An In-Depth Analysis of the Slingshot Interconnect

Daniele De Sensi  
Department of Computer Science  
ETH Zurich  
ddesensi@ethz.ch

Salvatore Di Girolamo  
Department of Computer Science  
ETH Zurich  
salvatore.digirolamo@inf.ethz.ch

Kim H. McMahon  
Hewlett Packard Enterprise  
kim.mcmahon@hpe.com

Duncan Roweth  
Hewlett Packard Enterprise  
duncan.roweth@hpe.com

Torsten Hoefler  
Department of Computer Science  
ETH Zurich  
torsten.hoefler@inf.ethz.ch

Abstract—The interconnect is one of the most critical components in large scale computing systems, and its impact on the performance of applications is going to increase with the system size. In this paper, we will describe SLINGSHOT, an interconnection network for large scale computing systems. SLINGSHOT is based on high-radix switches, which allow building exascale and hyperscale datacenters networks with at most three switch-to-switch hops. Moreover, SLINGSHOT provides efficient adaptive routing and congestion control algorithms, and highly tunable traffic classes. SLINGSHOT uses an optimized Ethernet protocol, which allows it to be interoperable with standard Ethernet devices while providing high performance to HPC applications. We analyze the extent to which SLINGSHOT provides these features, evaluating it on microbenchmarks and on several applications from the datacenter and AI worlds, as well as on HPC applications. We find that applications running on SLINGSHOT are less affected by congestion compared to previous generation networks.

Index Terms—interconnection network, dragonfly, exascale, datacenters, congestion

I. INTRODUCTION

The first US exascale supercomputer will be built within two years, marking an important milestone for computing systems. Exascale computing has been a long-awaited goal, which required significant contributions both from academic and industrial research. One of the most critical components having a direct impact on the performance of such large systems is the interconnection network (interconnect). Indeed, by analyzing the performance of the Top500 supercomputers [1] when executing HPL [2] and HPCG [3], two benchmarks commonly used to assess supercomputing systems, we can observe that HPCG is typically characterized by ~50x lower performance compared to HPL. Part of this performance drop is caused by the higher communication intensity of HPCG, clearly showing that, among others, an efficient interconnection network can help in exploiting the full computational power of the system. The impact of the interconnect on the performance of supercomputing systems increases with the scale of the system, highlighting the need for novel and efficient solutions.

Both the HPC and datacenter communities are following a path towards convergence of HPC, data centers, and AI/ML workloads, which poses new challenges and requires new solutions. Workloads are becoming much more data-centric, and large amounts of data need to be exchanged with the outside world. Due to the wide adoption of Ethernet in datacenters, interconnection networks should be compatible with standard Ethernet, so that they can be efficiently integrated with standard devices and storage systems. Moreover, many data center workloads are latency-sensitive. For such applications, tail latency is much more relevant than the best case or average latency. For example, web search nodes must provide 99th percentile latencies of a few milliseconds [4]. This is also a relevant problem for HPC applications, whose performance may strongly depend on messages latency, especially when using many global or small messages synchronizations. Despite the efforts in improving the performance of interconnection networks, tail latency still severely affects large HPC and data center systems [4]–[7].

To address these issues, Cray [8] recently designed the SLINGSHOT interconnection network. SLINGSHOT will power all three announced US exascale systems [8]–[10] and numerous supercomputers that will be deployed soon. It provides some key features, like adaptive routing and congestion control, that make it a good solution for HPC systems but also for cloud data centers. SLINGSHOT switches have 64 ports with 200 Gb/s each and support arbitrary network topologies. To reduce tail latencies, SLINGSHOT offers advanced adaptive routing, congestion control, and quality of service (QoS) features. Those also protect applications from interference, sometimes referred to as network noise [5], [11], caused by other applications sharing the interconnect. Lastly, SLINGSHOT brings HPC features to Ethernet, such as low latency, low packet overhead, and optimized congestion control, while maintaining industry standards. In SLINGSHOT, each port of the switch can negotiate the available Ethernet features with the attached devices, and can communicate with existing Ethernet devices using standard Ethernet protocols, or with other SLINGSHOT switches and NICs by using SLINGSHOT specific additions. This allows the network to be fully interoperable with existing Ethernet equipment while at the same time providing good performance for HPC systems.

In this study, we experimentally analyze SLINGSHOT’s performance features to guide researchers, developers, and

1Cray is a Hewlett Packard Enterprise (HPE) company since 2019.
system administrators. We use Mellanox ConnectX-5 100 Gb/s Ethernet NICs to test the ability of SlingShot to deal with standard RDMA over Converged Ethernet (RoCE) traffic. Moreover, by doing so we can analyze the impact of the switch on the end-to-end performance by factoring out some of the improvements on the Ethernet protocol introduced by SlingShot. We first analyze the latencies of a quiet system. Then, we analyze the impact of congestion on both microbenchmarks and real applications for different configurations, showing that SlingShot is only marginally affected by network noise. To further show the benefits of the congestion control algorithm, we compare SlingShot to Cray’s previous Aries network, which has a similar topology and uses a similar routing algorithm.

II. SLINGSHOT ARCHITECTURE

We now describe the SlingShot interconnection network. We first introduce the Rosetta switch and show how switches can be connected to form a Dragonfly [12] topology. We then dive into specific features of SlingShot such as adaptive routing, congestion control, and quality of service management. Lastly, we describe the main characteristics of the SlingShot additions to Ethernet and the software stack.

A. Switch Technology (Rosetta)

The core of the SlingShot interconnect is the Rosetta switch, providing 64 ports at 200 Gb/s per direction. Each port uses four lanes of 56 Gb/s Serializer/Deserializer (SerDes) blocks using Pulse-Amplitude Modulation (PAM-4) modulation. Due to Forward Error Correction (FEC) overhead, 50Gb/s can be pushed through each lane. The Rosetta ASIC consumes up to 250 Watts and is implemented on TSMCs 16 nm process. Rosetta is composed by 32 peripheral function blocks and 32 tile blocks. The peripheral blocks implement the SerDes, Medium Access Control (MAC), Physical Coding Sublayer (PCS), Link Layer Reliability (LLR), and Ethernet lookup functions.

