Prediction of the Impact of Network Switch Utilization on Application Performance via Active Measurement☆

Marc Casas[a,*], Greg Bronevetsky[b]

[a] *Barcelona Supercomputing Center, Universitat Politècnica de Catalunya*
[b] *Google Corporation*

Abstract

Although one of the key characteristics of High Performance Computing (HPC) infrastructures are their fast interconnecting networks, the increasingly large computational capacity of HPC nodes and the subsequent growth of data exchanges between them constitute a potential performance bottleneck. To achieve high performance in parallel executions despite network limitations, application developers require tools to measure their codes' network utilization and to correlate the network's communication capacity with the performance of their applications.

This paper presents a new methodology to measure and understand network behavior. The approach is based in two different techniques that inject extra network communication. The first technique aims to measure the fraction of the network that is utilized by a software component (an application or an individual task) to determine the existence and severity of network contention. The second injects large amounts of network traffic to study how applications behave on less capable or fully utilized networks. The measurements obtained by these techniques are combined to predict the performance slowdown suffered by a particular software component when it shares the network with others. Predictions are obtained by considering several training sets that use raw data from the two measurement techniques. The sensitivity of the training set size is evaluated by considering 12 different scenarios. Our results find the optimum training set size to be around 200 training points. When optimal data sets are used, the proposed methodology provides predictions with an average error of 9.6% considering 36 scenarios.

☆With the support of the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Expedient 2013 BP_B 00243). The research leading to these results has received funding from the European Research Council under the European Union's 7th FP (FP/2007-2013) / ERC GA n. 321253. Work partially supported by the Spanish Ministry of Science and Innovation (TIN2012-34557).

*Corresponding author

Keywords: Performance Modeling, Resource Sharing, Measurement Techniques

1. Introduction

HPC applications demand very capable communication networks to support their high message and data volumes and/or tight synchronizations. Indeed, constraints on available network bandwidth or latency as well as network hotspots induced by specific communication patterns are often the key bottleneck that limit application performance [37, 2, 42, 16, 4, 17]. Looking into the future, it is expected that the computational capabilities of individual computing nodes will continue to rise faster than the capabilities of the networks that connect them [27]. This means that application performance will become increasingly bottlenecked on the capabilities of the network, making it even more imperative for application developers to optimize their applications taking network performance into account. Specifically, developers will need to (i) predict how their applications will perform on future systems with poorer network-to-node performance ratios and (ii) develop ways to assign computing work to available resources to effectively balance network communication and on-node computation. To achieve these tasks developers will require powerful tools to enable them to understand the interactions between their applications and the networks they run on and how these interactions ultimately affect application performance. Specifically, two directions of this interaction will need to be quantified for developers. First, tools must quantify how the application's communication utilizes the network and whether the application's needs are approaching the limits of the network's capabilities. Second, tools must measure how the capabilities of the network influence application performance and most importantly, whether the network is the application's performance bottleneck. These analyses must apply to both current and future systems, as well as to both static and highly configurable applications (e.g. where the space of possible configurations is too large to be explicitly enumerated and analyzed).

This paper eluates a new approach to measure the relationship between network capacity and application performance [9]. Our basic insight is that this relationship should be modeled as the application consuming a resource provided by the network. As more of this resource is available, the application runs monotonically faster, with reduced improvements as application performance becomes bottlenecked on other resources. Further, if multiple software components (entire applications or individual tasks such as processes, threads or Charm++ chares [21])) run concurrently on the same network, they will share its resources. This sharing can be modeled as one component consuming some amount of network resources, making it unavailable to others and thus causing them to behave as if they were running on a less capable network. The heart of this idea is a "performance relativity" principle, that "from the perspective of software components less capable networks behave very similarly to networks that are partially utilized by other software components".

When several software components share a network there can be many more potential issues than when just a single software component runs on a dedicated but slow network. These issues have been measured in detail [22, 5, 20] concluding that interference strongly contributes to performance degradation. Since the slowdown suffered by each particular software component can be emulated by running each one of these components in a dedicated and slow network, the “performance relativity” principle holds even though the plethora of issues that can be raised by network interference. This principle enables two novel measurement techniques that can answer the above questions:

Impact experiments measure a software component’s use of the network based on the latency of a few additional packets sent over the network while the component runs. These measurements directly quantify the network’s ability to carry application communication and can be used to determine whether the network is congested and measure how close the application is to fully utilizing the network. The additional packets are triggered by extra tasks running on dedicated cores and they do not impact applications’ performance as the extra load is very low. We presented this idea in a previous paper [9] and some subsequent similar ideas have been proposed [15].

Compression experiments measure the relationship between network capability and a software component’s performance. The component is executed concurrently with a micro-benchmark that runs on cores connected to the same network and sends varying amounts of communication. As the effective network capability is varied we observe the component’s resulting performance, which corresponds to how it will perform on less capable networks or when more software components are executed on the same network. The idea of inserting messages and study its subsequent performance impact at parallel applications has been previously explored [36].

Finally we present several techniques that combine the two measurements to predict the performance degradation that a given combination of software components would suffer when executed concurrently on the same network. Each technique is based on a particular description of the available network capability when an application is running. Data from Impact measurements is used to compute latencies of the triggered packets. We consider four different approaches to describe the available network capacity when a particular parallel application is running: i) The average latency of all the packets triggered by the application ii) The average latency and the standard deviation of the packets triggered by the application iii) the histogram of the latencies of the packets triggered and iv) a mathematical queue [40]. By measuring the network capability that is left available while a given application or the Compression benchmark runs we can estimate the effect of multiple concurrent software components on each other as they share a network. The experimental and analytic procedures presented in this paper are focused on single-switch networks that connect multiple computing nodes.

This paper extends our previous work [10, 9] by building performance models from multiple data sets, that is, by providing a complete evaluation of the performance prediction sensitivity with respect to the training set size. Our

approach improves upon the state of the art in network performance modeling and measurement in at least 4 ways. The first 3 were already presented [10, 9] and are revisited in this paper, while the fourth one is completely new:

i) Impact experiments of network utilization and contention are significantly faster than similar analysis performed inside simulators and apply to real physical networks for which precise models may not exist due to intellectual property restrictions. Further, unlike indirect measurement techniques, Impact experiments directly probe the network's ability to carry out the application's communication requests. Since they focus on just the network and quantify its effective capabilities in terms of a generic queue-oriented metric, these experiments provide a simple and unfiltered view onto this resource.

ii) Compression experiments and Performance Degradation analysis make it possible to relate application performance to network capacity. While simulators can predict the performance of specific workloads on specific networks, a separate simulation run is required for each configuration. As the number of configuration options increases (e.g. number of atoms per core or the assignment of software components to different cores), the number of such experiments rises exponentially. In contrast, our approach scales linearly with the number of software components that must be measured independently.

iii) Our techniques are enabled by a new metric for measuring network utilization, the *switch utilization metric*. This metric has the key property of describing the utilization of the physical network. Combined with the Compression experiment, the switch utilization metric is connected to application performance.

iv) The sensitivity of application performance predictions to training set sizes is evaluated by considering from very restricted up to very large training set sizes. The considered sizes vary from 30 up to 408 points per set, covering thus a wide range of prediction scenarios.

