Exploiting Idle Resources in a High-Radix Switch for Supplemental Storage

Matthias A. Blumrich
NVIDIA Corporation
Westford, MA
mblumrich@nvidia.com

Nan Jiang
NVIDIA Corporation
Santa Clara, CA
tedj@nvidia.com

Larry R. Dennison
NVIDIA Corporation
Westford, MA
ldennison@nvidia.com

***Abstract*—A general-purpose switch for a high-performance network is usually designed with symmetric ports providing credit-based flow control and error recovery via link-level retransmission. Because port buffers must be sized for the longest links and modern asymmetric network topologies have a wide range of link lengths, we observe that there can be a significant amount of unused buffer memory, particularly in edge switches. We also observe that the tiled architecture used in many high-radix switches contains an abundance of internal bandwidth. We combine these observations to create a new switch architecture that allows ports to stash packets in unused buffers on other ports, accessible via excess internal bandwidth in the tiled switch. We explore this architecture through two use cases: end-to-end resilience and congestion mitigation. We find that stashing is highly effective and does not degrade network performance.**

***Index Terms*—High performance computing, Multiprocessor interconnection networks, Packet switching, High-radix switches, Buffer storage**

I. Introduction

In recent years, high-performance network systems have been dominated by low-diameter network topologies [1]–[3] constructed from high-radix switches [4]–[6]. A prominent characteristic of these topologies, compared to traditional, multi-dimensional tori [7]–[9], is their highly asymmetric link lengths. For example, in the popular dragonfly topology each switch has three types of connections: network endpoints, other switches in a group, and long optical links between groups. Even in a massive dragonfly, each group consists of a few hundred endpoints at most, and can be packaged in several adjacent cabinets. Therefore, links within groups do not need to be more than 5 meters in length, and links to endpoints can be much shorter. However, between groups—with system footprints up to 600 square meters [10] and growing [11]—the maximum link length (corner-to-corner) is 60 to 100 meters.

Asymmetric network links lead to asymmetric resource requirements for switch ports. In particular, the amount of buffering required by each port to implement link-level retransmission and credit-based flow control is directly proportional to the length of the connected link. While resources in a switch can be implemented asymmetrically to match the requirements of a specific topology [12], the ability to use the switch in other configurations is then reduced.

On the other hand, a more general-purpose network switch with symmetric ports can be utilized for a variety of topologies and system sizes. To do so, it must be designed with sufficient resources to satisfy the demands of various configurations. A good example is Intel's Omni-Path switch [4], which has 48 ports supporting links up to 100 meters [13] at 100 Gbps. Table I summarizes the connectivity of ports and the corresponding buffer requirements when such a switch is used to implement a large-scale dragonfly network. Weighting the last column by the third column reveals that approximately 72% of all the port buffering on each switch is not required for normal link-level retransmission and flow control due to the mismatch between the physical lengths of links and the buffering available in the symmetric ports. Similar analyses can be conducted for other low-diameter topologies such as the leaf switches in a multi-level fat-tree.

TABLE I
Asymmetry of links in a canonical dragonfly network switch.

Link Type	Length	% of Ports	Port Buffers Underutilized
Endpoint	$<$ 1m	25	99%
Intra-group	$<$ 5m	50	95%
Inter-group	$<$ 100m	25	0%

Observing the large amount of buffer resource underutilization when deploying general-purpose switches with symmetric ports in asymmetric networks, we have developed a novel switch architecture that allows the unused buffers from all ports to be utilized collectively as a common storage resource, accessible from any port on the switch. Our architecture is evolved from a typical, state-of-the-art tiled architecture used in modern high-radix switches. We observe that a tiled switch has an overprovisioning of internal bandwidth that can be used to access the unused buffers without impacting performance. We call our new method of accessing idle buffers using otherwise idle internal bandwidth "stashing".

In addition to developing the new switch architecture, we show that the additional storage provided by stashing can be used to implement a variety of network features and enhancements. One of the principle uses we identified is end-to-end retransmission for error recovery, a common feature of most protocols for traditional, large-scale networks including Transmission Control Protocol (TCP) [14] and InfiniBand's reliable connections [15]. However, evolving network systems with directly-attached accelerators may not devote endpoint

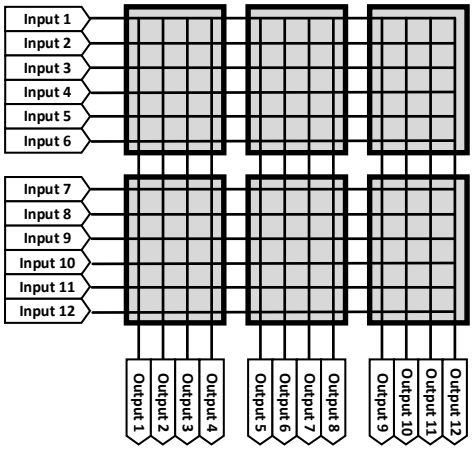


Fig. 1. A tiled switch with configuration P=12 R=2 C=3 I=6 O=4.

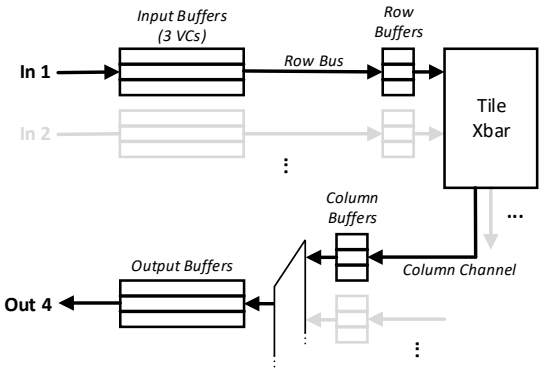


Fig. 2. Example datapath of a tiled switch with three virtual channels, from an input port to an output port.

resources to providing such capabilities because of the economics driving the component markets. A good example is NVIDIA's recently announced NVSwitch fabric [16] which interconnects commodity GPUs designed for multiple markets. We show how end-to-end retransmission can be implemented in the network switches instead, using stashing storage.

To further demonstrate the diverse applicability of our architecture, we also studied the use of stashing to enhance the performance of explicit congestion notification protocol (ECN) for congestion management. We show that the additional storage can be used to temporarily prevent head-of-line (HoL) blocking effects in the network while the congestion protocol activates. Through these two use cases, we demonstrate that the new architecture can fully utilize the otherwise idle buffers in the ports of a general-purpose switch to provide new services without affecting normal network performance.

II. Baseline Tiled Switch

Tiling has proven to be a scalable method for building high-radix, high-performance network switches [5], [17], [18]. We describe a typical tiled switch in this section and use it as the baseline in our simulations as well as the basis for our stashing switch architecture, described in Section III.

Figure 1 shows the basic architecture of a tiled switch with radix P. It consists of a two-dimensional array of identical tiles, with R rows and C columns, forming a large crossbar. Each tile (represented by a grey box) has I inputs and O outputs, and contains a smaller $I \times O$ *tile crossbar* that functions as a portion of the switch crossbar. In general, a valid tile arrangement must satisfy:

$$P = R \times I \tag{1a}$$

$$P = C \times O \tag{1b}$$

Each row of tiles is fed by I switch input ports, and each switch input is connected to all C tiles in a row via a multi-drop bus. For example, there are 12 row buses in Figure 1 and each connects to three tiles. The outputs from the R tiles in each column are point-to-point channels, so they must be merged with an R-to-1 multiplexer feeding each switch output. In Figure 1, each switch output is fed by two tile outputs—one from each row—requiring a 2-to-1 multiplexer.