The 32 tile blocks implement the crossbar switching between the 64 ports, but also adaptive routing and congestion management functionalities. The tiles are arranged in four rows of eight tiles, with two switch ports handled per tile, as shown in Figure 1. The tiles on the same row are connected through 16 per-row buses, whereas the tiles on the same column are connected through dedicated channels with per-tile crossbars. Each row bus is used to send data from the corresponding port to the other 16 ports on the row. The per-tile crossbar has 16 inputs (i.e., from the 16 ports on the row) and 8 outputs (i.e., to the 8 ports on the column). For each port, a multiplexer is used to select one of the four inputs (this is not explicitly shown in the figure for the sake of clarity). Packets are routed to the destination tile through two hops maximum. Figure 1 shows an example: if a packet is received on Port 19 and must be routed to Port 56, the packet is first routed on the row bus, then it goes through the 16-to-8 crossbar highlighted in the picture, and then down a column channel to Port 56. Thanks to the hierarchical structure of the tiles, there is no need for a 64 ports arbiter, and the packets only incur in a 16 to 8 arbitration.

![Fig. 1: Rosetta switch tiled structure.](image_url)

The 32 tiles in Rosetta implement a crossbar between the 64 ports. For performance and implementation reasons, the crossbar is physically composed by different function-specific crossbars, each handling a different aspect of the switching traffic:

- **Requests to Transmit** To avoid head-of-line blocking (HOL) [13], Rosetta relies on a virtual output-queued architecture [14], [15] where the routing path is determined before sending the data. The data is buffered in the input buffers until the resources are available, guaranteeing no further blocks. Before forwarding the data, a request-to-transmit is sent to the tile corresponding to the switch output port. When a grant to transmit is received from the output port, the data is forwarded.

- **Grants to Transmit** Grants to transmit are sent by the tile handling the output port to the tile from which the switch received the packet. In the previous example, the grants would be transmitted from the tile handling Port 56, to the tile handling Port 19. Grants are used to notify the permission to forward the data to the next hop. The use of requests and grants to transmit is a central piece of the QoS management.

- **Data** Data is sent on a wider crossbar (48B). To speed up the processing, Rosetta parses and processes the packet header as soon as it arrives, even if the data might still be arriving.

- **Request Queue Credits** Credits provide an estimation of queue occupancy. This information is then used by the adaptive routing algorithm (see Section II-C) to estimate the congestion of different paths and to select the least congested one.

- **End-to-End Ack** End-to-End acknowledgments are used to track the outstanding packets between every pair of network endpoints. This information is used by the congestion control protocol (see Section II-D).

By using physically separated crossbars, SlingShot guarantees that different types of messages do not interfere with each other and that, for example, large data transfers do not slow down requests and grants to transmit.
To analyze the impact of the switch architecture on the latency, we report in Figure 2 the latency of the switch when dealing with RoCE traffic. It is worth remarking that, because we are using standard RoCE NICs, the NIC sends plain Ethernet frames, and we cannot exploit all the features of SLINGSHOT’s specialized Ethernet protocol (Section II-F). Some of the features like link-level reliability and propagation of congestion information are however still used in the switch-to-switch communications. To compute the latency of the switch, we consider the latency difference between 2-hops and 1-hop latencies (we provide details on the topology in Section II-B). We observe that ROSETTA has a mean and median latency of 350 nanoseconds, with all the distribution lying between 300 and 400 nanoseconds, except for a few outliers.

B. Topology

ROSETTA switches can be arranged into any arbitrary topology. Dragonfly [12] is the default topology for SLINGSHOT-based systems, and it is the topology we refer to in the rest of the paper. Dragonfly is a hierarchical direct topology, where all the switches are connected to both computing nodes and other switches. Sets of switches are connected between each other forming so-called groups. The switches inside each group may be connected by using an arbitrary topology, and groups are connected in a fully connected graph. In the SLINGSHOT implementation of Dragonfly (shown in Figure 3), each ROSETTA switch is connected to 16 endpoints through copper cables (up to 2.6 meters), using the remaining 48 ports for inter-switches connectivity. The partitioning of these 48 ports between inter- and intra-group connectivity, as well as the number of switches per group, depends on the size of the system. In SLINGSHOT, the switches inside a group are always fully connected through copper cables. Switches in different groups are connected through long optical cables (up to 100 meters). Due to the full-connectivity both within the group and between groups, this topology has a diameter of 3 switch-to-switch hops.

Thanks to the low-diameter, applications performance only marginally depend on the specific node allocation. We report in Figure 4 the latency and the bandwidth between nodes at different distances, and for different message sizes on an isolated system. We consider nodes connected to ports on the same switch (1 inter-switch hop), connected to two different switches in the same group (2 inter-switch hops), and connected to two different switches in two different groups (3 inter-switch hops). For the same switch case, we observed no significant difference when using two ports on the same switch tile or on two different tiles.

First, we observe that, in the worst case, the node allocation has only a 40% impact on the latency for 8B messages and that, starting from 16KiB messages we observe less than 10% difference in latency between the different node distances. The same holds for bandwidth, with less than 15% difference between the different distances across all the message sizes. In some cases, we observe a slightly higher bandwidth when the nodes are in two different groups, because more paths connect the two nodes, increasing the available bandwidth.