This paper is structured as follows: Section 2 presents the experimental setup used and briefly describes the set of applications we use in our experiments. Section 3 describes Impact and Compression measurements and how they interact. Section 4 presents the models used to predict applications' slowdown. Section 5 presents and validates our methodology for predicting performance of real complex workloads that share the same network switch. Section 6 shows a complete evaluation of the predicting models considering different training set sizes.

2. Experimental design

The experiments in this paper were conducted on the Cab cluster at the Lawrence Livermore National Laboratory. Cab is composed of 1,296 compute nodes, each of which includes two 8-core 2.6GHz Intel Xeon E5-2670 processors with 32GB of RAM. The network is QLogic Quad-data-rate organized into a two-level fat tree. This paper focuses on the bottom-level switches of the network, which are QLogic 12300, with 36 ports, of which 18 are used to connect compute nodes and 18 more connect to the second-level switch. These

switches provide approximately 1μs of network latency and 5GB/s bandwidth per port. Each experiment on Cab was run on groups of 18 nodes, respectively, connected to a single bottom-level switch and our results are thus not affected by interference from other applications running on the same cluster.

Our experiments focus on the following applications:

- AMG [13] - An implementation of the Algebraic Multi Grid Solver by using the Hypre library applied to the 3D Laplace problem on a 160x160x160 cube domain.
- FFTW [14] - Fast Fourier Transform library that uses hierarchical composition of multiple FFT algorithms, applied to perform a 2D transform of a 2000x2000 matrix.
- Lulesh [1] - The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics simulation that is a materials science proxy application, executed on a 22x22x22 cube domain.
- MCB [11] - A continuous energy Monte Carlo Burn-up Simulation Code for studying nuclear waste transmutation systems, executed on 3,000,000 particles.
- MILC [3] - The MIMD Lattice Computation, a Quantum Chromodynamics simulation with lattice size nx=16, ny=32, nz=32, nt=36.
- VPFFT [30] -A structure sensitive crystal plasticity simulation code applied to a 4x4x4 sample grid considering 1000 time steps, 0.03 time units per step and a convergence threshold of 1e-5.

The set of applications is representative of the typical workloads that run in HPC infrastructures. AMG carries out several iterations of an iterative solver over the same linear system at different levels of granularity. It behaves like a CPU intensive benchmark when operating over a dense representation of the system and like a communication and memory bound application when the representation of the system is sparse. Thus, AMG runs will display very different phases. FFTW and VPFFT applications contain expensive all-to-all communications. The difference between these two applications is that VPFFT performs expensive computation between two communication phases while FFTW does not. As such, VPFFT has some flexibility to overlap communication and computation while FFTW is not that flexible. Lulesh is a typical finite difference method code with local communication phases interleaved by intensive computation phases. MCB is a monte carlo simulation code, which means that it does not have much communication and, therefore, its usage of the interconnecting network is expected to be low. Finally, MILC spends most of its time running the conjugate gradient solver, which means that most of its communications involve point to point communications with the neighbors and global reductions once in a while.

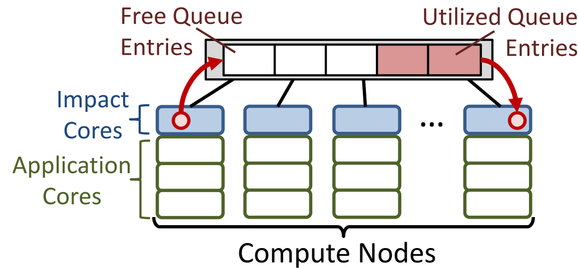

Figure 1: Impact Interference

3. Active Measurement

Our active measurement methodology adds extra-load into the network and measures some performance metrics provided, directly or indirectly, by this extra load. We follow two main approaches: The first one aims to inject a very light extra traffic into the network with the aim of not impacting the performance of the running application. By directly measuring the latency of the extra packets we inject, we infer the distribution of the latencies of the packets triggered by the main application, which would be very hard to measure without injecting the extra traffic. The second approach aims to inject heavy traffic into the network and measure, for each degree of interference, the performance degradation suffered by the main application. We explain the details of these two approaches in this section.

3.1. Impact

The basic idea behind Impact experiments is that the degree to which an application utilizes a network switch can be measured in terms of how well the network can service additional communication requests. Application messages are broken up into multiple small (few KB) packets and sent to the network switch. As illustrated in Figure 1, packets from one compute node arrive on one port of the switch, propagate through its internal circuitry and exit via the port of its destination node. Since the execution time of communication operations depends on the transit time of each packet, the distribution of these times captures the network's effective capability that is available to applications. Further, when some software component is already utilizing the network, the difference between this distribution during the component's execution and the same on an unloaded network measures the amount of network capability the component uses up and leaves unavailable to others.

We measure the latency of packets through the network switch using the simple micro-benchmark listed in Figure 2, which we denote ImpactB. Compute nodes on the same switch are paired and execute a ping-pong data exchange where the process with the even rank sends a message, the process with the odd rank receives it and replies with another, which is finally received by the initial process. The entire exchange is timed by the initiator process to determine the average latency of the two messages, which are set to be 1KB in size to

```
while(1) {
  if( my_node%2 == 0 && my_node!=n_nodes-1 ) {
        MPI_Isend(... , (my_rank+tasks_per_node)%(n_nodes), ... , &request);
        MPI_Irecv(... , (my_rank+tasks_per_node)%(n_nodes), ... , &request2);
  } else if  ( my_node%2 == 1 ) {
        MPI_Irecv(... , my_rank-tasks_per_node, ... , &request);
        MPI_Isend(... , my_rank-tasks_per_node, ... , &request2);
   }
   MPI_Wait(&request, status);
   MPI_Wait(&request2, status);
   usleep(100000);
}
```

Figure 2: Pseudo-code of the ImpactB micro-benchmark

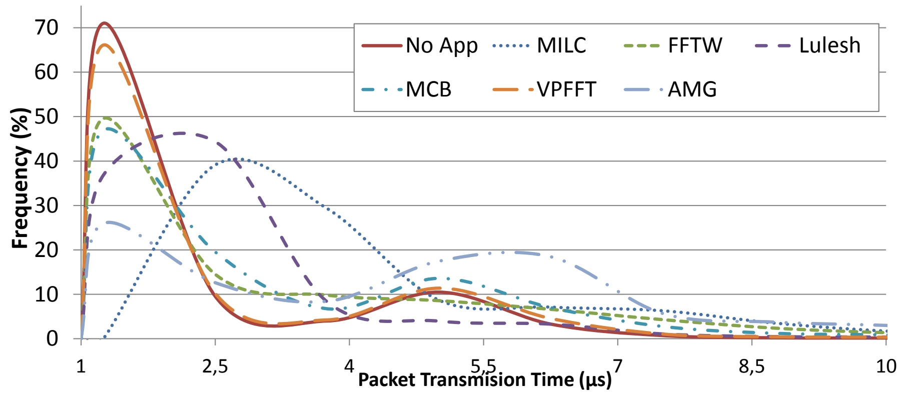

Figure 3: Distributions of Packet Latencies on Cab

ensure that they are communicated via a single network packet. Each ping-pong exchange is separated by a 100ms (i. e. $2.6 \cdot 10^8$ CPU cycles in our experimental setup) sleep to minimize ImpactB's effect on the executing application.

