, and when it arrives at the target it is again forwarded to the target memory. Note that $REQ_2$ is delivered to the target memory twice. This is acceptable as exactly-once delivery was not specified. This time, $ACK_2$ arrives back at the source and completes the transfer.

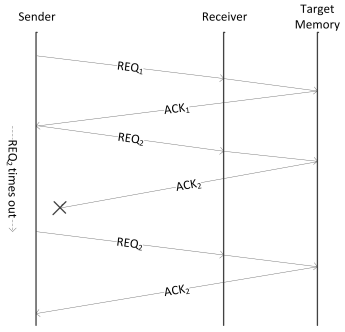



Fig. 4: Ordered Transfer, Slow Mode

*2) Ordered Transfer, Fast Mode:* Fast-mode ordered transfers overlap packet transmission. Both the source and target use connections to keep track of (multiple) outstanding REQs and to establish order when packets arrive out-of-order.

Fig. 5 shows an example of a sequence of three request packets $REQ_{1\text{-}3}$ transmitted as a fast ordered transfer. The source begins with sending a sequence of three REQ packets. The first REQ packet of a sequence includes the SYN flag to inform the target of the start of a new sequence. Each REQ packet contains the connection identifier (CID) and a sequence number (SEQ). Since CID is unique only at the source and not at the target, the target uses the tuple {source endpoint ID[9], CID} as a unique identifier of the connection and the associated state.

Transmission begins with the source sending a burst of REQ packets. Packets arrive out of order in that $REQ_1$ takes a longer path and arrives after $REQ_2$ and $REQ_3$. This is a typical scenario that occurs when packets are adaptively routed. $REQ_2$ arrives first and causes the target to open a connection as no

[8]Due to space constraints the description of the protocols can only cover regular operation and is omitting many possible corner cases.

[9]The sequence diagrams do not show the source and target endpoint IDs which are specified in every packet.

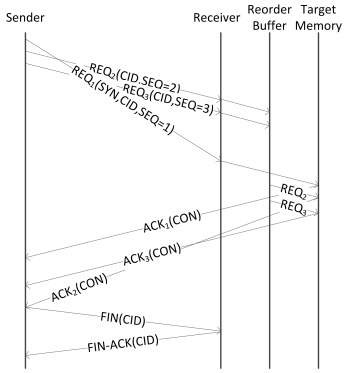



Fig. 5: Ordered Transfer, Fast Mode

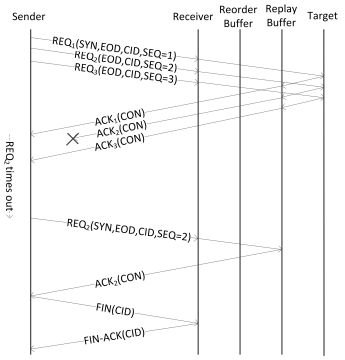



Fig. 6: Exactly-Once Delivery

connection is yet in place — as the example shows, the first REQ received opens a new connection, whether it has the SYN flag set or not. Since $REQ_2$ and $REQ_3$ are received out of order, they are kept in the reorder buffer until $REQ_1$ is received. When $REQ_1$ finally arrives at the target, it is directly forwarded to the target memory. The arrival of $REQ_1$ further unblocks the delivery of $REQ_2$ and $REQ_3$. Once the operations of the REQ packets have been reflected in the target memory, the corresponding ACK packets are returned.

Once all outstanding request packets have been acknowledged, the connection is explicitly closed with a FIN/FIN-ACK exchange. Note that ACKs can be reordered when they arrive at the source as is shown in Fig. 5. It is, therefore, not sufficient to wait for the ACK acknowledging the last sent REQ to determine when the FIN can be sent. Rather, the source needs to pair every outstanding REQ with the corresponding ACK before the connection can be closed.

The purpose of the FIN/FIN-ACK exchange is to explicitly close the connection and free the associated state at both the source and target so that it can be reused for another transfer. Note that the FIN/FIN-ACK exchange is only initiated after it is guaranteed that no data REQ or ACK packet is left in the network. Thus, the target can safely close the connection upon receipt of FIN. Connection closure completes with the source receiving FIN-ACK and removing its connection state (including freeing the connection ID). Note that optimizing the protocol by merging FIN with the last data REQ is not feasible as connection state is removed that might be needed to detect a duplicate packet. Also, note that replacing the FIN/FIN-ACK exchange with a timeout that triggers connection closure is inefficient in that connection state remains tied up longer. We consider connection state a critical resource that indirectly determines the number of outstanding REQs.

Though not a requirement, if the target returns the ACKs in order, ACKs can also function as fences. For example, when the source receives $ACK_3$ in the scenario of Fig. 5, it is guaranteed that $REQ_1$, $REQ_2$, and $REQ_3$ have become visible in the target memory[10].

[10]We assume that memory operations become visible in the same order the corresponding packets are received by the memory subsystem. We broadly use the term *target memory* to refer to the target memory subsystem.