It should be apparent from Figure 1 that various tile sizes are possible depending on engineering trade-offs such as clock frequency, switch radix, the availability of wiring tracks, and tile complexity. Previously published designs include 8×8 tiles for a 64-port switch [18] and 3×4 tiles for 36 ports [6]. A key characteristic general to all tiled switches is an overprovisioning of internal bandwidth.

Across the entire switch, the total number of column channels from the tile crossbars to the output multiplexers is $R \times C \times O$. Substituting equation (1b), we see that the internal column bandwidth of the tiled switch is R times higher than the switch radix, P. It is this excess internal bandwidth (together with the multi-drop behavior of each row bus) that allows us to stash packets without impacting switch performance.

Figure 2 is a simplified diagram of the datapath from one switch input to one switch output, assuming a multi-drop row bus. The internal buffering is critically important because it allows the internal routing of packets to be performed in two independent steps: input to tile, and tile to output.

Packets received by the switch are stored in the input buffers, and the Virtual Channels (VCs) compete for access to the row bus (represented by a horizontal line in Figure 1). The winner advances to its corresponding VC row buffer at a tile crossbar input. All the row buffers compete for access to the tile crossbar and the column channels (represented by the vertical lines in Figure 1). The winner of each tile crossbar output advances to its corresponding VC column buffer at an output port, and all the column buffers compete for the switch output buffers. Finally, the VCs in the output buffer compete for the link.

The large input and output buffers in Figure 2 are used to provide flow control and reliable transmission between switches. On the input side, sufficient buffering is required to implement lossless, credit-based flow control. On the output side, packets are stored until a positive link-level acknowledgment is returned from the receiving switch indicating that the transmission was successful. Both buffers must be sized for roughly one link round-trip time's (RTT) worth of data; that

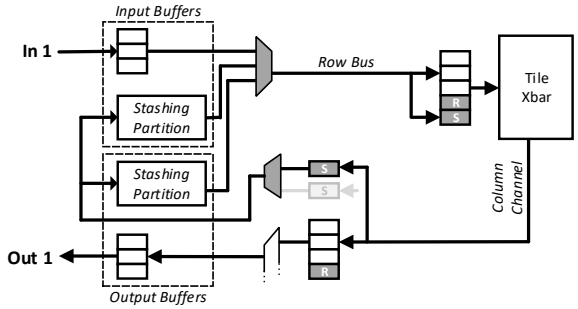


Fig. 3. Datapath of a tiled switch enhanced for stashing.

is, the product of the round-trip latency and the bandwidth. It is these large buffers we target for stashing, as described next.

III. Stashing Switch

The stashing switch introduces some modifications to the baseline tiled switch, as described in this section. The principle design challenges we address are accessing, isolating, and managing the idle storage.

A. Storage Access

The first design challenge we address is access to the stashing storage. We enhance the baseline datapath with the shaded components shown in Figure 3 (which can be compared to Figure 2). Here we show a single port, consisting of the input and output datapaths, to illustrate how the available port memory is combined for stashing. Each input and output port buffer is virtually partitioned into a small portion for normal use and a large stashing partition. The two stashing partitions (not shaded because they consist of existing memory) are then managed as a single stashing storage pool for the port.

We add multiplexers to access the separate read and write ports of the stashing partitions, and we add two extra VCs to the switch datapath: one for storage (S) and one for retrieval (R). These VCs are internal to the switch and not externally visible.

Packets to be stashed are written from a switch input buffer to the VC labeled "S", which arbitrates for the crossbar with the same priority as the other VCs. The additional path arrow pointing to the S buffer is meant to indicate that packets written to the storage VC can also be the duplicates of packets written to the regular VCs; the multi-drop nature of the row bus allows an input to broadcast a packet so that it can be received by multiple tiles in a row and/or multiple VCs at a tile simultaneously. At the output port, packets in the storage VC from all rows in a column arbitrate through a separate multiplexer and get written into one of the two stashing buffer partitions.

Packets retrieved from the stashing buffers arbitrate for the row bus alongside packets from the normal input buffer, but use the VC labeled "R". This VC has equal priority with the other VCs in the crossbar. Since the retrieved packets and normal packets from the same port share the row bus into the crossbar, some bandwidth contention may occur. There are several ways of addressing the additional bandwidth demand

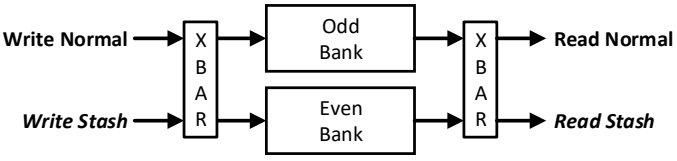


Fig. 4. A buffer supporting four ports, organized as two interleaved banks.

on the row bus, such as adding a duplicate row bus per input dedicated to retrieved packets. In our simulations we chose to use a modest 30% internal speedup of the tiled switch to overcome the additional row bus bandwidth demand, noting that core overclocking is a common technique for enhancing switch performance at the cost of some additional power consumption and engineering effort. On the output multiplexer, the retrieved packets are stored in their own "R" VC buffer alongside the existing VC buffers headed to the switch output. After the output multiplexer, retrieved packets are returned to their original output VC.

Various algorithms can be used to decide which port to send a stashed packet. However, unlike normal packets which have definitive output ports, stashing packets can go to any port with stashing buffer space. Therefore, we use a simple join-shortest-queue method to adaptively select a path that will likely send packets to a port with the least used stashing buffer. When a packet to be stashed is at the head of the input port, we send it to the tile column with the most storage VC credits available. Similarly, when the stashed packet is at the head of the tile crossbar input, we send the packet to the column channel with the most storage VC credits available. Ports with no stashing buffers available (e.g., the global ports of a dragonfly) are known a priori, so the stashing paths to those ports are statically omitted from the selection process.

Flow control for the stashing VCs is implemented via credits, similar to all other VCs. Depletion of credits on the stashing VCs results in back-pressure that can affect the performance of the switch. For example, when stashing for end-to-end retransmission, a packet cannot be sent unless there are stashing buffers available. Therefore, transfer from the normal input buffer to a tile depends on the availability of the normal VC *and* the availability of the storage (S) VC. If the stashing paths are back-pressured, then the switch input stalls, even if the normal path is available.

B. Storage Isolation

The second design challenge, storage isolation, involves partitioning buffer memory so that is can be used for both the normal functionality and for stashing. A dual-ported memory with one read and one write port is sufficient for normal use, but stashing adds a second set of read and write ports, along with the potential for all four ports to be active simultaneously. Obvious design options are 4-ported memory or dual-ported memory operating at twice the normal speed.

A better solution is illustrated in Figure 4, where the memory is divided into two banks shared by the read and write ports. Each bank stores flits, one per cycle, with all even-numbered offsets in one bank and odd in the other. Therefore, a

multi-flit packet write or read operation is interleaved between the two banks. The multiple interleaved bank method allows the memory to be divided into two virtual partitions of any size at a two-flit granularity.

