Mitigating Inter-Job Interference via Process-Level Quality-of-Service

Lee Savoie and David K. Lowenthal
Deptartment of Computer Science, The University of Arizona

Bronis R. de Supinski and Kathryn Mohror
Lawrence Livermore National Laboratory

Nikhil Jain
Nvidia Corporation

***Abstract*—Jobs on most high-performance computing (HPC) systems share the network with other concurrently executing jobs. This sharing creates contention that can severely degrade performance. We investigate the use of Quality of Service (QoS) mechanisms to reduce the negative impacts of network contention. Our results show that careful use of QoS reduces the impact of contention for specific jobs, resulting in up to a 27% performance improvement. In some cases the impact of contention is completely eliminated. These improvements are achieved with limited negative impact to other jobs; any job that experiences performance loss typically degrades less than 5%, often much less. Our approach can help ensure that HPC machines maintain high throughput as per-node compute power continues to increase faster than network bandwidth.**

***Index Terms*—network contention, quality of service, MPI**

I. Introduction

High performance computing (HPC) systems use thousands of nodes to execute multiple jobs concurrently. HPC installations use *space-shared* scheduling in which each node is assigned to at most one job at any time. This strategy prevents contention for per-node resources such as cores or memory. However, on most HPC systems, the interconnect is shared among all jobs and can become a severely contended resource, regardless of the topology, including fat trees [1], dragonfly networks [1], [2], [3], and tori [4]. Network contention increases job communication time, which degrades performance.

We explore using Quality of Service (QoS) to manage network contention. Many modern networks provide QoS capabilities, including popular HPC network fabrics such as InfiniBand. Although QoS has been used to prioritize traffic in other contexts [5], [6], [7], [8], it is not commonly used on HPC systems. By intelligently allocating bandwidth between jobs to reduce network contention, QoS could improve job performance and, thus, HPC system throughput.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-787578). In addition, this material is based upon work supported by the National Science Foundation under Grant No. 1526015.

This paper presents the design, implementation, and evaluation of MPI process-based QoS prioritization to reduce contention on fat-tree networks. We use an iterative, feedback-directed approach that takes input from all concurrent jobs and prioritizes MPI processes to improve performance. The algorithm generally ensures that processes that receive an increase in priority have a limited impact on the performance of other processes from the same or other jobs. Our algorithm only requires knowledge of per-process times per job timestep, which are easily determined if the application identifies the timesteps. With this assumption, our algorithm can easily be integrated into a global run-time layer.

We have implemented our system, called TraceR-QoS, using the TraceR network simulator [9]. Our results show that it reduces or eliminates the impact of contention for several different workloads and job placements. This improvement has relatively little cost to other concurrently executing jobs.

The contributions of this paper are as follows:

- A demonstration that QoS can selectively prioritize MPI processes to improve performance of individual jobs;
- An algorithm that transparently and effectively applies QoS with little application information; and
- Its open-source implementation, TraceR-QoS.

In some cases, TraceR-QoS completely eliminates the impact of network contention on a job (i.e., that job executes as if it were completely isolated). Further, TraceR-QoS improves job performance up to 27% and rarely degrades the performance of any other job more than 5%.

This paper is organized as follows. Section II provides background information on network contention and QoS, while Section III describes our per-process service level assignment algorithm. Section IV details our experimental setup, and Section V presents our results.

II. Overview

Network contention occurs when multiple network flows concurrently use the same network link. This sharing causes the length of the switch output queue that connects to the link to grow. Thus, packets have longer queue wait times and increased latency. In addition, all flows that share the link

experience reduced bandwidth. This slower message delivery increases job execution time and reduces system throughput.

Contention effects can propagate across a job. For example, process P_1, which does not experience contention, may be slowed because it waits for a message from process P_2 that does, even with no contention between P_1 and P_2. Further, if contention causes a message to process P to arrive later than it otherwise would, P is likely to send subsequent messages later, even if they use different links, which will delay the processes that receive those messages. These perturbations may propagate across the system and manifest as second and higher order effects that are difficult to predict.

Most modern networks provide Quality of Service (QoS) mechanisms to manage their traffic. QoS has many forms; in this paper we focus on traffic prioritization by assigning relative priorities to different network flows. The network forwards packets based on the priorities of their flows, giving more bandwidth to those with higher priority.