*3) Exactly-Once Delivery:* Fig. 6 shows the transfer of three ordered requests $REQ_{1\text{-}3}$ requiring exactly-once delivery (as indicated by the EOD flag). The sequence diagram begins with the transfer of the three REQs from the sender to the receiver and ultimately to the target. Upon receipt, the target executes the operations contained in the REQs, stores the results in replay buffers and returns the results in $ACK_{1\text{-}3}$. To demonstrate how the replay buffers come into play, $ACK_2$ is lost in the example. As a result, a timeout causes the sender to resend $REQ_2$. The receiver recognizes $REQ_2$ as a duplicate and regenerates $ACK_2$ by replaying the result by retrieving it from the replay buffer. A duplicate REQ must not be forwarded to the target as it contains a non-idempotent operation whose re-execution could generate a result different from the original one.

When the sender receives $ACK_2$, all REQs have been acknowledged and the connection can be torn down by initiating a FIN/FIN-ACK exchange.

Similar to the reorder buffer, the replay buffer is a resource that requires allocation and deallocation. When a REQ is forwarded to the target, a replay buffer needs to be allocated. And once it is guaranteed that a REQ cannot be resent, the replay buffer holding the corresponding ACK can be freed. In the discussed example, $ACK_1$ can be removed from the replay buffer when $REQ_2$ with the SYN flag set is received and $ACK_2$ and $ACK_3$ can be removed when FIN is received. Due to space constraints, we cannot provide more details on the management of the replay buffer including details on handling resource exhaustion.

### *E. Synchronized Transfer Protocol*

A synchronized transfer corresponds to a multi-data packet transfer followed by a synchronization operation as found in producer/consumer communication patterns. While data packets can be delivered in any order, the packet with the synchronization operation must only be delivered after all data packets were delivered.

An example of a synchronized transfer is given in Fig. 7. The first three requests $REQ_1$, $REQ_2$, and $REQ_3$ constitute the multi-data packet transfer while the last request $REQ_4$ contains

the synchronization operation[11]. Since the synchronization operation is a non-idempotent operation, the EOD flag is set. As shown in the example, the packets transporting the data can be directly forwarded into target memory upon arrival, even if they arrive out of order. Since the synchronization operation contained in $REQ_4$ must not be executed before the transfer of the data packet completes, it is held back in the reorder buffer. The receiver counts the ACKs returned from the target memory (or target memories) to determine that the data packet transfers have completed. The number of ACKs to be accounted for is given by the count CNT provided in the REQ containing the synchronization operation ($REQ_4$ in the discussed example). In the given example, the receiver determines that the transfers of the data packets and the synchronization operation are complete after it has counted four ACKs. At this time $REQ_4$ (containing the synchronization operation) can be forwarded to the target memory. Once all REQs have been acknowledged, the connection is torn down by a FIN/FIN-ACK exchange. Note that ACKs can be reordered and that it is not sufficient to initiate the FIN/FIN-ACK exchange when $ACK_4$ is received. Rather, all ACKs have to be accounted for before the connection is closed.

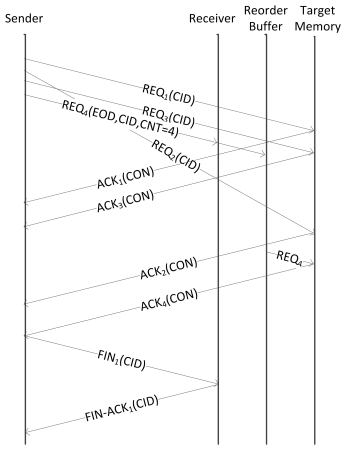



Fig. 7: Synchronized Transfer

## IV. Evaluation

We have evaluated the ordering protocols using Bksim, a successor of the cycle-accurate network simulator Booksim [12]. We use the Dragonfly topology for our simulation runs. We chose this topology over other topologies such as Fat Trees as packet latencies are more skewed and out-of-order delivery is more frequent thus putting stress on the ordering protocol.

We use a 1,056-node Dragonfly network with full bisection bandwidth. The network consists of 33 groups of eight 15-port routers. A router is connected to four endpoints, to seven other routers via local channels and to four groups via global channels. The local channels have a latency of 40 ns and the global channels have a latency of 500 ns - the simulated channel latencies model short intra-cabinet local links and long inter-cabinet links, respectively. No other latencies such as serialization or deserialization delays are explicitly modeled assuming that they are subsumed in the channel latencies. The bandwidth of the channels is 1 flit/ns.

Given the sparse global connectivity, Dragonfly networks rely on adaptive routing to maximize available bandwidth and avoid congestion. Our simulation model implements the progressive adaptive routing algorithm PAR6/2 [13] and uses six virtual channels to prevent routing deadlock.

If not specified otherwise, reported latency and throughput numbers are steady-state mean (average) values. The bar charts also show the population standard deviation.