In the largest system (shown in Figure 3), each group has 32 switches (for a total of 32 × 16 = 512 nodes, and switches inside each group are fully connected by using 31 switches ports. The remaining 17 ports from each switch are used to globally connect all the groups in a fully connected network. In this specific case, because each group contains 32 switches and each switch uses 17 ports to connect to other groups, each group has 32 × 17 = 544 connection towards other groups. This leads to a system having 545 groups, each of which is connected to 512 nodes, for a total of 279,090 endpoints at full global bandwidth[3]. This number of endpoints satisfies both

---

3In practice, the addressing scheme limits the number of groups to 511, for a total of 261,632 nodes.
exascale supercomputers and hyperscale data centers demand. Indeed, this is larger than the number of servers used in data centers [16], and much larger than the number of nodes used by Summit [17], the most performing supercomputer at the time being, that currently relies on 4,608 nodes and delivers 200P FLOP/s. Thanks to this large number of endpoints, each computing node can have multiple connections to the same network, increasing the injection bandwidth and improving network resiliency in case of NICs failures.

C. Routing

In Dragonfly networks (including SLINGSHOT), any pair of nodes is connected by multiple *minimal* and *non-minimal* paths [12], [18]. For example, by considering the topology in Figure 1, the minimal path connecting N0 to N496 includes the switches S0 and S31. In smaller networks, due to links redundancy, multiple minimal paths are connecting any pair of nodes [13]. On the other hand, a possible non-minimal path involves an intermediate switch that is directly connected to both S0 and S31. The same holds for nodes located in different groups. In this case, a non-minimal path crosses an intermediate group.

Sending data on minimal paths is clearly the best choice on a quiet network. However, in a congested network, with multiple active jobs, those paths may be slower than longer but less congested ones. To provide the highest throughput and lowest latency, SLINGSHOT implements adaptive routing: before sending a packet, the source switch estimates the load of up to four minimal and non-minimal paths and sends the packet on the best path, that is selected by considering both the paths’ congestion and length. The congestion is estimated by considering the total depth of the request queues of each output port. This congestion information is distributed on the chip by using a ring to all the forwarding blocks of each input port. It is also communicated between neighboring switches by carrying it in the acknowledgement packets. The total overhead for congestion and load information is an average of four bytes in the reverse direction for every packet in the forward direction. As more packets take non-minimal paths and therefore average hop count per packet increases, both the latency and the link utilization increase. Therefore, SLINGSHOT adaptive routing biases packets to take minimal paths more frequently, to compensate for the higher cost of non-minimal paths.

D. Congestion Control

Two types of congestion might affect an interconnection network: *endpoint* congestion, and *intermediate* congestion [6]. The endpoint congestion mostly occurs on the last-hop switches, whereas intermediate congestion is spread across the network. Adaptive routing improves network utilization and application performance by changing the path of the packets to avoid intermediate congestion. However, even if adaptive routing can bypass congested intermediate switches, all the paths between two nodes are affected in the same way by endpoint congestion. As we show in Section III-A, this was a relevant issue on other networks, particularly for many-to-one traffic. In this case, due to the highly congested links on the receiver side, the adaptive routing would spread the packets over the different paths but without being able to avoid congestion, because it is occurring in the last hop.

Congestion control helps in mitigating this problem by decreasing the injection bandwidth of the nodes generating the congestion. However, existing congestion control mechanisms (like ECN [19] and QCN [20], [21]) are not suited for HPC scenarios. They work by marking packets that experience congestion. When a node receives a packet that has been marked, it asks the sender to slow down its injection rate. These congestion control algorithms work relatively well in presence of large volume and stable communications (known as *elephant flows*), but tend to be fragile, hard to tune [22], [23], and generally unsuitable for bursty HPC workloads. Indeed, in standard congestion control algorithms, the control loop is too long to adapt fast enough, and while converging to the correct transmission rate, the offending traffic can still interfere with other applications.

To mitigate this problem, SLINGSHOT introduces a sophisticated congestion control algorithm, entirely implemented in hardware, that tracks every in-flight packet between every pair of endpoints in the system. SLINGSHOT can distinguish between jobs that are victims of congestion and those who are contributing to congestion, applying stiff and fast back-pressure to the sources that are contributing to congestion. By tracking all the endpoints pairs individually, SLINGSHOT only throttles those streams of packets who are contributing to the endpoint congestion, without negatively affecting other jobs or other streams of packets within the same job who are not contributing to congestion. This frees up buffers space for the other jobs, avoiding HOL blocking across the entire network, and reducing tail latencies, which are particularly relevant for applications characterized by global synchronizations.

The approach to congestion control adopted by SLINGSHOT is fundamentally different from more traditional approaches such as ECN-based congestion control [19], [20], and leads to good performance isolation between different applications, as we show in Section III-A.

E. Quality of Service (QoS)

Whereas congestion control partially protects jobs from mutual interference, jobs can still interfere with each other. To provide complete isolation, in SLINGSHOT jobs can be assigned to different traffic classes, with guaranteed quality of service. QoS and congestion control are orthogonal concepts. Indeed, because traffic classes are expensive resources requiring large amounts of switch buffers space, each traffic class is typically shared among several applications, and congestion control still needs to be applied within a traffic class.

Each traffic class is highly tunable and can be customized by the system administrator in terms of priority, packets ordering required, minimum bandwidth guarantees, maximum bandwidth constraint, lossiness, and routing bias [5]. The system administrator guarantees that the sum of the minimum...
bandwidth requirements of the different traffic classes does not exceed the available bandwidth. Network traffic can be assigned to traffic classes on a per-packet basis. The job scheduler will assign to each job a small number of traffic classes, and the user can then select on which class to send its application traffic. In the case of MPI, this is done by specifying the traffic class identifier in an environment variable. Moreover, communication libraries could even change traffic classes at a per-message (or per-packet) granularity. For example, MPI could assign different collective operations to different traffic classes. For example, it may assign latency-sensitive collective operations such as MPI_Barrier and MPI_Allreduce to high-priority and low-bandwidth traffic classes, and bulk point-to-point operations to higher bandwidth and lower priority classes.