Our approach applies to any network with traffic prioritization. We focus on InfiniBand, which implements QoS with *service levels*. Each packet is assigned to a service level, which corresponds to a priority. InfiniBand nodes and switches send packets in a weighted round-robin order in which the weights correspond to the service level priorities [10].

III. Algorithm

We define a job as an MPI application execution; each job consists of many MPI processes. We strive to minimize the information required from the executing jobs. We assume that we can measure the amount of work that each process performs in each time period, for example, by requiring each job to send a signal to the run time system when it completes each iteration. We also assume that each process can send packets from any service level and that we can change service level priorities at run time. While most systems do not currently provide these capabilities, they are easily supported. We can change the service level of a packet by changing a few bits in its header. Changing the priority of a service level is more complicated since we must update the priority table at every node and switch in the network. However, the InfiniBand subnet manager already must support these operations. Our algorithm just uses them more frequently.

In Algorithm 1, we initially run all jobs at the same service level (the default configuration). We record the default performance, which is the amount of work that each job completes (line 7). Subsequent iterations adjust service levels by increasing the priority of a candidate; candidates are all processes within a threshold (T) of the slowest process. If the selected process is in the default service level, we put it in a separate, unused service level (line 32). Otherwise, we increase the priority of its current service level (line 30).

We then run the jobs with this service level change. The resulting performance guides the algorithm's next step. If the service level improves performance relative to the current best service level assignments (line 16), the current set becomes the new best and we choose new candidates. Otherwise, we

Algorithm 1 Process Prioritization Algorithm

```
 1: function PROCESS_PRIORITIZATION_ALGORITHM
 2:     state ← DEFAULT
 3:     assignments ← no processes prioritized
 4:     repeat
 5:         SetServiceLevels(assignments)
 6:         Allow the jobs to run for P seconds
 7:         workDone ← work completed by all jobs
 8:         if state = DEFAULT then
 9:             default.workDone ← workDone
10:             best.workDone ← workDone
11:             best.assignments ← assignments
12:             candidates ← SelectCandidates()
13:             state ← RUNNING
14:         else if state = RUNNING then
15:             if workDone > best.workDone then
16:                 best.workDone ← workDone
17:                 best.assignments ← assignments
18:                 candidates ← SelectCandidates()
19:             if |assignments| = NUM_SLS  or
20:               |candidates| = 0 then
21:                 state ← IDLE
22:         else if state = IDLE then
23:             if workDone < default.workDone then
24:                 state ← DEFAULT
25:         if state = DEFAULT then
26:             assignments ← no processes prioritized
27:         else if state = RUNNING then
28:             proc ← candidates.pop()
29:             if proc in assignments then
30:                 increase assignments[proc].priority
31:             else
32:                 assignments[proc] ← new service level
33:         else if state = IDLE then
34:             assignments ← best.assignments
35:     until the job mix changes
36: function SELECTCANDIDATES
37:     candidates ← ∅
38:     for all jobs do
39:         worst ← min(job.procs.workDone) ∗ (1/T)
40:         for all procs in job do
41:             if proc.workDone < thresh then
42:                 candidates ← candidates ∪ {proc}
        return candidates
```

prioritize another candidate. The algorithm iterates until either no service levels or no candidates remain (line 19). We then use the best set of service levels until a job arrives or departs (line 35), at which point the algorithm restarts.

IV. Experimental Setup

For our experiments, we use a simulator (rather than a real system) for four reasons. First, we can use per-process QoS, which is not implemented on any HPC cluster to which we

Machine	Nodes	Levels	Link B/W	Num. Jobs	Processes/Job
small	64	2	3.2	8	8
medium	324	2	3.2	16	20

TABLE I: Experimental systems; *Levels* is the height of the fat tree; bandwidth is in GB/s.

have access. Second, we can run many tests on systems of varying size. Third, the results are repeatable. Fourth, we can more easily investigate the low-level effects of QoS.

We use TraceR [9], which is an HPC application trace replay tool. It is built on the CODES framework [11], which is built on the ROSS parallel discrete event simulator [12]. TraceR performs packet-level simulation of MPI applications on HPC networks. It efficiently and accurately predicts the performance of important applications on networks with different properties as previous work [13] has extensively validated.