### A. Source-Side vs. Target-Side Ordering

Our first experiment compares source-side and target-side ordering. We look at three transfer modes:

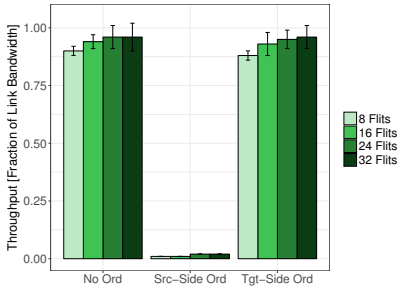
- *No Ordering:* Packets are delivered in the order they arrive. No ordering is applied and, thus, no ordering overhead is incurred. This mode represents the best-case performance scenario.
- *Source-Side Ordering (Ordered Transfer, Slow Mode):* Packets are serialized at the source. We only consider network delays and ignore any delays in the endpoints. That is, no delays occur when the target returns an ACK in response to a REQ or when the source injects the next REQ after it received an ACK for a previous REQ. We use the ordered transfer protocol in slow mode to simulate source-side ordering.
- *Target-Side Ordering (Ordered Transfer, Fast Mode):* Packet transmission is overlapped in that REQs are injected into the network as quickly as possible after they have become available in the source endpoint. Ordering is performed at the target with the help of a reorder buffer. We use the ordered transfer protocol in fast mode to simulate target-side ordering.

In our experiment, there is a single source with a single rank sending flows of packets to a destination in another group at the maximum accepted injection rate. There are 32 possible destinations, all located in the same group. The selection of a stream's destination is uniform random.

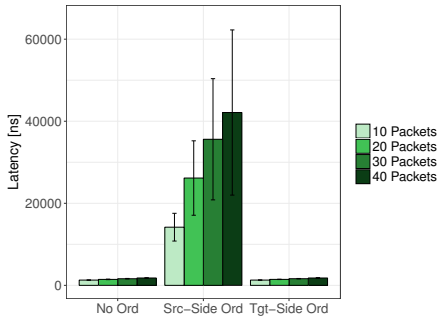
Fig. 8(a) shows the maximum accepted load as a function of packet length for each transfer mode. Two observations can be made. First, no ordering and target-side ordering achieve similar throughput which is close to the maximum link throughput. The graphs thus validate that our protocol provides ordering with negligible processing overhead. It can further be seen that packet length does not impact performance. Second, as expected, source-side ordering limits transfer rates to a small fraction of the available link bandwidth.

In the next experiment shown in Fig. 8(b), we determine the latency for a single ordered stream. The setup is the same as in Fig. 8(a). As we want to characterize the latencies inherent to the examined transfer modes, there is no other traffic present in the network that could cause congestion. Latencies are determined as the time interval between the injection of the

[11]Not shown is the packet header field that specifies whether the REQ belongs to an ordered transfer or a synchronized transfer.




(a) Accepted Throughput




(b) Stream Latency

Fig. 8: Source-Side vs. Target-Side Ordering: Three ordering modes are considered: no ordering, source-side ordering and target-side ordering. Graph (a) shows the accepted throughput (larger values are better). Streams have a fixed length of 10 packets. Packet length is varied from 8 to 32 flits. Graph (b) shows the stream latencies (first packet in to last packet out). Stream lengths are varied from 10 to 40 packets with each packet having a fixed length of 16 flits.

first packet at the source and the delivery of the last packet at the target.

As expected source-side ordering exhibits by far the largest latencies as a result of serializing packet transmission. Packets can at best be injected every network RTT. Total latency thus increases linearly with stream length. No ordering and target-side ordering show similar latencies demonstrating again that ordering can be accomplished with little latency overhead. These results are not surprising, they simply validate that fast mode provides wire-speed performance.

In both experiments, we find that about one third of the packets in fast mode arrive out of order. Though there is no congestion in the network, the high load leads to buffer levels in the network switches reaching the threshold that causes packets to be adaptively routed.

### B. Ordering Granularity

In the next experiment, we examine the impact of ordering granularity on performance and resource usage. In this section, we consider target-side ordering only as provided by the ordered transfer protocol in fast mode. The setup uses 16 source endpoints located in one group simultaneously injecting ordered streams to 16 destination endpoints located in another group in a 1:1 mapping. Each source injects a total of 128 packets evenly divided into a number of streams with the same destination.

The results are shown in Fig. 9. The bar chart on the left shows the total latency for all 128 packets injected by a source. Total latency is determined as the interval between the very first request packet injected at the source and the very last request packet received at the destination. The bar chart on the right shows the mean latencies of the request packets. The number of out-of-order packets varied from 31% (16 packets/streams) to 67% (128 packets/stream) - the percentages are vastly different as a packet routed over a longer path has the potential to hold up more packets in the reorder buffer for longer streams. Latencies are pretty much the same for all four scenarios. These results do not confirm our expectation that finer granularity reduces dependencies and, with it, latencies. More specifically, we expected to see shorter latencies for shorter streams. Our explanation is that latency penalties created by additional dependencies as shown in Fig. 2 are negligible relative to network latencies.