Traffic classes are completely implemented in the switch hardware. A switch determines the traffic class required for a specific packet by using the Differentiated Services Code Point (DSCP) tag in the packet header [24]. Based on the value of the tag, the switch assigns the packet to one of the multiple virtual queues. Each switch will allocate enough buffers to each traffic class to achieve the desired bandwidth, whereas the remaining buffers will be dynamically allocated to the traffic which is not assigned to any specific traffic class.

F. Ethernet Enhancements

To improve interoperability, and to better suit datacenters scenarios, SLINGSHOT is fully Ethernet compatible, and can seamlessly be connected to third-party Ethernet-based devices and networks. SLINGSHOT provides additional features on top of standard Ethernet, improving its performance and making it more suitable for HPC workloads. SLINGSHOT uses this enhanced protocol for internal traffic, but it can mix it with standard Ethernet traffic on all ports at packet-level granularity. This allows SLINGSHOT to achieve high-performance, while at the same time being able to communicate with standard Ethernet devices, allowing it to be used efficiently in both supercomputing and datacenter worlds.

To improve performance, SLINGSHOT reduces the 64 Bytes minimum frame size to 32 Bytes, allows IP packets to be sent without an Ethernet header, and removes the inter-packet gap. Lastly, SLINGSHOT provides resiliency at different levels by implementing low-latency Forward Error Correction (FEC) [25], Link-Level Reliability (LLR) to tolerate transient errors, and lanes degrade [26] to tolerate hard failures. Moreover, the SLINGSHOT NIC provides end-to-end retry to protect against packet loss. These are relevant features in high-performance networks. For example, FEC is required for all Ethernet systems at 100Gb/s or higher, independently from the system size, and LLR is useful in large systems (such as hyperscale data centers) to localize the error handling and reduce end-to-end retransmission.

G. Software Stack

Communication libraries can either use the standard TCP/IP stack or, in case of high-performance communication libraries such as MPI [27], [28], Chapel [29], PGAS [30] and SHMEM [31], the libfabric interface [32]. Cray contributed with new features to the libfabric open-source verbs provider and RxM utility provider to support the SLINGSHOT hardware. All HPC traffic is layered over RDMA over Converged Ethernet (RoCEv2) and data is sent over the network through packets containing up to 4KiB of data plus headers and trailers. Headers and trailers include Ethernet (26 bytes including the preamble), IPv4 (20 bytes), UDP (8 bytes), InfiniBand (14 bytes), and an additional RoCEv2 CRC (4 bytes), for a total of 62 bytes. Cray MPI is derived from MPICH [33] and implements the MPI-3.1 standard. Proprietary optimizations and other enhancements have been added to Cray MPI targeted specifically for the SLINGSHOT hardware. Any MPI implementation supporting libfabric can be used out of the box on SLINGSHOT. Moreover, standard API for some features, like traffic classes, have been recently added to libfabric and could be exploited as well. We report in Figure 5 the latencies for different message sizes and for different network protocols. We observe that for small message sizes, MPI adds only a marginal overhead to libfabric.

![Fig. 5: Half round trip time (RTT/2) for different message sizes (x-axis) and software layers.](image)

Moreover, we show in Figure [the bisection bandwidth (i.e., the bandwidth when half of the nodes send data to the other half of the nodes and vice versa) and the MPI_Alltoall bandwidth on SHANDY, a SLINGSHOT-based system using 1024 nodes (see Section III for details). We report the results for different processes per node (PPN) and different message sizes. This system is composed of eight groups, and all the bisection cuts cross the same number of links. In this system, each group has 56 global links out of 112 (8 towards each other group), to match the injection bandwidth. Each of the 4 groups in one partition is connected to each of the 4 groups in the other partition, and the total number of links crossing a bisection cut is 4·4·8 = 128. Because each link has a 200Gb/s bandwidth, and we are sending traffic in both directions, the peak bisection bandwidth is 128 · 200Gb/s · 2 = 6.4Tb/s.

In an all-to-all communication, each node sends 7/8 of the traffic to nodes in the other 7 groups and 1/8 of the traffic to nodes in the same group. Because this system has 56·8 = 448 global links, the all-to-all maximum bandwidth is 8/7 · 448 · 200Gb/s = 12.8Tb/s. Note that MPI_Alltoall can achieve twice the bisection bandwidth because half of the connections terminate in the same partition [34]. The plot shows that the MPI_Alltoall reaches more than the 90%
of the theoretical peak bandwidth, without any packet loss. We observe a performance drop for 256 bytes messages because, to reduce memory usage, the MPI implementation switches to a different algorithm \[35\] for messages larger than 256 bytes.

III. Performance Study

We now study the performance of the SLINGSHOT interconnect on real applications and microbenchmarks, by focusing on two key features of SLINGSHOT, namely congestion control and quality of service management. For our analysis, we consider the following systems:

- **CRYSTAL**: A system based on the Cray ARIES interconnect \[48\]. This system has 698 nodes. The CPUs on the nodes are *Intel Xeon E5-269x*. The system is composed of two groups, each containing at most 384 nodes.

- **MALBEC**: A SLINGSHOT system with 484 nodes. CPUs on the nodes are either *Intel Xeon Gold 61xx* or *Intel Xeon Platinum 81xx* CPUs. The system is composed of four groups, each containing at most 128 nodes. Each group is connected to each other group through 48 global links operating at 200Gb/s each. Each node has a Mellanox ConnectX-5 EN NIC.

- **SHANDY**: A SLINGSHOT system with 1024 nodes. Compute nodes are equipped with *AMD EPYC Rome* 64 cores CPUs. The system is composed of eight groups, each containing 128 nodes. Each group is connected to each other group through 56 global links operating at 200Gb/s each. Each node has two Mellanox ConnectX-5 EN NICs, each connected to a different switch of the same network, allowing a better load distribution and resilience in the event of NICs failures.