There are several ways to deal with bank collisions between the two ports [19]. Write sequences can simply avoid one another and remember which bank they started on (1 bit per packet), or they can be presented two flits at a time and written in the order of availability. Read sequences are just started in a non-conflicting order.

C. Storage Management

The last design challenge is managing the storage. One important aspect is the overall mechanism for distributing data around the switch and keeping track of it. Because this is heavily dependent on the use cases for stashing, we defer the discussion to Section IV. Here we focus on the management of each individual stashing buffer, which requires three primary operations: storing, retrieving, and deleting.

Port buffers should be designed to provide management functionality that partitions along with the buffer space. A general-purpose switch faces the challenge of providing many VCs for deadlock avoidance and multiple traffic classes. An elegant solution is the Dynamically Allocated Multi-Queue (DAMQ) [20], which allocates buffer pages to VCs as needed and manages them as a heap. Each buffer is divided into small, fixed-size pages which can be dynamically allocated to packets and tracked for later retrieval and/or deletion. As stashing has the same requirement, the heap management functionality could simply be partitioned into a DAMQ heap and a stashing heap.

Alternatively, the heap management functionality could be provided separately for the stashing partition. We sized such a scheme, making some reasonable assumptions about the flit size, page size and maximum packet count, and determined that storage overhead for the linked lists that manage the heap could be less than 10% of the buffer size.

IV. Use Cases

Although we envision a wide variety of network features that could exploit the additional storage resources provided by the stashing architecture, we limit this study to two distinct use cases in order to demonstrate how stashing can be integrated into network designs and the effect it has on performance.

A. Use Case: End-to-end Reliability

A practical use case for additional storage is end-to-end retransmission, which is a feature of the most common network protocols including Transmission Control Protocol (TCP) [14] and InfiniBand's reliable connections [15]. While most high-performance switches offer link-level retransmission to handle high error rates between adjacent devices [4], [12], comprehensive coverage for errors in the switch logic and buffering is more difficult to achieve and often lacking. End-to-end retransmission offers a straightforward mechanism to recover from transient failures of all kinds and drives up network reliability dramatically [21].

End-to-end retransmission for error recovery is traditionally implemented in endpoint software or by a NIC. However, in newer accelerator-centric systems consisting of high-performance, commodity endpoints directly connected to one another, there may be no economically feasible way to do so. Using the stashing buffers, we propose a proof of concept design of end-to-end reliability for such systems, where the functionality of end-to-end retransmission is moved into the first-hop switch. This allows new systems built with a variety of endpoint devices to enjoy a major fraction of the reliability benefit.

As discussed previously, our design assumes a dragonfly network topology, though similar designs are feasible for other high-radix, asymmetric topologies such as multi-level fat-trees. Each switch in a dragonfly consists of two types of ports: *end ports* connecting to endpoints, and *network ports* connecting to other switches[1]. All injected packets arriving at end ports must be stored for retransmission. We use the stashing buffers in our switch to hold a copy of these packets until an acknowledgment (ACK) is received. Packets arriving at network ports do not require retransmission and are treated in the same manner as in the baseline tiled switch.

When a packet reaches the head of an end port input buffer, it will arbitrate for both its normal path through the crossbar as well as a path to a stashing buffer on the storage VC. Selecting a path to a stashing buffer uses the join-the-shortest-queue mechanism described in Section III-A. A packet can only make forward progress if both paths are unblocked. If the storage VC is blocked due to stashing buffers around the chip being exhausted, packets are stalled at the input until additional stashing buffers are freed by returning ACKs. Because stashing for end-to-end retransmission is done on the first-hop switch, there is no correctness or deadlock concern; the network simply slows down its packet injection rate.

When the normal path and the stashing path are available, the input port sends the packet to the row bus. Due to the multi-drop nature of the row bus, the packet can be sent to the normal VC buffer and the stashing VC buffer simultaneously, regardless of which columns each path happens to be in. Therefore, we create a copy of the packet for stashing without consuming additional input port or row bus bandwidth.

The packet copy at a tile crossbar selects a switch output port based on the join-the-shortest-queue method and competes with normal traffic for the column channel. Once it reaches the stashing buffer, a location message containing the buffer index is sent back to the end port where the copy originated. The location message is stored in the originating end port's existing packet management data structure along with all the other relevant metadata used to track the original outstanding packet.

[1]Network ports can be further characterized as local or global, but this distinction is not important here.

When the original packet reaches the destination endpoint, an ACK is returned. We assume these ACKs are hardware generated and return immediately, independent of any higher-level network protocols. As a result, they do not introduce additional protocol deadlock conditions or delays.

When the originating end port receives the ACK, it retrieves the stashed copy location from the management data structure. If the ACK is positive, it sends a delete message to the copy location, freeing up the stashing buffer. If the ACK is negative, indicating an error, it sends a retransmit message to the copy location, causing a retrieval and retransmission. We did not simulate the retrieval or retransmission because this study is primarily concerned with the impact of stashing on performance and we expect retransmissions to be rare.

An ACK could return to the originating end port before the location message from the stashing port. In that case, the normal packet tracking data structure must remain in place until the location message arrives. If the ACK is positive, then the normal packet completion process can proceed, and the eventual arrival of the location message will be followed immediately by a deletion command. If the ACK is negative, then all retransmit processing simply waits until the location message arrives.

As described, this design requires side-band communication between the end ports and the stashing buffers around the switch to exchange the various bookkeeping messages. These messages are small, only containing metadata including the packet tracking index, port numbers, and the stashing buffer index. For our proof-of-concept study, we modeled a simple dedicated network to handle these side-band communications.

B. Use Case: Improving Congestion Control

Another potential application of the stashing buffer architecture is improving existing congestion control protocols by temporarily reducing HoL blocking in the network. Congestion control mechanisms deployed in HPC networks today, such as ECN, have been shown to work well for long-duration congestion [22]. In general, ECN-type protocols work by detecting signs of congestion, and then throttling traffic sources contributing to the congestion via feedback messages. However, due to the long network latency in a large-scale network (on the order of microseconds), congestion feedback can incur a sizable delay. While the feedback is in-flight, a large amount of congestion-forming traffic can be sent before the traffic sources react. Furthermore, it takes time for the source throttling mechanism to converge to an optimal solution, resulting in further delays in the congestion response. As a result, this type of *reactive* congestion control exhibits transient performance degradation and tree saturation during the onset of congestion.

Using stashing buffers, we can improve these ECN-type protocols by temporarily absorbing the congestion-forming packets while the protocol is in the process of responding. This temporary absorption shields other traffic in the network from transient effects and provides extra time for the ECN feedback and source throttling mechanism to fully respond. As a result, the stashing architecture can offer significant improvement in the latency performance of a network during congestion.

Using stashing buffers with ECN congestion control does not depend on the exact algorithm and can be adapted across variants found in production [22] and in the literature [23], [24]. The baseline ECN protocol we selected for this study is most similar to the one found in TCP/IP networks [25], which use transmission windows to control the injection rate of traffic sources. Unlike TCP/IP, the network we simulate is lossless and only relies on congestion notifications to resize transmission windows.