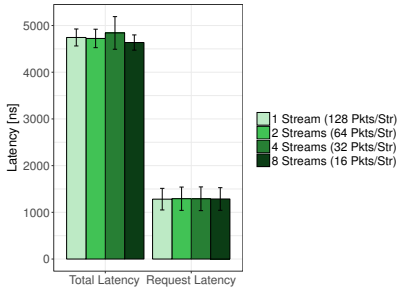



Fig. 9: Ordering Granularity: Shown is the total latency for all packets transferred between a source and destination as well as latencies for individual request packets for different stream lengths (smaller values are better). The number of transmitted packets is varied from 16 to 128 and the number of corresponding streams is varied from eight to one. Packets have a fixed length of 16 flits.

### C. Source-Side vs. Target-Side Synchronization

Next, we want to compare source-side and target-side synchronization as needed by the type of producer/consumer communication patterns described in Sections II. We conducted an experiment that determines the latency for a single stream of packets transferred from one Dragonfly group to another one with the following three transfer modes:

- *Source-Side Synchronization:* Data packets are delivered unordered. A flush operation is executed after all data packets have been injected and before the synchronization operation is injected. This guarantees that the synchronization operation is ordered with respect to the data packets.
- *Target-Side Synchronization (Synchronized Transfer):* This mode uses the synchronized transfer protocol that guarantees that the synchronization operation is held in a target-side buffer until all data packets have been delivered.
- *Target-Side Ordering (Ordered Transfer, Fast Mode):* All packets including data operations and synchronization operations are delivered in order. This mode uses the ordered transfer protocol.

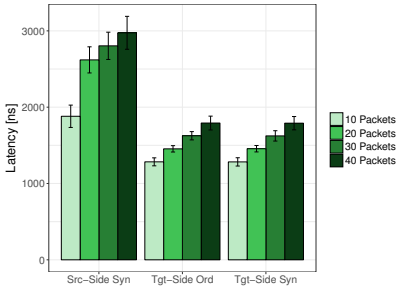



Fig. 10: Source-Side Synchronization vs. Target-Side Synchronization: Three transfer modes are considered: source-side synchronization, target-side synchronization, and target-side ordering. The graph shows the latency for a stream of packets (determined as the interval between the injection of the first data packet at the source and the delivery of the synchronization operation at the target). Stream lengths are varied from 10 to 40 packets with each packet having a fixed length of 16 flits.

The graph in Fig. 10 shows the stream latencies for the three modes and different stream lengths. As expected, source-side synchronization incurs higher latencies than the other two modes using target-side synchronization and target-side ordering, respectively. The higher latencies for source-side synchronization are caused by the flush operation that requires the source to account for the receipt of all ACKs in response to the injected REQs before the synchronization operation can be sent. Compared with target-side synchronization, this adds at least a full RTT (half a RTT to receive the last ACK and another half of a RTT to deliver the synchronization operation).

The experiment also considers target-side ordering as strict ordering implicitly provides synchronization. This mode offers stricter ordering than required as it orders all packets including data packets. In the experiment, about one third of all packets received are delivered out of order. By including this mode, we can show that target-side synchronization offers the same performance as target-side ordering though at much lower cost.

### *D. Reorder Buffer*

Our final experiments examine the reorder buffer. More specifically, we determine the resource needs at the receiver in terms of reorder buffer capacity and number of connections.

Our protocol requires a relatively small reorder buffer large enough to absorb packets from the network for a duration that is equivalent to the network skew. Network skew refers to the difference in end-to-end packet latencies. Latency differences are mainly caused by adaptive routing. For the considered Dragonfly topology and adaptive routing strategy, direct and indirect routes differ by a traversal through an intermediate group. Thus, for our simulations, we assumed a skew of 50 packets to cover the latency of the extra global channel (500 ns or 30 packets) and some switch buffer latency (20 packets)[12].

[12]We are not considering additional delays caused by network congestion when sizing the reorder buffer as we assume that congestion management and admission control will largely avoid oversubscription and make congestion the exception.

To provide O(1) access times when writing and reading the reorder buffer, we propose to implement the reorder buffer as a ring buffer indexed by the output of a hash table whose key is given by the tuple {source device id, CID, SEQ}. This organization allows for checking the presence of the next in-order packet when a packet arrives in a fixed amount of time. Hash table collisions can be mitigated by the addition of a small associative cache and the remaining collisions can be treated as reorder buffer overflow.