We consider two SLINGSHOT systems, of different size, to analyze the performance at different system scales. For all the experiments, we booked these systems for exclusive use, to have a controlled environment and avoid interference caused by other users.

A. Congestion Control

To evaluate the ability of SLINGSHOT to react to congestion, we divide the nodes in the system in two partitions: *victim* nodes and *aggressor* nodes. The aggressor nodes generate congestion that impacts the performance of victim nodes. We consider two types of congestion patterns: *endpoint* congestion and *intermediate* congestion, and we use the GPCNet code \[6\] to generate those congestion patterns. We generate endpoint congestion through a *many-to-one* (incast) communication pattern, where a number of nodes send data to the same endpoint by using MPI_Put, and intermediate congestion by using an *all-to-all* pattern implemented through MPI_Sendrecv. Both aggressors exchange 128KiB messages. This decision is based on characterization studies on production systems, that show an average message size of \(\sim 10^3\) bytes both in collective and point-to-point communications \[49\].

We consider the victim applications described in Table \[1\]. Moreover, we also analyze the impact of congestion on microbenchmarks, include standard MPI operations, and the ember microbenchmarks \[50\] reproducing some common communication patterns in HPC applications (*halo3d*, *sweep3d*, and *incast*). We first consider the results on 512 nodes. Then, we show the results for different node counts. We consider the following victim/aggressor splits: 460/52 (\(\sim 90\% / 10\%\)), 256/256 (\(\sim 50\% / 50\%\)) and 53/459 (\(\sim 10\% / 90\%\)). Because the implementation of some MPI collectives changes according to the number of nodes used, we have chosen these splits so that we run the victim with both power of two (256), even (460) and odd (53) number of nodes. To further increase the generated congestion, in some experiments we increase the number of processes per node (PPN) used by the aggressor. Each node used by the aggressor spawns PPN processes, each of them performing the same communications. Namely, the congestion pattern is concurrently executed PPN times.

Moreover, the allocation of the nodes to victims and aggressors determines how many switches and groups are shared between the two jobs and has a direct impact on the performance of the victim. In our experiments, we consider the three well-known allocation placement strategies \[51\] depicted in Figure \[7\]: *linear*, where we allocate the first \(n\) nodes to the victim and the remaining nodes to the aggressor; *interleaved*, where we interleave the nodes allocated to the victim and the aggressor; and *random*, where we randomly allocate the nodes to the victim and the aggressor.

We make sure that the data we report is statistically sound \[52\]: for each microbenchmark, we execute the victim at least 200 times and for at least 4 seconds. We stop the benchmark when both the previous two conditions are satisfied, and when the 95\% confidence interval is within 5\% of the median. We then consider for each iteration the maximum time among the ranks. For the applications, we
Moreover, a similar degradation due to congestion on Aries has also been observed on other interconnects [6], [11], [53].

Due to the large number of combinations of victims, aggressors, and allocations, we provide a data summary of the linear allocation results as a heatmap in Figure 9. Each element of the heatmap represents the mean congestion impact $C$ [6], i.e.,

$$C = \frac{T_c}{T_i}$$  \hspace{1cm} (1)$$

where $T_i$ is the mean execution time of the victim when executed in isolation, and $T_c$ is the mean execution time of the victim when co-executed with the aggressor. For example, we observe a higher tail latency on Aries when co-executed with the aggressor, whereas we do not observe any relevant effect on Silos. For Sphinx, we observe a smaller degradation because the communication to computation ratio is lower than that of the other applications. Moreover, we observe a higher tail latency on Aries, which further increases in the presence of congestion. It is worth remarking that the congestion impact itself is enough to characterize how much SLINGSHOT is affected by congestion. In addition, we are also comparing SLINGSHOT with an Aries interconnect, to also show the improvements compared to an existing interconnection network. Moreover, a similar performance degradation to that we observed on Aries has also been observed on other interconnects [6], [11], [53].

Table I: Applications used as victim in the congestion tests. We consider both HPC and datacenter (DC) applications. Img-dnn, Xapian, Sphinx and Silo are all single-client, single-server applications, coming from the Tailbench benchmark [47] for latency-sensitive datacenter applications. We selected this subset because it covers a wide range of latencies, from microseconds (Silo) to seconds (Sphinx).

<table>
<thead>
<tr>
<th>Type</th>
<th>Appl.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC</td>
<td>MILC</td>
<td>It is a set of numerical simulation codes working on quantum chromodynamics (QCD) [56]. We use the au3_rmd kernel, that decomposes a four dimensional grid, and mostly performs point-to-point neighbour communications and global reductions [57].</td>
</tr>
<tr>
<td>HPCG</td>
<td>A set of communication and computational patterns matching a wide set of applications. It relies on sparse triangular solvers and preconditioned conjugate gradient algorithms [6]. It mostly uses stencil communications and global reductions.</td>
<td></td>
</tr>
<tr>
<td>LAMMPS</td>
<td>A molecular dynamics code that models an ensemble of particles in a liquid, solid, or gaseous state [38]. This kernel performs reductions and point-to-point blocking and non-blocking communications, between nodes at different distances.</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform on a 3D domain [39]. It employs broadcasts, scatters, and point-to-point communications [40].</td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>Resnet-proxy</td>
<td>This is a ML/AI proxy application [41], reproducing the communication phases of a Deep500 benchmark [42].</td>
</tr>
<tr>
<td></td>
<td>Silo</td>
<td>A fast in-memory transactional database [43]. Widely used in online transaction processing systems (OLTP).</td>
</tr>
<tr>
<td></td>
<td>Sphinx</td>
<td>A search engine [44] using a search index built from a snapshot of the English version of Wikipedia. Multiple queries are executed, with a distribution similar to that of online search queries.</td>
</tr>
<tr>
<td></td>
<td>Xapian</td>
<td>A search engine [45] using a search index built from a snapshot of the English version of Wikipedia. Multiple queries are executed, with a distribution similar to that of online search queries.</td>
</tr>
<tr>
<td></td>
<td>Img-dnn</td>
<td>An application using a deep neural network-based autoencoder to identify handwritten characters [46].</td>
</tr>
</tbody>
</table>