Figure 3 shows the distribution of message latencies observed on Cab when executing ImpactB on an unloaded switch and when ImpactB is executed concurrently with our target applications. In these experiments the processes of ImpactB and the target application were spread over all the compute nodes connected to the switch. In our experiments, 2 ImpactB processes were executed on every node. Since Cab's nodes have 2 sockets, an ImpactB process was run on each socket.

The application processes were executed on the remaining cores. We executed 4 processes of MILC, FFTW, MCB, VPFFT and AMG on each socket, 8 per node for a total of 144 across all the 18 nodes connected to a switch. Lulesh, which needs a cubic number of processes, was run on 16 nodes, utilizing 2 cores on each socket, for a total of 64 MPI processes.

The remaining cores were left idle in these experiments. This assignment of application processes to cores was used to simplify the presentation of the

```
while(1) {
  for(partner=0; partner<P; partner++) {
    for(mesg=0; mesg<M; mesg++) {
      // Receive from same core ID on succeeding node
      MPI_Irecv( ... , (my_rank+tasks_per_node*(partner+1))%comm_size , ... );
      // Send to same core ID on the preceding node
      MPI_Isend( ... , (my_rank-tasks_per_node*(partner+1)+comm_size)%comm_size , );
     }
     usleep(B);
  }
  MPI_Waitall( ... );
}
```

Figure 4: Pseudo-code of the CompressionB interference micro-benchmark

performance prediction experiments in Section 5, which discusses performance prediction for multiple concurrently-executing applications.

The data shows that when the switch is not loaded, packet latency is 1.25μs on average, with many packets taking a little less or more time and a few packets taking significantly longer. When the applications are running the latency distribution shifts. The execution of FFTW and MCB on Cab shifts 20% of packets from taking approximately 1.25μs to take more than 2.5μs. The relatively mild perturbation induced by FFTW is due to the small input set size (2000x2000 matrix) we use. While it is true that a larger input set size would bring more perturbation, we chose the small size to have one example of application with communication phases with not much data transfered within them. In case of VPFFT, there is perturbation in a small portion of packets, which corresponds to packets sent when VPFFT is going through intensive communication phases. In contrast, the primary effect of Lulesh and MILC is to shift the mode of the distribution to the right, close to 2.5μs. Further, while Lulesh didn't cause an increase in the fraction of packets with very high latency, with MCB this effect was strong. Interestingly, AMG shifts 50% of the packets to take more than 5.5μ but does not perturb much the latency times of the other 50%. Such behavior is explained by the different execution phases AMG goes through, since some of the them are communication intensive and some others are computation intensive.

3.2. Compression

Compression experiments measure the relationship between the network capability available to a software component and its performance by incrementally reducing network capability and observing the effect of this on performance. Since it is not possible to adjust the properties of real switches and network simulations are expensive, we use the performance relativity principle (reduced network capability affects application performance similarly to resource sharing) to simulate reduced network capability via software interference. We execute the target software component on a subset of the available cores. On the remaining cores we execute the CompressionB micro-benchmark, the pseudo-code for which is listed in Figure 4. CompressionB is executed on the same number of cores on each node, where processes running on the same core ID on different nodes are

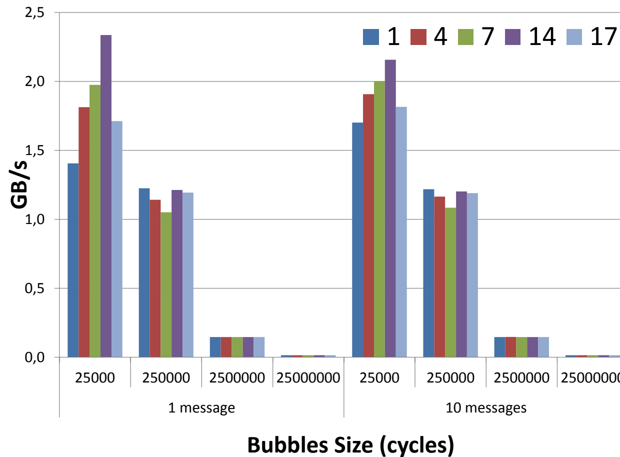

Figure 5: Network Bandwidth triggered per node by the CompressionB micro-benchmark

organized in a 1-dimensional communication ring. As illustrated in Figure 4, in each iteration every CompressionB process sends a message to P partner processes that precede it in the ring (all processes in its ring are on different nodes) and receives the messages sent by the P succeeding processes. Message size is 40 KB. After M messages have been sent in this way, the benchmark sleeps for B cycles, waits for all the `MPI_Irecvs` and `MPI_Isends` to complete, and repeats the communication pattern. Various settings of parameters P, M and B degrade network capability to different extents.

Figure 5 shows the bandwidth triggered per node by several CompressionB configurations when run on Cab. We map one CompressionB process on each socket, that is, 2 CompressionB instances per node. The experiments consider all the 18 nodes connected to a switch, which implies that there are 36 MPI processes in total. Parameter P, the number of partner processes, takes values 1, 4, 7, 14 and 17. Parameter B, the number of cycles the benchmark sleeps, has values $2.5 \cdot 10^4$, $2.5 \cdot 10^5$, $2.5 \cdot 10^6$, $2.5 \cdot 10^7$. Finally, parameter M, the number of messages sent in each round of communication, is either 1 or 10. As such, we consider 40 different input configurations of CompressionB. Since network links between computing nodes and first level switches have a 5GB/s capacity considering both incoming and outcoming bandwidth, we conclude that the 40 considered configurations sweep a wide range scenarios from using less than 1% up to 46% of the available network bandwidth per link. Our results stand for the outcoming bandwith per node triggered by the CompressionB micro-benchmark. This is achieved by using just 2 dedicated cores per node, one per socket, out of 16, which highlights the small amount of dedicated hardware CompressionB needs to emulate a large range of scenarios.

By performing multiple experiments where a different configuration of CompressionB is executed concurrently with a target software component it is possible to measure the degradation in the component's performance on less capable switches. The degradation is computed by co-running the target software

component with the CompressionB configuration C and comparing its performance with the one exhibited by the target software component when run on a dedicated network. If sofware component's performance is the same in both experiments, we conclude that the CompressionB configuration C does not bring any performance degradation. Since the CompressionB interference runs on dedicated cores, the performance degradation that software components may exhibit when co-run with CompressionB is entirely due to network interference issues.

3.3. Packet Latency Distributions of the Compression Benchmarks

To provide a clearer understanding of the traffic pattern injected into the network switch by the CompressionB benchmark, we show the packet latency distributions observed when co-run the 40 CompressionB configurations defined above with the ImpactB benchmark. Figures 6, 7, 8, 9 and 10 show the distributions obtained when setting parameter P, the number of partner processes, to 1, 4, 7, 14 and 17, respectively. Each Figure displays results corresponding to 10 different configurations where parameter B, the number of cycles the benchmark sleeps, has values $2.5 \cdot 10^4$, $2.5 \cdot 10^5$, $2.5 \cdot 10^6$ and $2.5 \cdot 10^7$ and parameter M, the number of messages sent in each round of communication, is either 1 or 10. Again, we map one CompressionB process on each socket, that is, 2 CompressionB instances per node. The experiments consider all the 18 nodes connected to a switch, which implies that there are 36 CompressionB MPI processes in total. On the other hand, 1 ImpactB MPI process was mapped per socket, 2 per node.