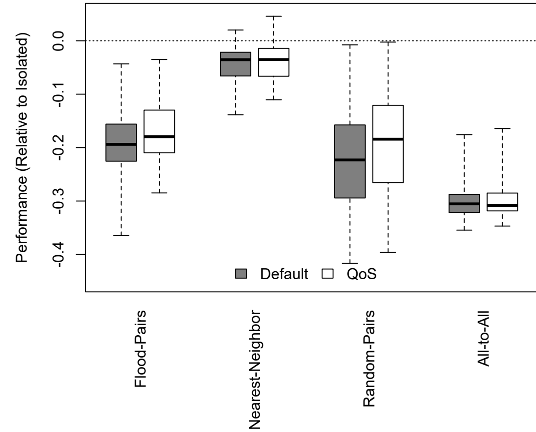
For our experiments, we add QoS to TraceR. For clarity, we refer to the baseline TraceR implementation as TraceR while TraceR-QoS represents our modifications to TraceR and CODES along with additional scripts that implement our algorithm. Our modifications are publicly available and have been submitted for inclusion in their respective projects. Our updates to TraceR are available at https://github.com/LLNL/TraceR/pull/23, our updates to CODES are available at https://xgitlab.cels.anl.gov/codes/codes/merge_requests/81, and the remaining scripts are available at https://bitbucket.org/lesavoie/cluster-2019-mitigating-inter-job-interference-via-process.

Table I describes the two simulated systems with fat-tree topologies with which we test our algorithm. We denote these systems as *small* and *medium* in this paper, as they represent a small- and medium-scale cluster, respectively. We base the link bandwidth for both systems on the measured bandwidth of actual systems. Table I also includes the number of jobs and the number of processes per job that we use on each system.

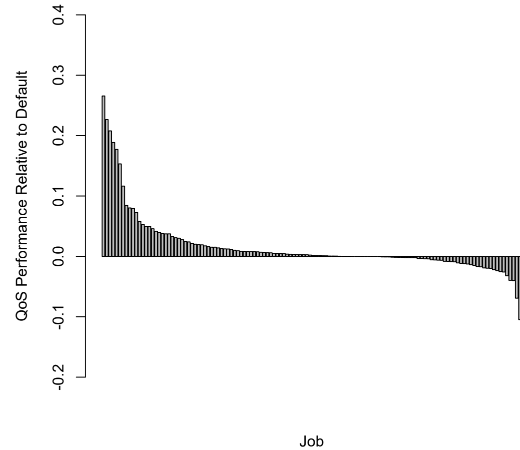
We collected traces on Catalyst, a system that has 300 compute nodes, each with two 12-core Intel Xeon E5-2695 CPUs and 128 GB of RAM. We use ScoreP [14] to collect traces in the OTF2 format [15]. We ran all tests on eight different randomly chosen node allocations, or assignments of processes to nodes, which approximates typical space-shared HPC systems in the steady state. This choice also allows us to test our algorithm with different contention characteristics.

We run three tests for every iteration of our algorithm. First, we run each job in isolation to measure un-contended performance. Second, we run all jobs together with a single service level, which is our default. Third, we use the service levels that our algorithm selects. We use the same traces for all three runs so we can accurately compare the performance of our algorithm to default and isolated runs.

We test our algorithm with four microbenchmarks that implement common communication patterns. These benchmarks can be configured with different message sizes, computation amounts, and computation imbalances. We call these benchmarks *Flood-Pairs*, *Nearest-Neighbor*, *Random-Pairs*, and *All-to-all*. *Flood-Pairs* is based on the CORAL bisection bandwidth test [16]. *Nearest-Neighbor* is a standard 2D nearest neighbor communication pattern without wraparound. *Random-Pairs* is similar to *Flood-Pairs*, except pairs are random. Finally, in *All-to-all*, each process repeatedly exchanges messages with every other process.



(a) Overall results



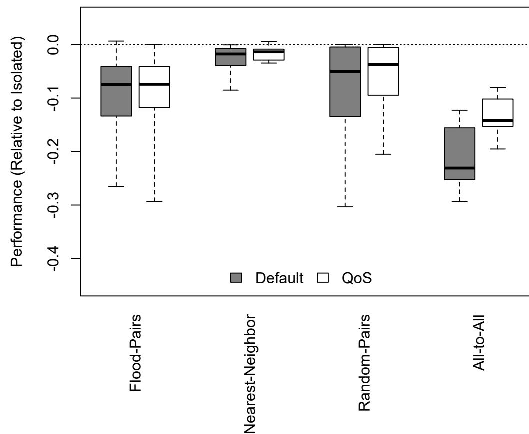
(b) Individual job performance

Fig. 1: Results on the *Medium* System (324 Nodes)