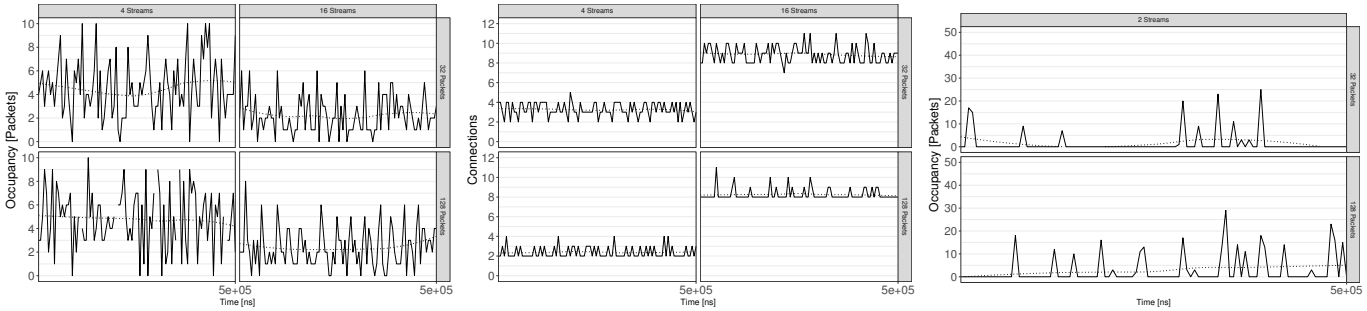
The setup for the first experiment has two sets of senders in one Dragonfly group with each set injecting ordered streams destined for a receiver in another Dragonfly group. Both sets of senders use the same global channel for minimal routes. Hence, there is pressure on the corresponding output queue of the switch feeding the minimal global channel forcing packets to be adaptively routed and possibly arrive out of order. This way, we avoid endpoint congestion as well as oversubscription of any network resources as our simulation model does not implement any congestion management beyond adaptive routing. The first experiments assume that the sender as well as the receiver never run out of connections.

Fig. 11(a) illustrates reorder buffer occupancy over the course of the simulation run for different numbers of streams and different stream lengths. We observe that the number of used reorder buffer elements overall is relatively small and does not significantly change as the number of streams and the stream lengths are varied. The graphs confirm that buffer occupancy and buffer capacity do not depend on the number of active streams. Despite the low occupancy levels, there is a significant fraction of packets that arrives out of order. For four streams, we determined that about one third of the packets are inserted into the reorder buffer and for 16 streams about one quarter. We notice that there is not necessarily a correlation between buffer occupancy and the number of out-of-order packets.

As mentioned, the setup for this experiment did not limit the number of available connections. We would thus expect the number of active receiver connections to match the number of simultaneously injected streams. As shown in Fig. 11(b), we observe slightly higher numbers of active receiver connections. Though each sender injects only one stream at the time, reordered packets can cause sequentially injected streams to overlap at the receiver and require two active connections during this transition period.

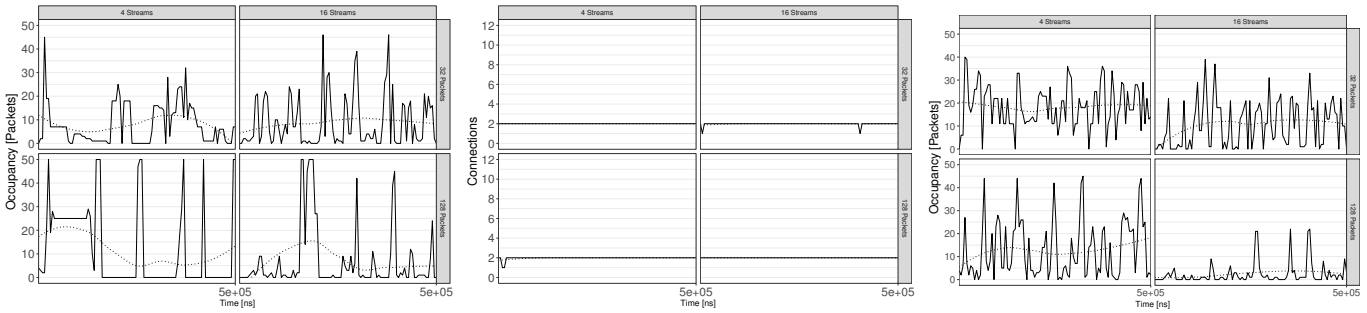
To apply more pressure to the reorder buffer and force higher occupancy levels, we rerun the previous experiment with two sources injecting the ordered streams as bursts of back-to-back packets. This way, the receiver receives packets at the full link rate during short periods of time - the load per destination is still an average of 0.35 of the full link capacity (0.7 combined as in the previous experiment) to avoid endpoint congestion. We further decreased the likelihood of adaptively routed packets[13]. With this experiment, occupancy levels now

[13]Against our initial intuition, fewer adaptively routed packets lead to higher reorder buffer usage as occasional out-of-order packets potentially hold up more packets in the reorder buffer than more frequent out-of-order packets.