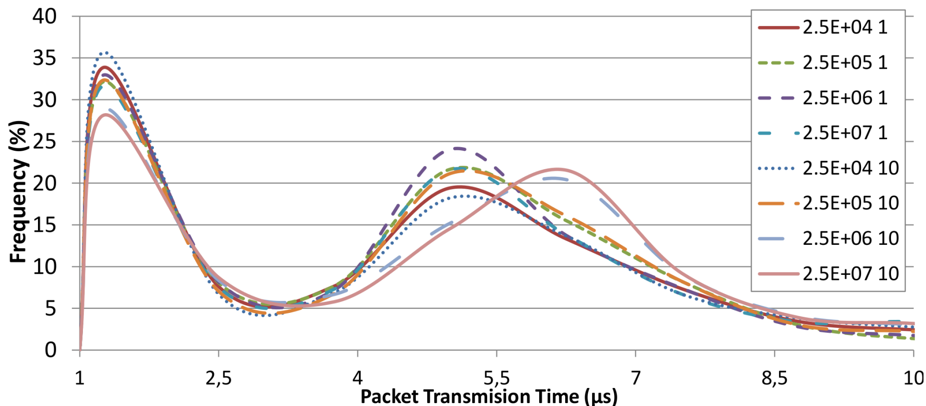

Figure 6: Distributions of Packet Latencies when CompressionB is co-run with ImpactB. Parameter P, the number of partner processes, is set to 1.

In Figure 6 we show the specific results when $P = 1$. As we can see, all the distributions have a bi-modal shape. The distributions' first mode is located close to the $1\mu s$ minumum possible latency while the second modes are around 5.5 μs. This behavior is due to the packet switching protocol implemented at the switch level, which gives some priority to some packets while others

are transfered with larger latency. We can clearly see in Figure 6 how the two configurations that inject the heaviest traffic into the network in terms of bandwidth for $P = 1$ according to Figure 5, $B = 2.5 \cdot 10^4, M = 10$ and $B = 2.5 \cdot 10^4, M = 1$, correspond to the ones with the highest degree of high priority packets. Indeed, they have the largest fast packet modes: 33.91% and 35.83% respectively. On the other hand, two of the CompressionB distributions with parameter P set to 1 that inject less bandwidth into the network according to Figure 5 ($B = 2.5 \cdot 10^6, M = 10$ and $B = 2.5 \cdot 10^7, M = 10$) have the slow packets mode switched to the right of the graph (around $6\mu s$) if compared with the others (around $5\mu s$). Interestingly, the $B = 2.5 \cdot 10^7, M = 1$ configuration injects less bandwidth into the network than the $B = 2.5 \cdot 10^6, M = 10$ one but has its slow packets mode located more to the left, which is an opposite effect as the one described a few lines above. This comparison between the results displayed by Figures 5 and 6 clearly illustrate the complex relationship between network bandwidth and contention. The explanation of this complex behavior resides in the fact that packet latencies distributions are determined by the message triggering pattern of the different CompressionB configurations rather than the absolute bandwidth they inject into the network switch.

In Figure 7 we show the obtained results when $P = 4$. Again, the distributions have a bi-modal behavior where the first mode is close to the $1\mu s$ minimum possible latency while the second one is close to the $5.5\mu s$ value. The two most bandwidth intensive CompressionB configurations when $P = 4$ according to Figure 7, $B = 2.5 \cdot 10^4, M = 10$ and $B = 2.5 \cdot 10^4, M = 1$, are the ones with the largest fast packet modes, which is a similar behavior as the one seen in Figure 6 for $P = 1$. There are 3 configurations ($B = 2.5 \cdot 10^5, M = 1$; $B = 2.5 \cdot 10^6, M = 10$ and $B = 2.5 \cdot 10^7, M = 10$) that have their high latency mode slightly switched to the right (around $6.25\mu s$) with respect the others. Interestingly, among these 3 configurations there are 2 that are not bandwidth intensive ($B = 2.5 \cdot 10^6, M = 10$ and $B = 2.5 \cdot 10^7, M = 10$) while another one is ($B = 2.5 \cdot 10^5, M = 1$). Figure 7 also shows how there are 4 CompressionB configurations ($B = 2.5 \cdot 10^7, M = 1$; $B = 2.5 \cdot 10^6, M = 1$; $B = 2.5 \cdot 10^4, M = 1$; $B = 2.5 \cdot 10^5, M = 10$) that have their low packet modes very close to $5\mu s$. There is another configuration ($B = 2.5 \cdot 10^4, M = 10$) displaying an intermediate behavior between the 2 sets of configurations we just described.

In Figure 8 we display results when $P = 7$. The same bi-modal behavior as the one previously observed appears in these results. In this case, the two most bandwidth intensive parameter sets ($B = 2.5 \cdot 10^4, M = 10$ and $B = 2.5 \cdot 10^4, M = 1$) are the ones with the largest low latency modes: 36.03% and 33.54%, respectively. Since the 4 configurations with the M parameter set to 10 have their high latency modes close to $6.25\mu s$ while the other 4 defined by the equation $M = 1$ have it close to $5\mu s$, it is clear that when $P = 7$ the M parameter determines where the high latency mode is located.

In Figure 9 we see the packet latency distributions when $P = 14$. As we can see in Figure 5, this set of distributions contains the most bandwidth intensive CompressionB parameter configurations. Configurations $B = 2.5 \cdot 10^4, M = 1$ and $B = 2.5 \cdot 10^5, M = 1$ have largest low latency modes. Also, these two

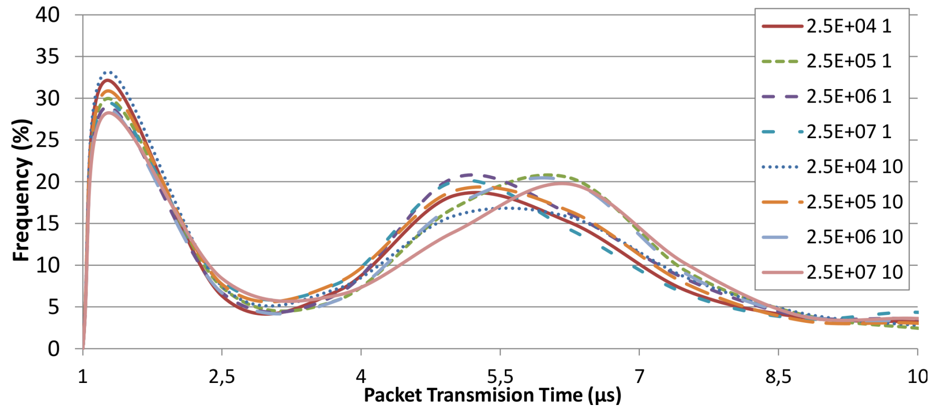

Figure 7: Distributions of Packet Latencies when CompressionB is co-run with ImpactB. Parameter P, the number of partner processes, is set to 4.