In our baseline ECN algorithm, each endpoint in the network maintains a separate transmission window for every other endpoint and can only transmit to an endpoint if space is available in its dedicated transmission window. When a source injects a packet into the network, the packet's size (in flits) is added to the destination's transmission window. When the source later receives a positive ACK, the packet's size is deducted. The size of each transmission window is initially set to a maximum value of 4096 flits, allowing an endpoint to stream traffic to any destination at full bandwidth.

Congestion detection is done by monitoring the occupancy of switch input buffers. When a buffer's occupancy exceeds 50% of its capacity, the input port enters a *congested state* and begins to set the ECN bit of any packet passing through it towards a destination. At a destination, the ECN bit is copied from the packet to the ACK and returned to the source. Every ACK received with the ECN bit set causes the source to reduce the size of the transmission window for the destination to 80% of the current size.

To recover from congestion, each transmission window has a timer which causes it to increase its size by one flit every 30 cycles until reaching the initial window size of 4096 flits. ECN parameters described here were carefully adjusted to fit our particular network configuration, described in more detail in Section V.

When applying the stashing buffers to the ECN protocol, the basic congestion detection and source throttling mechanisms are left unchanged. When an input port is in the congested state, it opportunistically sends packets destined for end ports to stashing buffers in addition to the normal packet marking behavior. Stashing only occurs when four conditions are satisfied: 1) the packet is at the head-of-line of a congested input, 2) the packet is destined to an end port, 3) the packet cannot advance on its normal VC through the crossbar (i.e., blocked due to congestion and lack of credits), and 4) the packet can advance on the stashing storage VC. Selecting a path to a stashing buffer uses the join-shortest-queue method described in Section III-A.

When a congested packet reaches a stashing buffer, it is stored and accessed from the buffer in FIFO order. Packets in the stashing buffer are immediately allowed to contend for their intended outputs through the crossbar using the stashing retrieval VC. Since packets traveling to and from the stashing buffers use separate storage/retrieval VCs, any congestion created by them does not result in HoL blocking on normal

packets in the crossbar. Additionally, the number of congested packets stored in a port's stashing buffer is not included in the normal ECN congestion calculation for that port.

By stashing blocked packets from a congested input port, we are alleviating HoL blocking effects on that port. Any uncongested traffic that shares an input with a congested traffic flow is now able to make more forward progress compared to the baseline. Since there are only a limited number of stashing buffers on a single switch, congested input ports will eventually be unable to stash away packets and HoL blocking will resume. However, stashing congested packets is meant to be a temporary measure that provides more time for the ECN mechanisms to throttle the congestion at its source. Once ECN responds, the affected input ports will leave the congested state and packet stashing stops.

C. Other Use Cases

Speculative Reservation Protocol [26] and the related Last-Hop Reservation Protocol [27] are mechanisms for detecting congestion and throttling problematic senders. The common feature of these protocols is that messages causing congestion are dropped and then scheduled for retransmission at a reduced pace. Stashing allows the throttling and retransmission mechanisms to be implemented in the first-hop switches, including storage of the speculative packets.

Support for packet order enforcement is important for inherently unordered networks such as the dragonfly because programming models sometimes require strict order. For example, completion of a large buffer transfer, such as a halo exchange, is usually indicated by a flag which must strictly follow all of the data. In lieu of one-at-a-time sends, hardware can dramatically accelerate ordered transfers by providing reordering buffers at the destinations. However, such buffers are a limited resource and may result in dropped packets when they are exhausted. End-to-end retransmission provides recovery, dramatically simplifying the implementation and allowing for eager solutions.

V. Methods

We evaluated the new switch architecture using a modified version of the network simulator Booksim [28], [29] and the two use cases described in the previous section. The simulator is cycle-accurate and models operations at a flit granularity. We fully implemented the internal pipeline of the baseline tiled switch, described in Section II, and the new stashing architecture described in Section III.

The switch models are symmetric and parameterized for different numbers of ports and internal tiles. For the 20-port switches used in the experiments, the tiling configuration is $P = 20$, $R = C = 4$, and $I = O = 5$. Following the description in Section II, the row bus from each input port is connected to four tile crossbar inputs in the same row. Each of the tile crossbar outputs in the same column is connected through a 4-to-1 multiplexer to a switch output port.

Each 5x5 tile crossbar uses a virtual-output-queued (VOQ) architecture, with four packets worth of storage per VC. Tile crossbar arbitration uses a separable output-first allocator [30] where all VCs have equal priority, including the additional storage (S) and retrieval (R) VCs introduced by the stashing architecture (Figure 3). Each tile output multiplexer has four packets worth of storage per VC and uses robin-robin arbitration. In the experiments, we operate the switch with a $1.3\times$ higher clock speed compared to the network channels. We assume the network channels are operating at 1 GHz with a flit size of 10 bytes (10 GB/s bandwidth). The maximum packet size is 24 flits, and all data packets are acknowledged by the destination using single-flit ACKs.

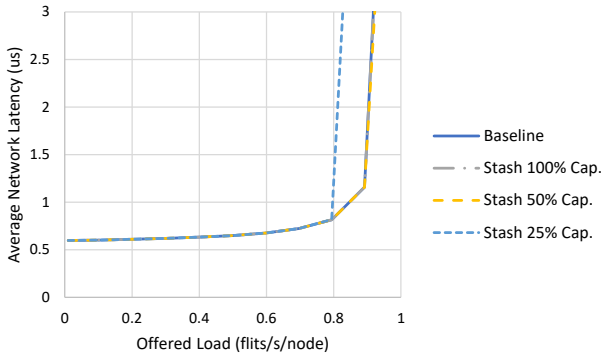
The network used for both case studies is a 3080-node canonical dragonfly topology with full bisection bandwidth. Of the 20 switch ports, five are connected to endpoints, ten to local switches in the same group, and five to switches in other groups. Since the ports are symmetric, they are simply assigned randomly. Each dragonfly group consists of 11 switches in a fully-connected topology, and the whole network contains a total of 56 groups. The one-way latency of the endpoint, local, and global channels is set at 5, 40, and 500 ns, respectively. The global channel latency is designed to simulate long, optical links in a large-scale dragonfly that spans multiple cabinets. For the routing algorithm, we implemented PAR6/2 progressive adaptive routing using six VCs to prevent routing deadlock [31].

With the channel bandwidth of 10 GB/s and the maximum round-trip latency of $1\mu s$ on the global channels, each port's input and output buffers contain 10 KB of storage, for a total of 20 KB per port, or 400 KB for the entire 20-port switch. The input and output storage is shared among the six network VCs using a DAMQ structure.

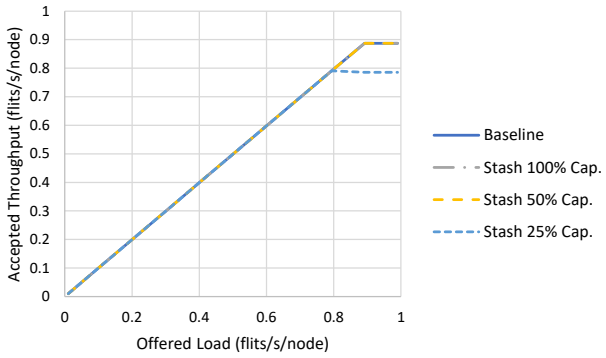
In the stashing switch, a portion of the input and output storage is partitioned for stashing buffers while leaving sufficient storage to completely satisfy the normal demands of all ports. We conservatively partitioned the buffers with 7/8 of the storage configured for stashing on the five endpoint ports, 3/4 on the ten local ports, and none on the five global ports. Therefore, each switch contains a total of 237.5 KB of stashing storage capacity. We conduct some sensitivity tests where we artificially restrict the amount of stashing storage at each port to 50% and 25% of that total capacity.