(a) Reorder Buffer Occupancy for UR Traffic (b) Active Receiver Connections for UR Traffic (c) Reorder Buffer Occupancy for Bursty Traffic

Fig. 11: Resource Usage for Receiver with Unlimited Number of Connections: 4 and 16 sources send one ordered stream each to two destinations (graphs show reorder buffer occupancy for one destination). The sources are located in one Dragonfly group and the destinations are located in another group. Streams have a fixed length of 32 and 128 packets, respectively. Packets have a fixed length of 16 flits. The combined traffic load is 0.7 of the full link capacity. The reorder buffer has a capacity of 50 packets. Packet distribution is uniform random (UR) in (a) and (b) with graphs (a) showing reorder buffer occupancy and graphs (b) the number of active receiver connections over time. Graphs (c) show reorder buffer usage for bursty packet injection. The dotted lines are trend lines (calculated as conditional mean).




(a) Reorder Buffer Occupancy w/o Throttling (b) Active Receiver Connections w/o Throttling (c) Reorder Buffer Occupancy with Throttling

Fig. 12: Resource Usage for Receiver with Limited Number of Connections: Same configuration as in Fig. 11 except that each receiver has only two connections available. Graphs (c) show reorder buffer usage when the number of outstanding packets is limited to 25 packets per stream. The dotted lines are trend lines (calculated as conditional mean).

reach about half of the reorder buffer capacity (Fig. 11(c)).

The final experiment limits the number of receiver connections to two to demonstrate protocol operation when resources are scarce. Otherwise, the setup is identical to the previous experiment. Results are shown in Fig. 12. We now observe large fluctuations in buffer occupancy. Occasionally, the buffer fills up and packets are dropped - about 5% of the packets are dropped for the scenario with four streams and a stream length of 128 packets. This happens when a receiver connection becomes available only after the first packets of a stream have arrived. In this scenario, later packets are streamed into the reorder buffer in fast mode while earlier packets are delivered in slow modes. This can last for several RTTs filling up the reorder buffer quickly. Several solutions can be considered to limit the number of packets going into the reorder buffer. Fig. 12(c) demonstrates a simple solution that limits the number of outstanding request packets per sender connection to half the reorder buffer size[14,15].

[14]Future work needs to investigate such techniques in more detail.

[15]Limiting a connection's use of the reorder buffer capacity is also needed to contain similar effects caused by the loss of an erroneous packet.

## V. Related Work

Though other protocols can deal with packet reordering, their use cases are quite different from the memory-semantic data center interconnect assumed here. TCP illustrates this well. It is optimized for wide area networks with long control loops and endpoints whose characteristics including available resources are often unknown. Since TCP has little knowledge about endpoint resources, techniques such as speculative connection set up or injecting packets into the network at the maximum possible rate right away when a connection is opened, are not feasible as they could easily lead to resource oversubscription. Further, TCP is not optimized for multipathing as it attributes out-of-order packets to congestion. When reordering is caused by multipathing, TCP performs poorly as it attributes segment timeouts to congestion rather than multipathing and, in response, decreases the number of segments that can be outstanding at the sender [14]. If timeouts are caused by multipathing, this behavior, however, is counterproductive. Recognizing that different signaling mechanisms are needed to disambiguate delayed packet delivery caused by congestion and multipathing, protocol extensions such as Multipath TCP [15] have been proposed.

Other HPC interconnects enable adaptive routing for unordered packet streams only and restrict ordered packet streams to deterministic routes. For example, Cray Aries [2] provides ordered delivery for request packets that access the same address (SC-LOC) by using deterministic routing. In a similar way, if ordering is required in the Blue Gene/L torus network [16], packets are either deterministically routed, or adaptively routed and ordered by software in the endpoint [17].

Exposing different transfer and ordering modes at the API allows applications and communication libraries to better utilize available network resources. For example, the Blue Gene/L torus network [16] offers both adaptive and deterministic routing, and the Cray Gemini [18] and Aries networks [2] expose both relaxed and strict ordering at the network API. In [19], the performance of one-sided and two-sided communication is evaluated with respect to the different transfer modes offered by the CRAY Gemini interconnect. Significant performance improvements are reported for relaxed ordering over strict ordering. The authors conclude that out-of-order message delivery has to be exposed at the application level to optimize system performance.

Gen-Z [20] is a newly emerging general-purpose interconnect that also adheres to memory semantics and adopts relaxed ordering. Gen-Z implements an unordered interconnect that supports multipathing. In-order delivery is provided through so-called *strong order domains*, that restrict traffic to single pathing.

To efficiently use networks that offer different ordering modes, applications need to have awareness of the underlying transport mechanisms and further need ways to specify ordering domains and applicable ordering rules. Newer languages like UPC [21] recognize this. UPC programs let the programmer specify whether accesses are destined for private or shared memory and, further, whether data adheres to either strict or relaxed consistency making it possible for the compiler and libraries to optimize corresponding data transfers. UPC provides the kind of primitives needed to efficiently interface protocols such as ours. For example, the sequence of UPC operations *put_nb, put_nb, ..., syncnb* [22] can be directly mapped to the synchronized transfer protocol and thus benefit from path diversity and adaptive routing as well as low latency thanks to target-side synchronization logic.