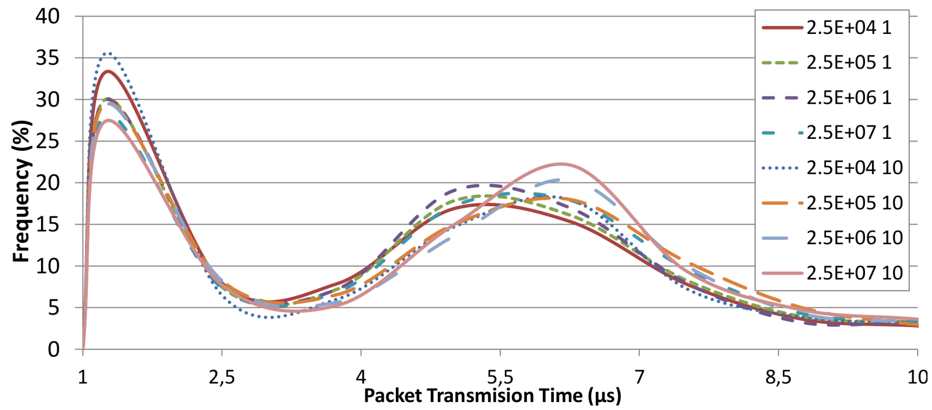

Figure 8: Distributions of Packet Latencies when CompressionB is co-run with ImpactB. Parameter P, the number of partner processes, is set to 7.

configurations have their high latency modes around $5\mu s$. In contrast, almost all the other configurations with parameter P set to 14 have their high latency modes switched to the right and close to the $6.25\mu s$ point. On the other hand, configuration $B = 2.5 \cdot 10^4, M = 10$ has a behavior located between these two sets of CompressionB configurations. Interestingly, the set of results displayed in Figure 9 is useful to illustrate the different behaviors in terms of packet latency distributions that CompressionB configurations with similar bandwith injection rates have. For example, configurations $B = 2.5 \cdot 10^4, M = 1$ and $B = 2.5 \cdot 10^4, M = 10$ inject large amounts of bandwidth into the network switch (above 2.0 GB/s according to Figure 5) but they have different behaviors in terms of their packet latency distribution. In contrast, configuration $B = 2.5 \cdot 10^4, M = 1$ has a latency distribution very close to $B = 2.5 \cdot 10^5, M = 1$ according to Figure 9 but the first one injects much more bandwith into the

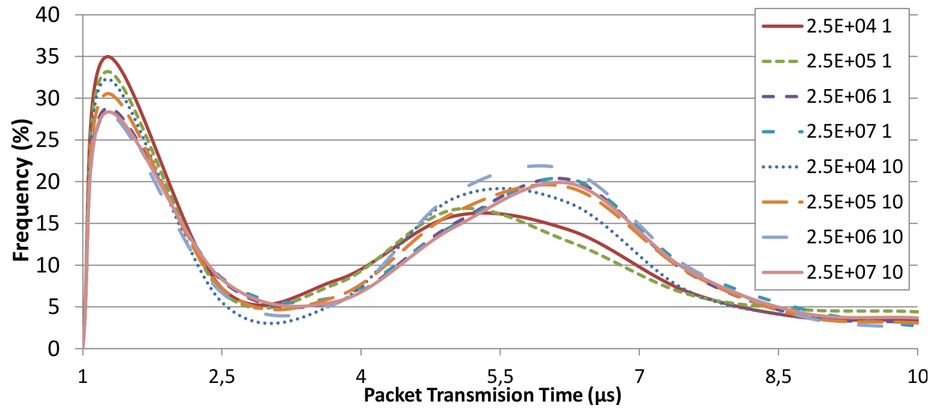

Figure 9: Distributions of Packet Latencies when CompressionB is co-run with ImpactB. Parameter P, the number of partner processes, is set to 14.

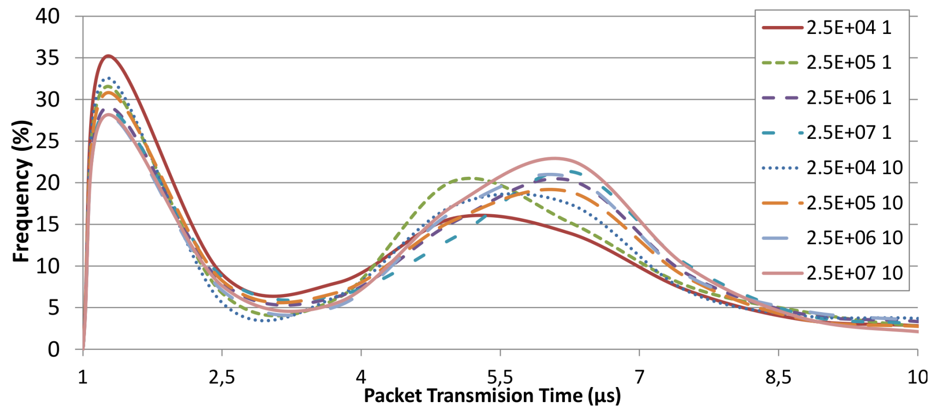

Figure 10: Distributions of Packet Latencies when CompressionB is co-run with ImpactB. Parameter P, the number of partner processes, is set to 17.

network switch (2.35 GB/s according to Figure 5) than the second one (1.21 GB/s). Again, this experiments demonstrate that there is no direct relation between the bandwith injected into the network switch by a particular software component and the amount of contention that packets face when they go through the network switch.

Results shown in Figure 10 display experiments performed by setting parameter P to 17. The only two configurations that have their large latency mode close to $5.5\mu s$ are $B = 2.5 \cdot 10^4, M = 1$ and $B = 2.5 \cdot 10^5, M = 1$. All the other configurations have this mode more switched to the right. Again, there is no direct correlation between the results shown in Figure 5 in terms of bandwidth and the ones shown in 10.

4. Modeling

In this section we describe the four modeling approaches we follow to get slowdown predictions when applications share network resources. These four approaches can be divided into two main categories: The look-up table based models and the queue model. Three of our four approaches are look-up table based while just the fourth is the queue approach. All of them use information obtained from the impact and compression measurement to compute the performance slowdown predictions.

4.1. Look-up Table Models

The look-up table models use a description of the intensity of the extra traffic injected by the compression benchmark and the performance degradation each application suffers for each level of traffic injection. As the compression benchmark has many different input configurations, we can consider from very light-weight to heavy traffic injections, which describes the application behavior under very different contexts. Additionally, the degree of perturbation each application brings, measured by using the impact benchmark, is also used.

To predict the performance slowdown of a particular application when it shares the network switch with a second workload, the model takes the level of perturbation that the second application brings, which is measured by the impact benchmark and summarized in a certain way, looks into a look-up table for the input configuration of the compression benchmark that brings the closest degree of perturbation, and then takes the subsequent performance degradation, previously measured by the compression benchmark, as the prediction.

We consider three different Look-up Table Models:

4.1.1. The Average Look-Up Table (AverageLT)

This model uses the average latency of the packets that travel through the switch as a metric to summarize network's usage. As such, to predict the slowdown of application A when co-runs with application B, the model takes the average latency of the packets triggered by B, μ_B, which can be computed via impact measurements, and then looks for the input configuration C_i of the compression benchmark that has the closest average value μ_{C_i}, which is computed by co-running the C_i configuration with the ImpactB benchmark. Once this identification is done, the model takes the slowdown that application B suffers when is co-executed with the selected compression workload.