We report in Figure 8 the time distribution for the Tailbench applications, both when executed in isolation, and when executed with an incast aggressor, on both Aries and SLINGSHOT. We also annotate the 99th and 95th percentiles, to show the impact of tail latency. We executed these experiments using the linear allocation and a 10%/90% victim/aggressor ratio. For Silo, Xapian and Img-dnn we observe severe performance degradation due to congestion on Aries, whereas we do not observe any relevant effect on SLINGSHOT. For Sphinx, we observe a smaller degradation because the communication to computation ratio is lower than that of the other applications. Moreover, we observe a higher tail latency on Aries, which further increases in the presence of congestion. It is worth remarking that the congestion impact itself is enough to characterize how much SLINGSHOT is affected by congestion. In addition, we are also comparing SLINGSHOT with an Aries interconnect, to also show the improvements compared to an existing interconnection network. Moreover, a similar performance degradation to that we observed on Aries has also been observed on other interconnects [6], [11], [53].

Due to the large number of combinations of victims, aggressors, and allocations, we provide a data summary of the linear allocation results as a heatmap in Figure 9. Each element of the heatmap represents the mean congestion impact $C$ [6], i.e.,

$$C = \frac{T_c}{T_i}$$  \hspace{1cm} (1)$$

where $T_i$ is the mean execution time of the victim when executed in isolation, and $T_c$ is the mean execution time of the victim when co-executed with the aggressor. For example, we observe any relevant effect on Silos, whereas we do not observe any relevant effect on Aries. For the same scenario, we don’t observe any slowdown on SLINGSHOT.

We report the applications and microbenchmarks results using two different (logarithmic) color scales, to better appreciate the differences. Indeed, applications are usually less affected...
by congestion because, differently from microbenchmarks, they also have computation phases. Because communications are just a part of the overall execution time, even when communications are severely affected by congestion, this does not directly translate into a large performance degradation.

As reported in the heatmap, SLINGSHOT is always less affected by congestion compared to ARIES. In the worst case, we observed a 1.3x slowdown on SLINGSHOT, compared to a maximum 93x slowdown on ARIES. Moreover, the congestion impact increases when increasing the fraction of nodes allocated to the aggressor application, and has a larger impact on small message communications, due to the larger impact of end-to-end latency on the overall performance. The effects of congestion can be seen not only on microbenchmarks but also on full applications. For example, LAMMPS is 17x slower when executed together with an incast aggressor with a 50/50 split on ARIES. Intermediate congestion (generated through all-to-all communication), does not significantly affect the systems we are analyzing, because the adaptive routing algorithm successfully routes the packets around the congested links. This means that, the additional load generated by the all-to-all does not manifest as congestion.

Similar trends can be observed also for different node count, higher PPNs, and other allocations. For space reasons, we do not report all the heatmaps for each of these cases. Instead, we summarize each heatmap by showing the distribution of the heatmap elements (congestion impacts, Equation [1]) across all the victim/aggressor combinations. We show the result of this comparison in Figure [10].

First, we show in Figure [10] the congestion impact for different allocations. For example, for the linear allocation, we are showing the same data of Figure [9]. However, instead of showing all the individual congestion impacts, we now report their distribution. For readability purposes, we cut the long tails of the distributions, and we annotate on top of each violin the maximum value. We observe that whereas on ARIES the congestion impact for the linear allocation is never higher than 100, for the interleaved and random allocations we observe values up to 150. We observed a similar effect on SLINGSHOT but on a different scale. In this case, in all but one cases we observe congestion impact values lower than two. Moreover, differently from ARIES, the distribution on SLINGSHOT is less spread, which indicates that the congestion control algorithm is performing well across a wide set of victims and allocations.

In Figure [10], we report a similar analysis, but now the aggressors are using 24 processes per node (PPN) instead of 1, thus generating a higher load on the network. In this case, the impact of congestion increases for ARIES, especially for random allocations. On the other hand, SLINGSHOT is only minimally impacted, showing a maximum congestion impact ∼ 200 times lower than on ARIES.

Lastly, in Figure [10] we report the congestion impact when using fewer nodes (128). Until now, we compared a SLINGSHOT system (SHANDY, 1024 nodes) against a smaller ARIES one (CRYSTAL, 698 nodes). To factor out possible performance variations coming from different system sizes, we now compare CRYSTAL with a smaller SLINGSHOT system (MALBEC, 484 nodes). We also fix the number of nodes per dragonfly group to 64, in order to allocate the same number of groups (two) in both cases. On ARIES, the maximum conges-
Congestion impact goes from 154 (Figure 10 (A)) to 40 (Figure 10 (C)) when using 128 nodes instead of 512. This can be explained by the lower generated traffic (aggressors have now fewer nodes), but also by the higher fraction of available global bandwidth. On SLINGSHOT, the same experiment makes the maximum congestion impact go from 2.3 to 1.5. We conclude that SLINGSHOT is less affected by congestion, even when varying the system size and the number of allocated nodes.

The results of Figure 11 show the congestion impact on the applications when using all the 1024 nodes on SHANDY. We report the data when using a random allocation because that is the one generating the most congestion (see Figure 10). We can observe that even at full system scale the congestion control effectively protects applications from congestion, with a maximum 3.55x slowdown on LAMMPS when 75% of the nodes are allocated to the incast congestor. Data on MILC and HPCG with a 25%/75% aggressor/victim ratio is missing. Indeed, they should run on 768 nodes, but they can only run on a number of nodes which is a power of two.