Network endpoints generate traffic at the granularity of messages, with sizes specified for each experiment. Endpoints use a mechanism similar to InfiniBand queue-pairs to transmit messages: source endpoints have a separate send queue for each destination and destination endpoints have a separate receive queue for each source. Multiple active send queues at a source arbitrate for the injection port on a per-packet, round-robin basis.

For application trace simulation, we conducted experiments using the SST/Macro full system simulator, version 6.1 [32]. SST/Macro is a coarse-grained simulator that is capable of modeling entire HPC systems from the application layer down to the network hardware. The simulator implements an MPI library which allows it to translate complex MPI operations



(a) Latency vs. Throughput



(b) Offered Load vs. Accepted Throughput

Fig. 5. Performance impact of stashing storage for end-to-end reliability under a uniform-random traffic pattern.

into individual network messages suitable for lower-level network simulation.

We modified SST/Macro to use Booksim as the network layer, allowing us to test our architectural designs directly with system-level experiments. Endpoints of the trace simulations use an existing SST/Macro configuration similar to the NERSC Edison system [33], with 24 cores and 50 GB/s of memory bandwidth. Each endpoint has a single 10 GB/s injection channel connected to the network, matching the switch channel bandwidth. Detailed instructions on downloading and running the simulation infrastructure appear in the reproducibility appendix.

VI. Results

A. End-to-end Reliability

The end-to-end reliability mechanism stresses two aspects of the stashing buffer architecture: storage bandwidth and capacity. First, all end ports can stream packets at the maximum injection rate simultaneously and each of these packets creates a copy that needs to be stored in a stashing buffer. Although the copies travel on a separate VC, they still double the internal crossbar bandwidth pressure from the end ports and compete with the normal traffic for tile crossbar and column channel bandwidth. Second, if the stashing buffer capacity on a switch is full or cannot be accessed due to contention, the storage VC in the crossbar will back up, which can stop traffic from leaving end ports and degrade the network injection rate.

Since we expect network errors that trigger retransmission to be rare, our tests focus on how the addition of stashing traffic affects the error-free performance of the network. As the experimental control, we simulated a network constructed using the baseline tiled switch without stashing or retransmission. Endpoints in the baseline can have an arbitrary number of outstanding packets and should offer the best performance under error-free conditions. We compare the baseline to networks constructed with three versions of stashing switches: one with all the available buffer capacity (100% Cap), one restricted to half of the available capacity (50% Cap), and one restricted to one quarter of the available capacity (25% Cap). The restricted versions reveal sensitivity to total buffer capacity since performance is negatively affected whenever all the stashing buffers are exhausted.

We begin by evaluating a synthetic, uniform-random traffic pattern with single-packet messages. Figure 5a shows the latency versus throughput curves for the various networks, derived by averaging over all endpoints. The stashing networks with 100% and 50% capacity have nearly identical performance to the baseline even at very high network injection rates. This shows that the additional stashing storage bandwidth inside the switches has no appreciable impact on performance. Additionally, there is more than enough stashing capacity to cover the round-trip latencies across all network loads for the dragonfly we simulated. Only by severely restricting the capacity to 25% do we observe an early saturation for the network.

Figure 5b shows the offered versus accepted load of the networks, again averaged over all endpoints. When a network is below saturation, it is able to deliver all of the offered traffic and the graph appears linear. Once it saturates, its accepted load plateaus, regardless of further load increase. Similar to Figure 5a, the stashing networks with full and half capacity show nearly identical performance to the baseline, reaching 90% saturation throughput. Part of the reason this is not higher is because of bandwidth consumed by ACKs.

The network with 25% stashing capacity shows a lower saturation throughput of approximately 78%. The expected saturation throughput of this network can be calculated from the network latency and stashing capacity, which is approximately 60 KB per switch. This is shared among the five endpoints, resulting in an average of 12 KB worth of packets in flight per endpoint. Figure 5a shows that the average network latency just before saturation is approximately 0.8 us (1.6 us round-trip). Applying Little's Law, we calculate a theoretical injection rate sustainable by 12 KB of stashing storage as 75%, closely resembling the simulation result.

The uniform-random traffic experiments show that under benign network conditions there is more than enough stashing capacity to sustain a high volume of traffic. However, the communication patterns of real applications are typically not random, with traffic bursts and congestion as applications progress through communication phases.

We tested the end-to-end retransmission buffers under more realistic conditions using MPI traces of HPC applications

TABLE II
DESIGNFORWARD APPLICATION TRACES

Application Name	Description	Size (ranks)
BIGFFT	3D FFT with 2D domain decomposition pattern, medium problem size	1024
AMG	Algebraic multigrid solver for unstructured mesh physics packages	1728
MultiGrid	Geometric multigrid V-Cycle from production elliptic solver (BoxLib)	1000
Fill Boundary	Halo update from production PDE solver code (BoxLib)	1000
AMR	Full adaptive mesh refinement V-Cycle from production cosmology code (BoxLib/Castro)	1728
MiniFE	Finite element solver mini-application	1152

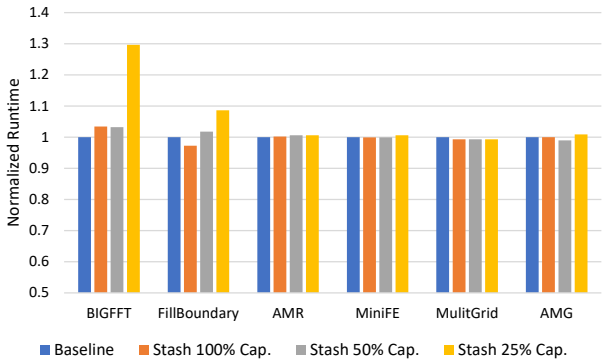
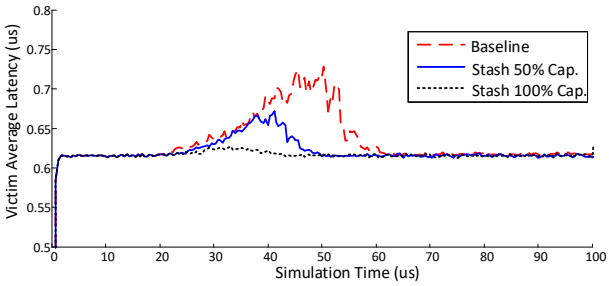


Fig. 6. Execution time of MPI application traces normalized to the baseline with no stashing and retransmission buffers.