4.1.2. The Average and Standard Deviation Look-Up Table (AverageStDevLT)

This model works in a very similar way as the previous, but instead of using just the average to select the input configuration of the compression benchmark to be used to predict, it uses the average and the standard deviation. As such, it takes the interval $I_B = [\mu_B - \sigma_B, \mu_B + \sigma_B]$ and the intervals $I_{C_i} = [\mu_{C_i} - \sigma_{C_i}, \mu_{C_i} + \sigma_{C_i}]$ for all the input configurations C_i of the compression benchmark, computes the lengths of the intervals $I_B \cap I_{C_i}$ and selects the configuration C_i that maximizes it. The idea behind this approach is to use the intervals I_{C_i} as proxies of the whole distribution of packets' latencies.

4.1.3. The Probability Distribution Function (PDF) Look-Up Table (PDFLT)

This model works very similarly as the previous ones, but it uses the whole distribution of latencies instead of just the average and the standard deviation. As such, if the application A runs with B, the model takes the distribution of the packet latencies triggered by B, f_B and the distributions f_{C_i} of all the considered compression workloads. Then, it computes the integrals $\int_0^\infty f_B f_{C_i}$. Since we have that $\int_0^\infty f_B f_{C_i} \leq \int_0^\infty f_B \int_0^\infty f_{C_i} \leq 1$, these integrals are well defined. Since the closer distributions f_B and f_{C_i} are, the bigger the integrals' values are, the model selects the configuration C_i that maximizes $\int_0^\infty f_B f_{C_i}$.

4.2. Switch Utilization Metric

While packet latency distributions can provide some insight into the effective capability of the switch, they do not vary monotonically with application performance since it is not clear whether one distribution represents more or less network utilization than another (e.g. compare Lulesh and MCB's distributions). However, they can be used to extract the appropriate metric by modeling the behavior of a switch as a mathematical queue and leveraging the results of queuing theory (QT) [40] to infer the state of this queue based on its observable behavior (the packet latencies).

We represent the real switch as a queue by considering that each packet arrives at one switch port, is processed by internal switch circuitry and then departs via another port. As Figure 1 illustrates, when the packet arrives at this queue other packets may already be waiting in the queue to be routed, forcing the packet to wait until these packets are processed. The length of the queue inside the switch depends on the pattern of packet arrival times at the switch. Specifically, we use the M/G/1 [32] queue model to represent switch routing logic. Since a M/G/1 queue is a stochastic process modeling the number of customers in a queue waiting for a service, it fits well with our situation where packets arrive to the network switch after being triggered by a software component. Once they reach the switch, they wait until they are sent to the proper node, that is, they behave like if they were customers waiting for a service.

QT defines the utilization of a queue as the proportion of its entries that are used by the arriving traffic. Utilization ρ can be expressed as the rate $\frac{\lambda}{\mu}$, where λ is the mean rate of packet arrivals and μ is the mean rate of packet service times. If $\rho \geq 1$ then the queue's waiting time will grow, which implies that the switch will be contended and application performance will degrade significantly. Parameters λ and μ must be known to measure ρ. μ is a hardware parameter that is measured by sending multiple individual packets into an idle switch and measuring their minimum latency. λ is an application specific parameter that can only be directly measured by using switch counters, which are not available in general as they require root privileges. However, λ can be computed via the the Pollaczek-Khinchine formula [18]:

$$W = \frac{\rho + \lambda \mu Var(S)}{2(\mu - \lambda)} + \mu^{-1} \tag{1}$$

Where W is the total average time spent by packets in the queue either waiting and being serviced and $Var(S)$ is the variance of the service times. Since utilization $\rho = \frac{\lambda}{\mu}$, we can write the formula as:

$$W = \frac{\frac{\lambda}{\mu} + \lambda\mu Var(S)}{2(\mu - \lambda)} + \mu^{-1} \tag{2}$$

which can be transformed to compute λ as follows:

$$\lambda = \frac{2 - 2W\mu}{-2W + \frac{2}{\mu} - \mu Var(S) - \mu^{-1}} \tag{3}$$

$Var(S)$ can be computed from the single-packet experiments on an idle switch and, importantly, W is just the average latency of the packets communicated by ImpactB while the target application runs. Since utilization $\rho = \frac{\lambda}{\mu}$, we can compute it by using the the above formula given the parameters obtained through ImpactB measurements. In our context, we call ρ the *switch utilization metric*. This metric aims to describe the fraction of switch capability used by parallel codes.

4.3. Switch Utilization of CompressionB

To quantify the fraction of switch capability that various configurations of CompressionB use, we run it together with ImpactB just like any other software component ImpactB may measure. This measurement makes it possible to relate performance degradation to the fraction of switch queue capability removed by CompressionB. The result is a high-level description of application performance in terms of a generic measure of network capability, the switch utilization fraction.

Our CompressionB+ImpactB experiments are executed using the same configuration as above, where we map 1 ImpactB and 1 CompressionB process on each socket, for 2 ImpactB and 2 CompressionB tasks per node. Figure 11 shows the range of different switch utilization percentages that can be achieved by all the considered variants of CompressionB when run on Cab. We consider the same 40 different input configuration of CompressionB as in section 3.2

The data shows that main determinant of switch utilization is the number of cycles the benchmarks sleeps, with utilization decreasing with longer sleeps. Further, utilization rises with increasing partner counts and message counts. The effect of partner count is strongest for longer sleep times while the effect of message count is strongest for shorter sleep times. In total, we consider 40 different input configurations, which allow us to cover switch utilization between 26% and 92%. The broad range of switch utilization provided by these configurations enables us to precisely evaluate applications performance degradations due to reduced switch capability.

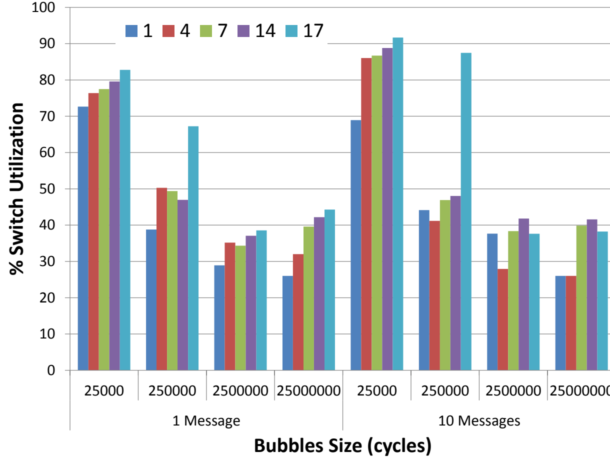

Figure 11: Switch usage compression benchmark on Cab. We consider compression workloads with 1, 4, 7, 14 and 17 partners.

4.4. Application performance impact due to reduced network capability

We use CompressionB to measure the relationship between available network switch capability and the performance of our target applications. Each experiment uses the same configuration as in Section 3.1, with 2 CompressionB processes per node. We assign 1 CompressionB process per socket. The other cores are assigned to the application or left idle. Figure 12 shows the percentage performance degradation on Cab of FFTW, Lulesh, MCB, MILC, VPFFT and AMG (y-axis, logarithmic scale) as the percentage of switch utilized by CompressionB changes across its full range (x-axis) due to the use of different configuration parameters. Performance degradation is computed as

$$\frac{\texttt{Run time with interference - Run time with no interference}}{\texttt{Run time with no interference}} \quad (4)$$

Reducing switch capability has the most effect on FFTW and VPFFT. FFTW runs more than 50% slower on Cab when even 40% of the switch queue is utilized and up to 250% slower as utilization reaches 92%. VPFFT also shows a very significant performance degradation, reaching a slowdown higher than 250% when 87% of the queue is used. VPFFT behavior is not as consistent as the ones observed in the other applications, showing oscillations from 132% to 263% of slowdown when 87% of the switch is used. MILC is also significantly affected, running approximately 20% more slowly on Cab at 40% switch utilization and over 100% more slowly at 92% utilization. This is because both applications are very sensitive to the latency of messages, meaning that if on average the queue is 40% full the stochastic nature of packet arrivals means that there are many packets that arrive when the queue is very long. Recall that the packet latency distributions shown in Figure 3 have some high latency packets even when the switch is idle. When the switch is partially utilized the fraction of high latency packets can become considerable, significantly degrading the performance of FFTW, VPFFT and MILC.