PGAS programming models and other one-sided communication models are gaining popularity as they relax ordering constraints. The case study in [23] shows that decoupling data transfers and synchronization is a much better fit for unordered networks. Communication libraries like NVSHMEM also recognize the need for relaxed ordering in networked GPU clusters to reduce orchestration between computation and communication [24].

A semantic gap exists between the relaxed memory models implemented by many-core processors and (distributed) programming frameworks that makes it difficult to exploit available concurrency. The lack of rigorous specifications [25] often leads to over-constrained assumptions about the underlying memory model. Furthermore, programming frameworks as well as network interfaces often provide only crude ordering mechanisms such as memory barriers. The high cost of memory barriers is, for example, shown in [26].

## VI. Conclusion

Many-core processors rely on memory models with relaxed ordering. With the emergence of network interconnects based on memory semantics, relaxed memory consistency models should apply equally to local and remote memories. To optimally use the memory subsystem, applications specify the ordering rules that apply to the issued memory operations. A similar interface between application and the network transport layer of the network is needed so that the ordering constraints of the application and the ordering modes supported by the network can be closely matched.

We have introduced two ordering protocols: The *ordered transfer protocol* provides strict packet ordering and the *synchronized transfer protocol* offers relaxed ordering for producer/consumer communication patterns. We further described an *exactly-once delivery* option for removing packet duplicates. The protocols share a common set of mechanisms thereby simplifying implementation and verification.

We designed the protocols such that their critical paths are free from any network traversal times or timeouts. A speculative connection setup procedure is used that avoids delays caused by a handshake between source and target prior to data transmission.

The protocols are light-weight enabling implementations with a small footprint for nodes as for example found in accelerated systems. Specifically, resource requirements are minimized in that connection state depends on the bandwidth-delay product and reorder buffer sizes are determined by the network skew.

Experiments confirm that ordering can be achieved at wire speed. We further showed that minimal ordering logic using a single reorder buffer suffices for many producer/consumer communication patterns.

## VII. Future Work

More research into the interplay of ordering and congestion management is needed. The impact congestion management techniques have on ordering performance needs to be characterized and collaborative techniques that minimize the ordering pressure congestion management adds need to be explored.


## Acknowledgment

This work was funded by and performed under a subcontract between The Regents of the University of California and NVIDIA. The subcontract was issued under Prime Contract No. DE-AC02-05CH11231 between the University and the United States Government, represented by the Department of Energy for the management and operation of the Lawrence Berkeley National Laboratory and the performance of research and related work.

## REFERENCES


[1] R. E. Kessler and J. L. Schwarzmeier, "CRAY T3D: A New Dimension for Cray Research," in *Compcon*. IEEE, 1993, pp. 176–182.

[2] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, J. Reinhard *et al.*, "Cray Cascade: A Scalable HPC System Based on a Dragonfly Network," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, p. 103.

[3] B. Bhattacharjee, S. Boag, C. Doshi, P. Dube, B. Herta, V. Ishakian, K. Jayaram, R. Khalaf, A. Krishna, Y. B. Li *et al.*, "IBM Deep Learning Service," *IBM Journal of Research and Development*, vol. 61, no. 4, pp. 10–1, 2017.

[4] "Top500 List - November 2017," [accessed 22-May-2018]. [Online]. Available: http://www.top500.org/list/2017/11/

[5] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and A. Ng, "Deep Learning with COTS HPC Systems," in *International Conference on Machine Learning (ICML*, 2013, pp. 1337–1345.

[6] "Microsoft Unveils Project Brainwave for Real-Time AI," August 2017, [accessed 22-March-2018]. [Online]. Available: https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2017, pp. 1–12.

[8] "NVIDIA DGX-2," [accessed 28-March-2018]. [Online]. Available: http://www.nvidia.com/en-us/dat-center/dgx-2/

[9] B. Klenk, L. Oden, and H. Fröning, "Analyzing Communication Models for Distributed Thread-Collaborative Processors in Terms of Energy and Time," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 318–327.

[10] N. Jiang, L. Dennison, and W. J. Dally, "Network Endpoint Congestion Control for Fine-Grained Communication," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2015, pp. 1–12.

[11] J. Mankin, "CSG280: Parallel Computing Memory Consistency Models: A Survey in Past and Present Research," 2007.

[12] 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 *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 86–96.

[13] M. García, E. Vallejo, R. Beivide, M. Odriozola, and M. Valero, "Efficient Routing Mechanisms for Dragonfly Networks," in *International Conference on Parallel Processing (ICPP)*. IEEE, 2013, pp. 582–592.

[14] M. Laor and L. Gendel, "The Effect of Packet Reordering in a Backbone Link on Application Throughput," *IEEE network*, vol. 16, no. 5, pp. 28–36, 2002.