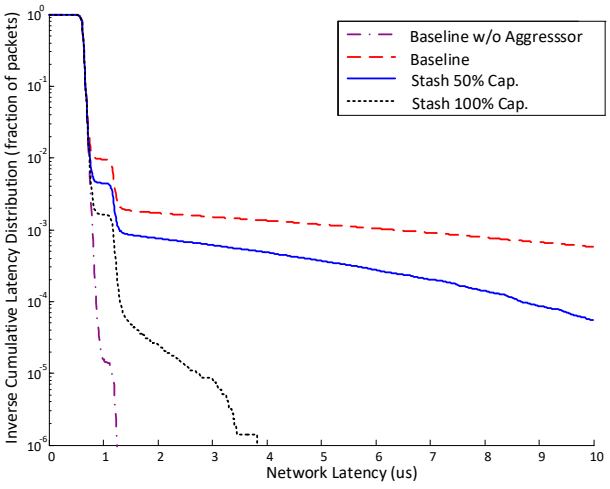
from the Department of Energy's DesignForward project [34]. Details of the traces are listed in Table II. Following the recommendation provided with the traces, we simulated with one rank per *endpoint* instead of one rank per *core*. Unless otherwise stated, application ranks are mapped to endpoints in the system *contiguously* without gaps. As a result, most network switches are either fully utilized, with five active endpoints, or idle, with no active endpoints. Furthermore, we did not model computation time in order to focus on the communication aspects of the applications.

Figure 6 shows the normalized execution time of the traces relative to the baseline network. In four of the traces (AMR, MiniFE, MultiGrid, AMG), all networks with stashing show nearly identical performance to the baseline, including the network with only 25% of available capacity. This can be explained by several factors. First, these traces have a low average network load, and the impact of restricted capacity is only dominant at high average loads, as shown in Figure 5. Second, although an endpoint can experience a high instantaneous injection rate (e.g., when sending a large message), it can monopolize all of the stashing capacity on the switch as long as other endpoints are not all experiencing high load at the same time. In other words, the network performance of bursty applications is more likely to be governed by the available stashing storage *per switch*, rather than the average storage per endpoint.

For the two traces that cause higher average network load than the others—BIGFFT and FillBoundary—we see more performance degradation caused by limited stashing capacity. With these bandwidth-bound applications, it is more likely that multiple endpoints sharing a switch will simultaneously experience high injection rates. As a result, the average stashing storage per endpoint has a more dominant effect, and the network with only 25% storage capacity shows significantly worse performance. Even so, for the two networks with larger stashing capacity, the increase in the execution time is at most 2% compared to the baseline.



(a) Victim Average Latency over Time



(b) Victim Traffic Latency Distribution

Fig. 7. Network transient response to the onset of congestion.

Interestingly, the data also shows that for some traces, networks with stashing can outperform the baseline. This anomaly is caused by the interaction between stashing and network congestion. Stashing capacity limits the number of outstanding packets that endpoints can inject before receiving acknowledgments, making them self-pacing. If the traffic is prone to congestion, self-paced endpoints can reduce the severity by capping the injection pressure on the network, leading to better overall application throughput in some situations.

B. Improving Congestion Control

Stashing congested packets stresses several aspects of the architecture, including storage bandwidth, capacity, and retrieval

bandwidth. To study the impact of a congested aggressor on a victim sharing the network, we used two sets of synthetic traffic patterns.

In the first experiment, the victim runs a uniform random traffic pattern on 3020 endpoints with a 40% injection rate. The aggressor has 48 source endpoints sending traffic to 12 destinations at maximum rate, creating a dozen 4:1 oversubscribed hotspots scattered across the network. All traffic uses single-packet messages. We tested an ECN-enabled network using three switch models: a baseline tiled switch, a stashing switch with full storage capacity (100% Cap), and a stashing switch with half of the total capacity (50% Cap). A network without ECN suffers severe performance degradation in our scenario and is not included in the results.

Figure 7a shows the average network latency of the victim packets over time. The aggressor is activated at $20us$ causing a sudden increase in traffic that slows the victim down as congestion mounts. This is followed by a transition phase as ECN engages and the congestion continues to build. At about $60us$, ECN has successfully throttled the aggressor traffic sources and the behavior of the victim returns to normal.

In both networks using stashing buffers, the victim suffers less after the activation of the aggressor. The packets that would have created the transient congestion in the network are temporarily stashed away. This reduces the HoL blocking experienced by the victim and allows its latency to remain close to normal. As one would expect, the network with maximum stashing capacity outperforms the network with only 50% capacity; more stashing buffers allow a network to absorb more of the transient congestion and keep it from affecting the victim.

While the dozen hotspots across the network appear to create a visible but small impact on the *average* latency of the baseline network, it is highly skewed. Victim packets that share paths with the aggressor can experience significantly higher latencies during the transient. We demonstrate this using a cumulative latency distribution for the same experiments, shown in Figure 7b. In addition to the data from the three previous networks, we added the baseline network without the aggressor present as a reference.

Since the victim traffic has a benign communication pattern, the simulation without the aggressor shows a very tightly bound latency distribution. With the aggressor present, the victim traffic in the baseline network shows a very long tail, indicating that some packets experience very high latencies. This is because they are blocked behind the temporary congestion created by the aggressor. As stashing capacity is increased, the tail latencies drop because victim packets are far less likely to be blocked. At full capacity, the maximum latency is only about $3\times$ the best case (baseline without an aggressor). The effect of this on a victim application can be dramatic because a blocked message might be part of a collective operation.

To illustrate the behavior of the stashing buffers during the congestion event, we focus on a single switch connected to one of the aggressor hotspots, in Figure 8. The curves show the offered load of the aggressor traffic and the utilization of

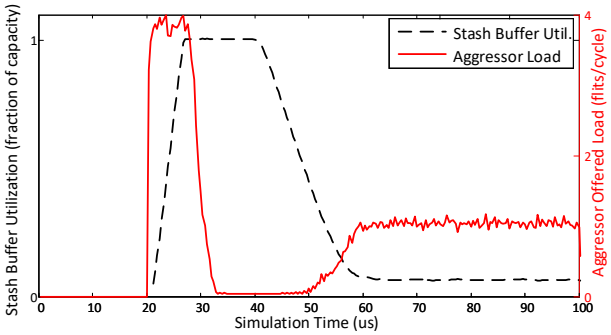


Fig. 8. Congestion control buffer usage during congestion event.

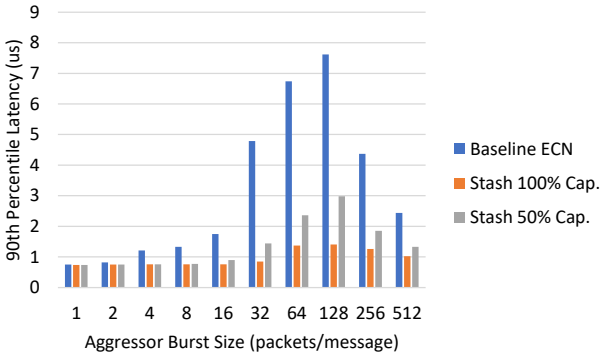


Fig. 9. Victim traffic tail latency when sharing the network with an aggressor traffic of various burstiness.

the stashing buffers. When the aggressor starts at $20us$, with an average bandwidth of 4 flits/cycle, the offered load shoots up and the stashing buffer utilization quickly follows.

About $12us$ after the onset of the aggressor, the offered load decreases sharply as ECN feedback begins to throttle back the traffic sources. The utilization of the stashing buffers remains high as the network continues to stash and relieve the pressure during the transient. Eventually, at about $60us$, the ECN algorithm converges to a steady state where the aggressor's average offered load is decreased to 1 flit/cycle and the hotspot is alleviated. Consequently, the stashing buffer utilization falls to near zero.