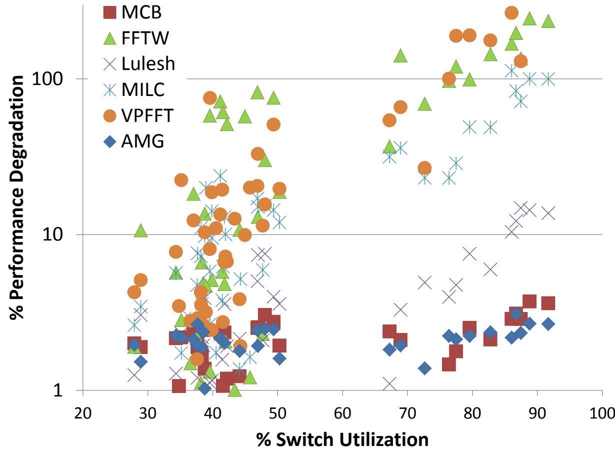

Figure 12: Performance degradations suffered by applications in terms of the switch utilization metric. The y-axis is expressed as a logarithmic scale.

In contrast, Lulesh, MCB and AMG are significantly less affected by reductions in switch capability. The performance of Lulesh degrades by 8% at 50% switch utilization and 15% at 92% utilization. MCB and AMG are almost completely insensitive to switch utilization, slowing by no more than 3.5% across the full utilization range.

The above experiments make it possible to estimate the performance of software components when executing on switches with different capabilities. Specifically, to focus on a particular scenario it is necessary to choose the switch utilization fraction that corresponds to the removal of the given amount of switch capability and run the application with CompressionB configured to emulate this utilization fraction.

5. Prediction

Experimental techniques ImpactB and CompressionB can be combined to make quantitative predictions about how the performance of multiple software components (application tasks or entire applications) will suffer when they are executed concurrently on the same switch. Our approach allows to train a model for each software component by running it together with several combinations of the CompressionB benchmark and just one time with the ImpactB benchmark. Therefore, the training cost grows linearly with the number of involved software components. Taking into account that the total number of pairings of n software components is n^2 and that the total cost of our training methodology is $n(k+1)$ where k is the number of CompressionB configurations considered in the training set, the prediction cost becomes negligible as the number of software components grows up.

We evaluate the accuracy of the proposed prediction algorithms by running pairs of our target applications concurrently on the same switch to observe

	FFTW	Lulesh	MCB	MILC	VPFFT	AMG
FFTW	45	5	3	11	12	7
Lulesh	5	5	3	6	2	3
MCB	3	5	4	7	5	6
MILC	25	12	1	4	3	14
VPFFT	9	0	2	5	7	2
AMG	0	5	4	5	3	4

Table 1: Measured Performance slowdowns for all the combined workloads. Numbers express percentages.

whether the model correctly predicts how much they degrade each other's performance. We run each benchmark in continuous loops and we measure the average slowdown over many concurrent runs. In these experiments each application is executed using the configurations used in the experiments reported in Sections 3.1 and 3.2. Specifically, for the experiments run with MILC, FFTW, MCB, AMG and VPFFT we ran 4 processes on each socket, one per core, for a total of 144 processes per application on the 18 dual-socket nodes connected to one switch. Since we co-run 2 applications, all the 288 cores sharing the same switch are used. Since Lulesh must run on cubic numbers of processes, we ran 2 Lulesh processes on each socket on 16 nodes, for a total of 64 processes. This process mapping utilizes at most half the available cores, leaving enough cores for two applications to run concurrently without sharing cores. Our experiments include combinations where two copies of a single application run concurrently on the same nodes and switch, as well as combinations where two different applications execute together. The former evaluates our model's accuracy on the use-case of HPC capability computing where different amounts of a single application's work may be assigned to a single switch. The latter accounts for the use-cases more typical in cloud computing or HPC capacity computing where multiple applications may share a single switch, as well as applications that run processes dedicated to molecular dynamics and processes for FFT computations concurrently on different nodes on the same network. In Table 1 we depict the measured slowdowns for all the possible application pairs. In each row we show all the possible performance slowdowns each applications experiments when is co-run with itself and with the other 5 applications. Numbers represent percentages.

5.1. Look-up Table Predictions

Using the methodologies explained in Section 4.1 we get some predictions of the measured performance slowdowns when the considered applications share the network resources. To apply these methodologies, we run each one of the applications with all the 40 input configurations of CompressionB considered in Section 4.3. Besides that, we run each one of these 40 configurations with the ImpactB to figure out, for each input configuration, the average packet transmission latency, the standard deviation and the complete distribution of the

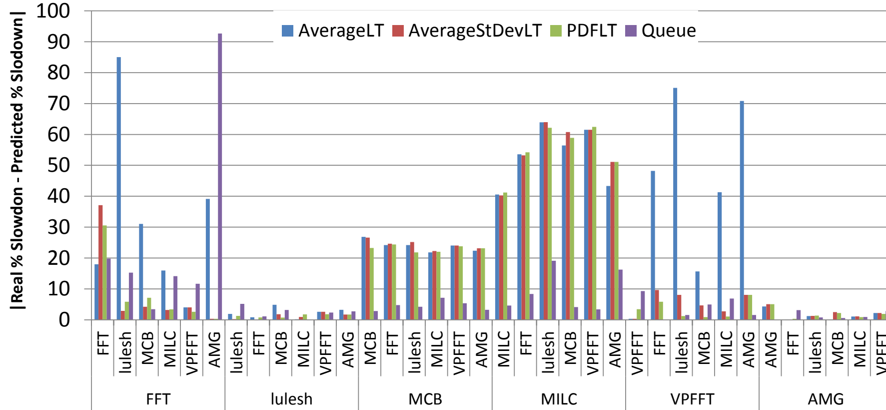

Figure 13: Performance predictions for combined workloads in Cab

packets latency. With these data we can apply the three methodologies explained in Section 4.1: AverageLT, AverageStDevLT and PDFLT. As such, if we want to predict the performance of application A when it runs with application B, we compute the closest configuration of ImpactB to application B and use the measured slowdown when A run with it as a prediction.

Figure 13 presents the results of the 36 experiments (6 experiments where each application was run with itself and 30 experiments for different application pairs) executed on Cab. The y-axis shows the difference between the measured and the predicted percent performance degradations of each application in each pairing, while the x-axis shows each pairing $X - Y$. Since experiments where two different applications are executed concurrently result in two different performance degradations, they are listed separately on the x-axis, for a total of 36 different degradation measurements. The x-axis is divided into 6 different boxes: Each box contains data referring to the application written horizontally when co-run with each one of the applications listed vertically.