We complete our analysis on the effects of congestion by analyzing the impact of bursty congestion SLINGSHOT. Indeed, in the previous experiments we always considered persistent congestion, generated by sending messages with a fixed size of 128KiB during the entire victim execution. To analyze the impact of bursty congestion, we execute a 128 byte MPI_Alltoall microbenchmark (victim) with an incast aggressor. This is one of the cases where we observed the highest congestion impact on SLINGSHOT (see Figure 9). We run this test on all the MALBEC nodes, splitting them equally between aggressor and victim, with an interleaved allocation strategy.

We report the results of this analysis in Figure 12. Each heatmap corresponds to a different message size for the incast aggressor. On each heatmap we report the congestion impact when varying the number of messages in a burst (Burst Size, on the y-axis) and the time between two subsequent congestion bursts (Bursts Gap, on the x-axis). For example, the bottom-left element in the first heatmap, represents the case where the aggressor sends $10^6$ consecutive messages, each one containing 8 bytes. Before sending the next burst of $10^6$ messages, the aggressor will wait 1 microsecond.

We observe that the incast aggressor does not affect the victim when sending too small messages or too large messages. Indeed, small messages do not generate enough congestion, whereas for large messages the congestion control algorithm fully kicks in and throttle the aggressor. On the other hand, for medium size messages, some congestion builds up before the congestion control algorithm detects and reacts to it, and we observe an increase in the congestion impact up to 1.21. However, as we shown in Figure 9 this is negligible when compared to what happens on other types of systems. Moreover, we observe the highest congestion impact for large bursts and for small gaps between subsequent bursts. We also observe no differences between bursts of $10^6$ messages and the persistent congestion. This shows that SLINGSHOT is tolerant to both persistent congestion, and bursty and short-lived congestion.

### B. Traffic Classes

We now evaluate the ability of SLINGSHOT to provide performance guarantees to jobs running by using traffic classes. It is worth remarking that traffic classes and congestion control are orthogonal concepts. Traffic classes can be used to protect...
a job (or parts of it) from other traffic, and they can allocate resources fairly or unfairly between users and jobs. However, even if resources are assigned fairly, congestion can still occur due to jobs filling up the buffers. Congestion control is used to avoid such situations within and across traffic classes.

All the experiments presented in the following have been executed on MALBEC. We taper the bandwidth to 25% of the available bandwidth, to force co-running jobs to interfere with each other. We execute a job performing an 8B MPI_Allreduce together with a job performing a 256KiB MPI_Alltoall. Each job uses 64 nodes and 16 processes per node, and they are placed using the interleaved allocation. We report in Figure 13 the congestion impact of the MPI_Allreduce when using the same traffic class of the MPI_Alltoall and when using a separate traffic class. Each point represents the mean over 100,000 runs. The MPI_Alltoall is started around 0.4 milliseconds after the beginning of the test. We observe that when MPI_Allreduce runs in the same traffic class of the MPI_Alltoall, it experiences a congestion impact of 2.85 (i.e. is 2.85 times slower compared to when executed in isolation). On the other hand, when executed in a separate traffic class it only experiences a 1.15x slowdown compared to the isolated case.

We now further investigate the capacity of SLINGSHOT to enforce specific limits on traffic classes. We execute two jobs, each running a bisection bandwidth test, with the second one starting after 0.9 milliseconds from the beginning of the test. Each job uses 16 processes per node and runs on 64 nodes. Jobs are placed by using the interleaved allocation. We configure two traffic classes: TC1 with a minimum bandwidth requirement of 80% of the available bandwidth, and TC2, with a minimum 10% bandwidth required.

We report the results of this experiment in Figure 14. On the upper part, we report the results we obtain when both jobs run on the same traffic class (TC1). At the beginning of the execution, the first job runs on an empty system and gets 100% of the available bandwidth. When the second job starts, the available bandwidth is fairly shared between the two jobs. Eventually, when the first job terminates the second job ramps up and uses all the available bandwidth.

On the lower part of Figure 14, we report the results when the first job runs in TC1 and the second job runs in TC2. In this case, when the second job starts, the bandwidth of the first job drops to 80% of the available bandwidth, matching the minimum bandwidth required for TC1. The second job required a minimum bandwidth of 10%, and it gets the 20% of the available bandwidth. Indeed, there is an extra 10% of bandwidth which was not allocated to either TC1 or TC2. SLINGSHOT decides to dynamically allocate this extra bandwidth to TC2 because it is the traffic class with the lowest bandwidth share. Eventually, when the first job terminates, the second job uses all the available bandwidth.

Fig. 13: Congestion impact for an 8B MPI_Allreduce, co-executed with a 256KiB MPI_Alltoall on MALBEC (with a 25% tapering) with and without traffic classes.

Fig. 14: Performance of two bisection bandwidth tests on MALBEC (with a 25% tapering) when running in the same traffic class (top) and when running into two separate traffic classes (bottom).

IV. STATE OF THE ART

A. Interconnection Networks

Existing large-scale computing systems are characterized by different types of interconnection networks, either based on open standards or proprietary technology. These networks have different topologies and provide different features. In this section, we highlight the main characteristics of the most common and actively developed interconnection networks, to better understand the similarities and differences with SLINGSHOT.

InfiniBand is an open standard for high-performance network communications. Different vendors manufacture InfiniBand switches and interfaces, and the InfiniBand standard is not tied to any specific network topology. The most commonly used InfiniBand implementations rely on Mellanox hardware, with switches arranged in a fat tree topology [54]. For example, both Sierra [55] and Summit [17], the two fastest supercomputers at the time being, use such configuration. Mellanox networks also provide other features to improve application performance, such as switch offloading of MPI collective operations, adaptive routing, congestion control, and traffic classes. However, congestion control is usually not used in large production systems due to difficulties in the tuning of
the algorithm [6]. Regarding interoperability with Ethernet, Mellanox adopts a different approach than SLINGSHOT, requiring traffic to be converted between InfiniBand and Ethernet by using dedicated gateways.