To test the congestion mitigation of stashing in a more generalized environment where one application can be subject to transient hotspots caused by another "bandwidth hog", we used a victim traffic pattern on half of the 3080 endpoints and an aggressor on the other half. The victim traffic pattern was again uniform-random with a 40% injection rate and single-packet messages. The aggressor also used a uniform-random traffic pattern, but ran at the maximum injection rate.

We varied the message size of the aggressor from single-packet to 512 packets per message. Larger messages simulate a bursty traffic environment because all packets belonging to the same message are injected consecutively and to the same destination. When multiple large messages from different sources converge to the same destination, transient congestion can arise even with the uniform-random communication pattern.

The experiment shows that while the accepted throughput of the victim traffic remained at 40% for all networks across

all burst sizes, the *latency* experienced by the victim is quite different. Figure 9 shows the 90th percentile latency of the victim's packets in the presence of the aggressor. That is, each data point is a lower bound on the 10% of packets with the highest latency.

Across all burst sizes, the stashing switches always outperform the baseline network. As the burstiness of the network and the resulting congestion increases, the stashing switches absorb the impact of the transient effectively while the baseline suffers. As a result, the victim traffic is far less likely to experience transient HoL blocking.

Because of the extended feedback delay of ECN, it performs poorly for short-term congestion and performs best for extended, long-term congestion. Short-term congestion disappears before ECN can react and long-term congestion benefits from ECN's steady-state behavior. Therefore, we see a peak in the curves of Figure 9, showing how increasing message sizes push latencies steadily higher until the congestion becomes severe enough (at 256 packets/message) to benefit from ECN. As the message size continues to increase, the length of congestion events also continues to increase and ECN becomes increasingly more effective at reaching a steady state. The networks with stashing see very little perturbation across the message sizes and always outperform the baseline, showing again how stashing effectively absorbs the congestion transients until ECN kicks in.

We have shown that the stashing architecture can be used to significantly improve the latency distribution of a network experiencing congestion. Even though the fraction of packets suffering extremely long latencies is small, the impact can be significant at the application level, depending on the operations carried in those packets. For example, in collective operations such as barriers and reductions, a few long-latency packets will nearly always extend the critical paths of those operations.

VII. Related Work

Dynamically allocating otherwise idle resources is not a new concept. Architectures such as dynamic-allocated multiple-queues (DAMQ) [20] have long been proposed and deployed in switches to efficiently share buffers between multiple co-locating FIFOs. Other works have proposed flexible buffer management systems for sharing between variable numbers of VCs in both network-on-chip [35] and large-scale networks [36]. These schemes largely focus on the sharing of buffers within a single port of the switch. Our work can be viewed as a progression to the next level of dynamic buffer sharing. With stashing, we have expanded the scope of sharing to the entire switch chip without affecting the normal operation of the switch.

This chip-level dynamic sharing of buffers is made possible by the abundance of bandwidth offered by the tiled switch architecture, first proposed in [17], and implemented in [18]. Since then, other tiled switches with different port counts and internal configurations have been deployed [5], [6]. However, all these designs still use dedicated input and output buffering per port. Our design is a significant improvement on the existing architecture, allowing the otherwise unused bandwidth in the switch to be repurposed for moving stashing packets without penalty.

With regard to our use case studies, there have been many recent works on the various aspects of ECN-type protocols. These include how congestion should be detected in the network [23], [24], precision of congestion feedback [37], [38], and endpoint response when congestion is detected [39]. These studies focus on the various aspects of the ECN algorithm and do not directly address the transient congestion during ECN activation. Our congestion use case is orthogonal to these studies and can be paired with any of these algorithms to improve their transient response to congestion.

Using extra buffers to alleviate HoL blocking caused by tree-saturation has been proposed by Escudero-Sahuquillo, *et al.* [40], [41]. They designed a method to identify and separate the congested traffic into its own set of dedicated queues. Our congestion control use case shares some commonalities, since both works use additional buffering in the network to mitigate the impact of congestion. However, our contribution goes beyond, as we propose to use existing buffering resources available on the switch as opposed to adding dedicated queues. In addition, our stashing mechanisms work in collaboration with ECN protocols to ensure that the packets are not stashed indefinitely, allowing the stashing buffers to be used more dynamically as congestion comes and goes in a network.

VIII. Conclusion

This paper introduced the stashing architecture, which provides global access to the idle buffer resources of a tiled switch using available internal bandwidth. We showed how to create fully accessible *stashing buffers* from underutilized port memory by statically partitioning the switch input and output buffers and then adding two internal virtual channels to facilitate access. This enables a variety of network features to be realized without adding additional storage to the switch.

We demonstrated and evaluated the architecture with two separate case studies: using stashing to implement network end-to-end retransmission, and improving ECN-type congestion control by alleviating HoL blocking. Our simulations of end-to-end retransmission showed that storing to the stashing buffers does not degrade the normal network performance. Our simulations of ECN improvement showed that the worst-case packet latencies suffered by victim applications as a result of network congestion could be dramatically reduced when stashing was used to absorb transients.

As the cost of producing cutting-edge technology continues to increase, it becomes prohibitively expensive to design and build custom components. Techniques like stashing allow new capabilities of commodity designs to be unlocked by small architectural modifications, resulting in low-cost solutions to difficult problems. This advantage is clearly demonstrated by our end-to-end storage implementation, which enables highly reliable communication for directly attached accelerators.

REFERENCES

[1] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 77–88. [Online]. Available: https://doi.org/10.1109/ISCA.2008.19

[2] M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang, "Tianhe-1a interconnect and message-passing services," *IEEE Micro*, vol. 32, no. 1, pp. 8–20, Jan 2012.

[3] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, and E. Zahavi, "Dragonfly+: Low cost topology for scaling datacenters," in *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, Feb 2017, pp. 1–8.

[4] 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 *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 1–9.

[5] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable hpc system based on a dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 103:1–103:9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389136

[6] Y. Dai, K. Wang, G. Qu, L. Xiao, D. Dong, and X. Qi, "A scalable and resilient microarchitecture based on multiport binding for high-radix router design," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 429–438.

[7] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 83–87. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2010.23

[8] N. R. Adiga *et al.*, "An overview of the bluegene/l supercomputer," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=762761.762787

[9] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, "The ibm blue gene/q interconnection fabric," *IEEE Micro*, vol. 32, no. 1, pp. 32–43, 2012.

[10] J. Dongarra, "Report on the Sunway TaihuLight System," University of Tennessee, Department of Electrical and Computer Science, Tech. Rep. UT-EECS-16-742, Jun 2016.

[11] Cable News Network. (2017, Jul.) Japan is building the fastest supercomputer ever made. https://www.cnn.com/2017/06/13/tech/supercomputer-japan/index.html. Accessed: 2018-03-19.

[12] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC Series Network," Cray, Inc., Tech. Rep. WP-Aries01-1112, 2012.

[13] Intel Corporation. Intel omni-path cable products. https://www.intel.com/content/www/us/en/products/network-io/high-performance-fabrics/omni-path-cables.html. Accessed: 2018-03-15.