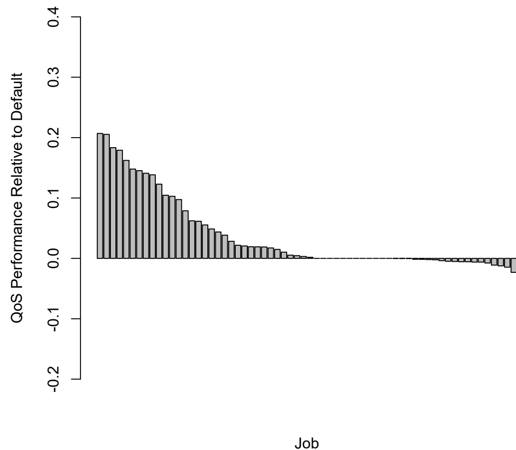
V. Results

Figures 1 and 2 show the results on the *medium* and *small* systems. On *medium* we run four instances of each microbenchmark, and on *small* we run two instances of each microbenchmark. We introduce some computational imbalance in these jobs; the computation time for each process was chosen randomly between 0 and Y, and Y was chosen such that the process that computes the longest spends roughly 65% of its time in computation. For a given process, the amount of computation is the same for an entire run. We set T to 0.9.

These results include both the running and idle time of our algorithm, so they accurately depict its benefits and overheads. We include two charts for each set of results. The first chart shows the overall results for each job type as a box plot; we include results with all jobs in the same service level (labeled



(a) Overall results



(b) Individual job performance

Fig. 2: Results on the *Small* System (64 Nodes)

Default) and with our algorithm (labeled *QoS*). The x-axis lists the microbenchmark and the y-axis shows the performance of each job relative to the same job run in isolation (negative numbers indicate that the job completed less work than it did on the isolated run, the most likely scenario). We mark $y = 0$, which indicates that a job achieved the same performance as it did on the isolated run, with a dashed line. We combine multiple instances of the same microbenchmark because they are identical. Thus, each box plot shows the results from several jobs across several placements. The second chart shows the improvement that our algorithm achieves over the default performance for individual jobs, sorted from the greatest improvement to the greatest degradation.

Our algorithm improves performance on the *medium* system for more than half of the jobs, while incurring relatively small slowdowns in the other jobs. The largest single-job performance improvement over default is 27%, and many jobs have improvements over 10%. All single-job performance degradations are 4% or less, with the exception of two jobs with 7% and 10% degradation. Further, our algorithm reduces the impact of contention by up to 77% (for one of the *Flood-Pairs* jobs) and completely eliminates it (for some of the *Nearest-Neighbor* jobs). The results on the *small* system are similar; performance improves for most jobs—over 20% in two cases. Several jobs had at least 10% improvement, with the worst case only 4% degradation. As with *medium*, in some cases our algorithm eliminates the impact of contention.

On *medium*, some *Nearest-Neighbor* jobs ran faster in default and QoS cases than in isolation despite incurring contention with other jobs. Although we expect performance to be worse, in this case contention with other jobs reduces the impact of contention within a job. Suppose processes P_1 and P_2 are part of the same job and share a link, so they contend with each other. If P_2 experiences contention from another job, P_1 can send more packets and complete its message faster. If P_1 is on the critical path and P_2 is not, the performance of the job will improve. This situation occurs when the default or QoS case is faster than the isolated case.

VI. Related Work

QoS research has a long history. The internet protocol (IP) implements differentiated services [17], in which flows are divided into classes that are handled differently. QoS has also been applied in areas as diverse as wireless networking [5], data centers [6], and video streaming [7], [8].

QoS use in HPC networks is limited. Others have used QoS to prioritize traffic at the job level [18], [19] in simulated systems. We previously presented the first empirical investigation of QoS and its impact on network contention on a real HPC system [20], which showed a reduction in overall throughput, unlike the simulated approaches. We found that assigning a high priority to an entire job prioritizes some processes unnecessarily, increasing the impact of network contention on other jobs. Thus, we explore per-process QoS in this paper.

Researchers have considered several other methods to reduce network contention on fat trees [1], dragonfly networks [1], [2], [3], and tori [4]. Adaptive routing [21], [22] seeks to route traffic away from congested links. However, adaptive routing does not necessarily reduce contention significantly [1]. Researchers have also studied job placement strategies that minimize contention [3], [23], [24], but these approaches typically reduce machine utilization.