Cray ARIES [43] is the 7th generation of Cray interconnection networks. It is based on a Dragonfly topology and supports different systems configuration up to 92 544 nodes (Trinity [56], the largest ARIES system currently deployed, has 19 420 nodes). It provides a peak injection bandwidth of 81.6 Gb/s per node, and a rich set of features including adaptive routing, collective operations offload, and remote atomic operations. It uses fewer optical links than fat trees networks, reducing the cost of the network.

**Tofu Interconnect D (TofuD)** [57] is the third generation Tofu interconnection networks, which will be used by the Fugaku supercomputer [58] (formerly known as Post-K). TofuD provides a peak injection rate of 300Gb/s per node and, like its predecessors, it is based on a 6D mesh/torus. Around 25% of the links used by the interconnect are optical. To reduce latency and improve fault resiliency, TofuD uses a technique called dynamic packet slicing, to split the packets in the data-link layer. This can either be used to split the packet and improve the transmission performance or to duplicate the packet to provide fault tolerance in case the link quality degrades. Moreover, this interconnect provides an offload engine, called Tofu Barrier, to execute collective operations without involving the CPU.

The **Dragonfly+** [59] is currently used by the Niagara supercomputer [60]. It is a variation of the Dragonfly interconnect [12], where the switches inside a group are connected through a fat-tree network. Similarly to the Dragonfly network, this interconnect is characterized by different minimal and non-minimal paths between each pair of nodes. The implementation used in the Niagara supercomputer relies on Mellanox InfiniBand hardware. To select the optimal path, Dragonfly+ uses a variation of the OFAR adaptive routing [61], which at each hop re-evaluates the optimal path to use. Explicit control messages are sent among the switches to notify congestion and avoid creating hotspots in the network.

Several other low-diameter networks [62] have been proposed by the research community, including but not limited to SlimFly [63], Megafly [64], HyperX [65], [66], Jellyfish [67] and Xpander [68] topologies. On the data centers side, Clos [69] is the most prevalent deployed topology. Whereas the above mentioned low-diameter topologies are claiming to have substantial cost-performance improvements, they have been scarcely employed because of hard-to-deploy routing schemes. Also, classical congestion control mechanisms (e.g., ECMP [70]) are not effective in such low-diameter networks due to the scarcity of minimal paths [18]. SLINGSHOT addresses these issues by providing a low-diameter network with an effective congestion control algorithm, setting a stepping stone towards HPC data centers.

Overall, SLINGSHOT introduces a set of key features that can be taken as reference for next-generation large-scale computing systems. First, the end-to-end congestion control algorithm can quickly react to congestion and is stable across a wide set of applications and microbenchmarks. Moreover, traffic classes provide additional flexibility and open new software optimization opportunities. Lastly, it is natively interoperable with existing Ethernet devices, and thanks to novel adaptive routing strategies, it provides high network utilization also for in-order RoCE traffic (see Figure 6).

### B. Interconnection Networks Benchmarking

In this work we described the SLINGSHOT interconnection network and, for the first time, we extensively evaluated it across a wide set of microbenchmarks and real applications. We reported both the isolated performance and the performance under the presence of congestion.

Regarding the evaluation of the under-load system, different works analyzed the impact of congestion (also known as network noise) on application performance [5], [6], [11], [71]–[74] on different types of networks. The GPCNet benchmark [6] has been recently proposed as a portable benchmark for estimating network congestion. We used in this work the same definition of endpoint/intermediate congestion and of congestion impact used by GPCNet. Whereas the authors of GPCNet also report some preliminary results on a SLINGSHOT system, they do not provide a detailed view of the system performance. Indeed, the main goal of GPCNet was to design a portable congestion benchmarking infrastructure by using a small set of victim microbenchmarks (random ring and MPI_Allreduce) to easily compare different systems. However, this does not represent a wide spectrum of real scenarios. On the other hand, we focus on the impact of congestion on SLINGSHOT by using different microbenchmarks and both on HPC and datacenters applications. Moreover, the GPCNet paper only analyzes the impact of congestion for a fixed victim message size, allocation, and aggressor/victim ratio. However, as we show in Section III-A, all these factors play a role in the observed congestion and they can be helpful to understand the system performance.

### V. Conclusions

Interconnection networks have a significant impact on the performance of large computing systems, both in supercomputers and hyperscale datacenters. In this paper, we describe and evaluate SLINGSHOT, the latest interconnection network designed by Cray. We describe SLINGSHOT’s main features: high-radix Ethernet switches, adaptive routing, congestion control, and QoS management. We then evaluate SLINGSHOT’s performance, both in isolation and when executing different concurrent workloads.

Our results demonstrate that applications running on SLINGSHOT are much less affected by congestion compared to previous generation networks and that the congestion control algorithm works on a wide set of different microbenchmarks and HPC and datacenter applications. We also show that allocation policies have a much lower impact on performance on SLINGSHOT compared to previous generation networks.
Lastly, we demonstrate how SLINGSHOT can provide bandwidth guarantees to jobs running in separate traffic classes. The information we provide can be used by HPC and datacenter system operators, administrators, users, and programmers to optimize, deploy, and manage parallel applications. A deep understanding of the interconnect’s features is a prerequisite to ensure optimized operations and utilization of computing resources in clouds and datacenters.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments, and the Slingshot team at HPE for providing access and support in using the systems. We thank Steve Scott (HPE) for invaluable input. We would also like to thank Shigang Li for providing the code for the Resnet-proxy application. Daniele De Sensi is supported by an ETH Postdoctoral Fellowship (19-2 FEL-50). This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 programme (grant agreement DAPP No. 678880).

REFERENCES