[14] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[15] *InfiniBand Architecture Specification, Volume 1, Release 1.3*, InfiniBand Trade Association, Mar 2015.

[16] NVIDIA Corporation. (2018) Nvlink fabric. https://www.nvidia.com/en-us/data-center/nvlink. Accessed: 2018-05-24.

[17] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta, "Microarchitecture of a high-radix router," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 420–431. [Online]. Available: https://doi.org/10.1109/ISCA.2005.35

[18] S. Scott, D. Abts, J. Kim, and W. J. Dally, "The blackwidow high-radix clos network," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 16–28.

[19] M. Katevenis, P. Vatsolaki, and A. Efthymiou, "Pipelined memory shared buffer for vlsi switches," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 39–48.

[20] Y. Tamir and G. L. Frazier, "High-performance multiqueue buffers for vlsi communication switches," in *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, May 1988, pp. 343–354.

[21] Wikimedia Foundation, Inc. (2018, Mar.) End-to-end principle. https://en.wikipedia.org/wiki/End-to-end_principle. Accessed: 2018-03-28.

[22] E. Gran, M. Eimot, S.-A. Reinemo, T. Skeie, O. Lysne, L. Huse, and G. Shainer, "First experiences with congestion control in infiniband hardware," in *Parallel Distributed Processing, 2010 IEEE International Symposium on*, pp. 1 –12.

[23] J. Duato, I. Johnson, J. Flich, F. Naven, P. Garcia, and T. Nachiondo, "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks," in *High-Performance Computer Architecture. 11th International Symposium on*, 2005, pp. 108 – 119.

[24] J.-L. Ferrer, E. Baydal, A. Robles, P. Lopez, and J. Duato, "A scalable and early congestion management mechanism for mins," in *Parallel, Distributed and Network-Based Processing, 18th Euromicro International Conference on*, 2010.

[25] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–63, Sep. 2001, updated by RFCs 4301, 6040. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3168.txt

[26] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally, "Network congestion avoidance through speculative reservation," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2012.6169047

[27] N. Jiang, L. Dennison, and W. J. Dally, "Network endpoint congestion control for fine-grained communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 35:1–35:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807600

[28] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[29] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally, "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.

[30] D. U. Becker and W. J. Dally, "Allocator implementations for network-on-chip routers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–12.

[31] M. Garcia, E. Vallejo, R. Beivide, M. Odriozola, and M. Valero, "Efficient routing mechanisms for dragonfly networks," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 582–592.

[32] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen, "A simulator for large-scale parallel computer architectures," *Int. J. Distrib. Syst. Technol.*, vol. 1, no. 2, pp. 57–73, Apr. 2010. [Online]. Available: http://dx.doi.org/10.4018/jdst.2010040104

[33] B. Austin and N. J. Wright, "Measurement and interpretation of microbenchmark and application energy use on the cray xc30," in *Proceedings of the 2Nd International Workshop on Energy Efficient Supercomputing*, ser. E2SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 51–59. [Online]. Available: http://dx.doi.org/10.1109/E2SC.2014.7

[34] (2013) Characterization of the DOE mini-apps. [Online]. Available: http://portal.nersc.gov/project/CAL/designforward.htm

[35] C. A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, M. S. Yousif, and C. R. Das, "Vichar: A dynamic virtual channel regulator for network-on-chip routers," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 333–346.

[36] P. Fuentes, E. Vallejo, R. Beivide, C. Minkenberg, and M. Valero, "Flexvc: Flexible virtual channel management in low-diameter networks," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 842–854.

[37] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms:

Congestion control theory and ieee standardization," in *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, sept. 2008, pp. 1270 –1277.

[38] J.-L. Ferrer, E. Baydal, A. Robles, P. Lopez, and J. Duato, "Congestion management in mins through marked and validated packets," in *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 254–261. [Online]. Available: http://dx.doi.org/10.1109/PDP.2007.32

[39] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[40] J. Escudero-Sahuquillo, P. García, F. Quiles, J. Flich, and J. Duato, "Fbicm: Efficient congestion management for high-performance networks using distributed deterministic routing," in *Proceedings of the 15th International Conference on High Performance Computing*, ser. HiPC'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 503–517. [Online]. Available: http://dl.acm.org/citation.cfm?id=1791889.1791941

[41] J. Escudero-Sahuquillo, E. G. Gran, P. J. Garcia, J. Flich, T. Skeie, O. Lysne, F. J. Quiles, and J. Duato, "Combining congested-flow isolation and injection throttling in hpc interconnection networks," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 662–672. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2011.80

APPENDIX

A. Abstract

This artifact contains the directions to the simulation infrastructure used to generate results in the paper *Exploiting Idle Resources in a High-Radix Switch for Supplemental Storage*.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: Full system and network simulation**
- **Program: SST/Macro 6.1, Modified Booksim 2.0**
- **Compilation: GCC 7.3 or higher**
- **Dataset: Department of Energy DesignForward application traces**
- **Run-time environment: Linux**
- **Hardware: Any**
- **Publicly available?: Yes**

2) How software can be obtained (if available): Code for various simulator components can be downloaded from GitHub. We made significant modifications to the Booksim simulator to implement the switch architectures presented in this study. We have also added interfacing code to allow SST/Macro to use Booksim as the network layer to produce results from MPI application traces.

SST/Macro6.1: https://github.com/sstsimulator/sst-macro/tree/v6.1.0_Final
Modified Booksim: https://github.com/njiang37/sc18_stashing_switch

3) Hardware dependencies: None.

4) Software dependencies:

- Autoconf: 2.68 or later
- Automake: 1.11.1 or later
- Libtool: 2.4 or later
- GCC 7.3.0 or late
- Flex 2.6.0
- Bison 3.0.4
- MATLAB 2010 or later (for reading statistics files)

5) Datasets: MPI traces from the DesignForward project can be downloaded from http://portal.nersc.gov/project/CAL/designforward.htm

C. Installation

Clone the SST/Macro repository and version

```
$ git clone https://github.com/sstsimulator/sst-macro.git
$ cd sst-macro
$ git checkout v6.1.0_Final
```

SST/Macro can be compiled and installed following the user guide included with the simulator.

Clone the modified Booksim repository

```
$ git clone https://github.com/njiang37/sc18_stashing_switch
```

Follow the README file in the Booksim repository for a detailed walk-through of integrating the two simulators. The integration process assumes a freshly installed SST/Macro 6.1.0 repository since it requires minor modifications to SST/Macro files to instantiate the Booksim network.

D. Experiment workflow

Experiments are run on the simulator using different configuration files. Run scripts and configuration files for both synthetic traffic experiments and MPI traffic experiments are included in the Booksim repository's *sim_scripts* directory.

E. Evaluation and expected result

The network simulator prints out periodic updates of network status such as latency and throughput which are collected in a log file and parsed. The network simulation also generates a statistics file in MATLAB m-file format. Cumulative distribution of network latency is generated from these statistics files using MATLAB. Output log parser and MATLAB script files are included in the Booksim repository's *result_scripts* directory.

For application traces, the SST/Macro simulator outputs the estimated execution time of the trace.

F. Experiment customization

Various options in the SST/Macro or Booksim configuration files can be changed to modify system and network behavior. See the user guides included with each simulator for customization details.