VII. Summary

Node processing power improvements will continue to outpace network bandwidth improvements in HPC systems. Thus, network performance and network contention will become more important. We explored how QoS mechanisms can mitigate the impacts of network contention. We introduced an algorithm that uses QoS to improve the performance of individual jobs by up to 27%, usually without degrading the performance of other jobs by more than 5%. Our algorithm sometimes eliminates the impact of contention, especially for jobs that have imbalanced computation. Future HPC systems must employ network management techniques such as those that we introduced as network performance becomes a larger component of overall system performance.

References

[1] S. Smith, C. Cromey, D. K. Lowenthal, J. Domke, N. Jain, and A. Bhatele, "Mitigating inter-job interference using adaptive flow-aware routing," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. IEEE Computer Society, Nov. 2018.

[2] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on Xeon Phi based Cray XC systems," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, USA, November 12-17 2017.

[3] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! Job interference study on dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, 2016.

[4] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013.

[5] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar, "Providing quality of service over a shared wireless link," *Comm. Mag.*, vol. 39, no. 2, pp. 150–154, Feb. 2001. [Online]. Available: http://dx.doi.org/10.1109/35.900644

[6] T. Voith, K. Oberle, and M. Stein, "Quality of service provisioning for distributed data center inter-connectivity enabled by network virtualization," *Future Generation Computer Systems*, vol. 28, no. 3, pp. 554 – 562, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11000392

[7] C.-H. Ke, C.-K. Shieh, W.-S. Hwang, and A. Ziviani, "A two markers system for improved MPEG video delivery in a diffserv network," *IEEE Communications Letters*, vol. 9, no. 4, pp. 381–383, April 2005.

[8] W. Kumwilaisak, Y. T. Hou, Q. Zhang, W. Zhu, C. C. J. Kuo, and Y.-Q. Zhang, "A cross-layer quality-of-service mapping architecture for video delivery in wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 10, pp. 1685–1698, Dec 2003.

[9] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating HPC networks via simulation of parallel workloads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 14:1–14:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014923

[10] S. A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken, "An overview of QoS capabilities in Infiniband, advanced switching interconnect, and Ethernet," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 32–38, July 2006.

[11] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, "CODES: Enabling co-design of multilayer exascale storage architectures," in *Proceedings of the Workshop on Emerging Supercomputing Technologies*, May 2011.

[12] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low memory, modular time warp system," in *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, May 2000.

[13] N. Jain, A. Bhatele, L. H. Howell, D. Böhme, I. Karlin, E. A. León, M. Mubarak, N. Wolfe, T. Gamblin, and M. L. Leininger, "Predicting the performance impact of different fat-tree configurations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017.

[14] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[15] ParaTools, Inc. (2019) Open Trace Format. [Online]. Available: http://www.paratools.com/otf/

[16] (2014, Jun.) Coral benchmark codes. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[17] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated service," United States, 1998.

[18] A. Jokanovic, J. C. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg, "Effective quality-of-service policy for capacity high-performance computing systems," in *International Conference on High Performance Computing and Communication and International Conference on Embedded Software and Systems (HPCC-ICESS)*, June 2012, pp. 598–607.

[19] M. Mubarak, N. McGlohon, M. Musleh, E. Borch, R. Ross, R. Huggahalli, S. Chunduri, S. Parker, K. Kalyan, and C. Carothers, "Evaluating quality of service traffic classes on the megafly network," in *International Supercomputer Conference*, Jun. 2019.

[20] L. Savoie, D. K. Lowenthal, B. R. de Supinski, and K. Mohror, "A study of network quality of service in many-core MPI applications," in *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, 8 2018, pp. 1313–1322.

[21] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kale, "Maximizing throughput on a dragonfly network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 336–347. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.33

[22] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. H. Oral, D. E. Maxwell, V. G. V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems, and J. Yin, "The design, deployment, and evaluation of the CORAL pre-exascale systems," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, November 11-16 2018.

[23] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: Key to protect job performance predictability," in *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015, pp. 449–459.

[24] E. Zahavi, A. Shpiner, O. Rottenstreich, and I. Keslassy, "Links as a service (LaaS): Guaranteed tenant isolation in the shared cloud," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, March 2016.