[15] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural Guidelines for Multipath TCP Development," IETF, Tech. Rep., March 2011. [Online]. Available: www.rfc-editor.org/info/rfc6182

[16] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken *et al.*, "Blue Gene/L Torus Interconnection Network," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 265–276, 2005.

[17] G. Almási, C. Archer, J. G. Castanos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman *et al.*, "Design and Implementation of Message-Passing Services for the Blue Gene/L Supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 393–406, 2005.

[18] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *Symposium on High Performance Interconnects (HOTI)*. IEEE, 2010, pp. 83–87.

[19] K. Z. Ibrahim, P. H. Hargrove, C. Iancu, and K. Yelick, "An Evaluation of One-Sided and Two-Sided Communication Paradigms on Relaxed-Ordering Interconnect," in *International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2014, pp. 1115–1125.

[20] "Gen-Z Consortium," [accessed 9-February-2018]. [Online]. Available: http://genzconsortium.org

[21] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[22] "The Berkeley UPC Runtime Specification, Version 3.12," [accessed May-23-2018]. [Online]. Available: http://upc.lbl.gov/docs/system/upcr.ps

[23] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick, "Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap," in *International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2009, pp. 1–12.

[24] S. Potluri, D. Rossetti, D. Becker, D. Poole, M. G. Venkata, O. Hernandez, P. Shamis, M. G. Lopez, M. Baker, and W. Poole, "Exploring OpenSHMEM Model to Program GPU-based Extreme-Scale Systems," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2014, pp. 18–35.

[25] F. Z. Nardelli, P. Sewell, J. Sevcik, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave, "Relaxed Memory Models Must be Rigorous," in *Exploiting Concurrency Efficiently and Correctly Workshop*, 2009.

[26] S. Xiao and W.-c. Feng, "Inter-Block GPU Communication via Fast Barrier Synchronization," in *International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.


## APPENDIX

### A. Abstract


This artifact contains all components of the simulation environment and the scripts needed to generate the reported results. The simulator is based on Bksim, an event-driven architectural simulation library written in C++. Further provided are the scripts and programs needed to generate the graphs shown in the paper.


### B. Description

*1) Check-list (artifact meta information):*

- **Algorithm:** Full protocol and network simulation
- **Program:** Bksim code to run simulations, bash scripts (incl. Python code and R code) to generate graphs
- **Compilation:** GCC 7.10 or higher
- **Run-time environment:** Ubuntu 16.04
- **Hardware:** Any
- **Output:** Simulation log files, statistics, graph plots
- **Experiment workflow:** git clone projects, build environment, run scripts
- **Publicly available:** Yes

*2) How software can be obtained:* The simulator code is found on GitHub. It is provided in three parts: 1) the libbksim simulation kernel is located in repository NVIDIA/nvr-libbksim, 2) the accompanying library libnvx in NVIDIA/nvr-libnvx, and 3) the ordering protocols in NVIDIA/nvr-ord-prot-sc18.

*3) Hardware dependencies:* None.

*4) Software dependencies:*

- Autoconf 2.69 or later
- Automake 1.15 or later
- Bison 3.0.4
- Flex 2.6.0
- GCC 7.1 or later
- libbksim 2.1.4 or later
- libtool 2.4.6 or later
- libboost 1.58.0 or later
- libnvx 1.0 or later
- log4xcc 0.10.0 or later
- Python 3.5.1
- Rstudio 1.1.419

*5) Datasets:* None.

### C. Installation

Clone repositories:

```
$ git clone https://github.com/NVIDIA/nvr-libbksim.git
$ git clone https://github.com/NVIDIA/nvr-libnvx.git
$ git clone https://github.com/NVIDIA/nvr-ord-prot-sc18.git
```

Follow the instructions in the README file found in project nvr-ord-prot-sc18.

### D. Experiment workflow

Navigate to directory ~/nvr-ord-prot-sc18/src and run script ./sim_scripts/run_sim_all to run simulations and generate graphs.

~/nvr-ord-prot-sc18/src is the working directory of the simulator. A simulation is executed by running the executable bksim. The working directory contains the following subdirectories:

- *configs* contains Bksim configuration files to specify network topologies and other simulation parameters.
- *logs* contains trace files and statistics output files.
- *plots* contains the graphs.
- *python* contains programs to pre-process statistics data in preparation for plotting.
- *results* contains the files with the raw performance results.
- *rstudio* contains R code to generate the graphs.
- *sim_scripts* contains the bash scripts to run the simulations and generate graphs.

### E. Evaluation and expected result

The simulator generates trace files and periodic statistics reports. The trace level is specified in file log4cxx.conf found in the simulator's working directory. Shell scripts, Python programs and R code are used to extract results and generate plots.

### F. Experiment customization

Scripts and configuration files can be modified to change the network topology and customize traffic generation.