Figure 13 shows the results we achieve using AverageLT, AverageStDevLT and PDFLT. The accuracy is quite good when the techniques predict slowdowns for Lulesh and AMG. The AverageLT model shows high deviations when it tries to predict slowdowns for FFT and VPFFT. However, the other two techniques improve AverageLT accuracy when predicting the behavior of these two applications. For MILC and MCB none of the techniques considered in this section, AverageLT, AverageStDevLT and PDFLT, achieve good results, which means that we have to increase the number of ImpactB configurations we consider to build our look-up tables or either apply more complex models. In the next section, we discuss the prediction methodology and the results achieved when the queue model is used.

5.2. Queue Model Predictions

Using the model explained in Section 4.2, we can measure the fraction of the switch queue that software components A and B use by conducting impact

experiments on A and B. They will result in quantities $U_A\%$ and $U_B\%$ that measure the fraction of the switch queue each component utilizes. Compression experiments on these components produce mappings p_A and p_B that map switch utilization fractions to the performance degradation in each component, like we show in Figure 12. We then use the configurations of CompressionB that also utilize $U_A\%$ and $U_B\%$ of the switch queue to model the effects of A and B, respectively on other software components with which they share a switch. We thus predict the performance degradation of A when executed concurrently with B to be $p_A(U_B)$. Specifically, this means that A's performance will degrade as much when sharing the switch with B as it did when it shared the switch with the configuration of CompressionB that utilizes the same fraction of the switch queue as B does. The converse prediction is made for B. This analysis can be performed for any combination of application tasks, their configurations (e.g. number of molecules simulated or the size of their communication stencil) or even multiple concurrently executing applications. Since predicting the degradation of B when co-runs with A is a totally independent process of predicting A's performance when co-runs with B, the accuracy of these predictions does not have to be necessarily the same.

Results in Figure 13 show that overall the queue model has very good predictive capability. For Lulesh, MCB, VPFFT and AMG the model properly predicts the performance slowdowns for all the possible co-running applications, as it clearly separates the pairings that induce little performance degradation from those that induce significant degradation. For MILC and FFTW, the model shows some deviations sometimes, which can be divide into three different categories: (i) the model predicts zero degradation while in reality performance degrades by 3%-5%, (ii) it predicts a degradation that a few percent higher or lower than reality (MILC with Lulesh and MCB) or (iii) the model predicts a notable degradation where in reality it was small (FFTW with AMG).

The only significant error is when the model predicts the performance of FFTW when co-executing with AMG. According to the model, the performance of FFTW would degrade significantly more than it actually does. The explanation of this high error is that, as AMG executions go through phases that do not significantly use the network, the switch capacity available to FFTW is close to 100% during a significant portion of its co-run with AMG, which is something that the queue model has not considered as it assumes a constant utilization of the network during the applications' runs.

5.3. Summary of the Results

In Figure 14 we summarize the results we have obtained with the four considered methodologies. For each method, we show two boxes that represent the second and third quartiles of the errors we have got predicting the 36 considered workloads. The line between the two boxes represents the median and the two error bars show the range covered by the errors of the first and the fourth quartile respectively. As we can see, the AverageStDevLT model outperforms the AverageLT, which is not surprising since the former uses more data than the latter. The accuracy of models AverageStDevLT and PDFLT is almost the

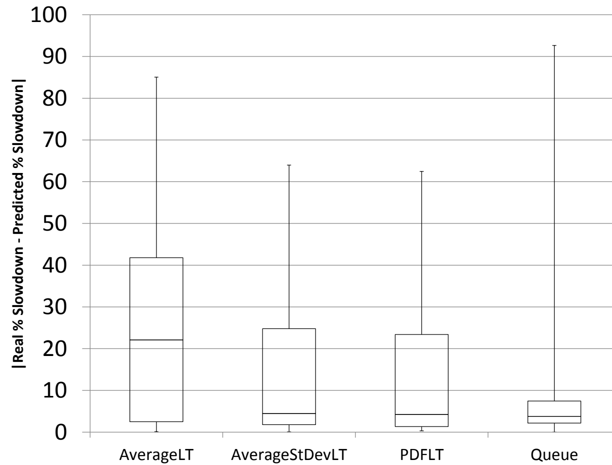

Figure 14: Summary of the results obtained in Cab

same, which means that just the average and the standard deviation of the packets' latency is already a good description of the whole distribution of latencies and that adding more information regarding this distribution does not increase the accuracy of the models. However, look-up table approaches do not achieve a satisfactory accuracy as more than one third of the predictions have an accuracy worse than 20%.

The queue model over-performs in general the look-up table approaches as more than 75% of its predictions have an error lower than 10%. Even more, as we can see in Figure 13 all of its predictions except one have an error lower than 20%. However, the queue model still shows a high deviation in one of the considered workloads.

6. Training Set Sensitivity

The previous sections evaluate the prediction accuracy of the 4 proposed techniques considering the training set defined in Section 4.3, which consists of 40 different configurations obtained combining parameters P, the number of partner processes, B, the number of cycles the compression benchmark sleeps and M, the number of messages sent in each round of communication. From these results, it is not clear whether or not that training set is optimal in terms of the prediction accuracy it provides or the overhead it involves. To provide a precise analysis of the prediction accuracy versus training overhead trade-off, we consider 12 different training sets and evaluate them in terms of prediction accuracy for each one of the 4 considered prediction techniques. In Table 2 we show in detail the 12 different training sets we consider for our evaluations. Per each set, we specify all the different values parameters B, P and M take. Set 1 is the smallest represented in the Table. It considers a Bubble Size parameter

Training Set	Bubble Size (B) in terms of # of CPU cycles	Number of Partners (P)	Number of Messages (M)
Set1	2.5E9	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set2	2.5E9 2.5E8	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set3	2.5E9 2.5E8 2.5E7	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set4	2.5E9 2.5E8 2.5E7 2.5E6	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set5	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set6	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set7	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5 1.25E5	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set8	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5 1.25E5 2.5E4	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set9	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5 1.25E5 2.5E4 1.25E4	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set10	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5 1.25E5 2.5E4 1.25E4 2.5E3	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set11	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5 1.25E5 2.5E4 1.25E4 2.5E3 1.25E3	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10
Set12	2.5E9 2.5E8 2.5E7 2.5E6 1.25E6 2.5E5 1.25E5 2.5E4 1.25E4 2.5E3 1.25E3 2.5E2	1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17	1 10

Table 2: Data set descriptions in terms of CompressionB parameters

of $2.5 \cdot 10^9$ cycles, 15 different numbers of communication partners from 1 to 17 and two different numbers of messages sent per iterations: 1 and 10. Therefore, this training set has a total of 30 sample points. As shown in the Table, the number of training points per set raises from 30 (Set 1), to 408 (Set12).

To design these training sets we consider the results shown in Figure 11, where it is clear that the largest variation in terms of switch utilization is obtained by changing the value of the bubble size parameter. From the data shown in Figure 11 it is also clear that the number of communication partners and the number of messages sent also provide some variations, but not as much as B. The design of the 12 training sets is made taking these considerations into account and trying to get large sets that sweep all the possible switch utilization numbers from 0% up to 100% and also small ones that consider a subset of this range.

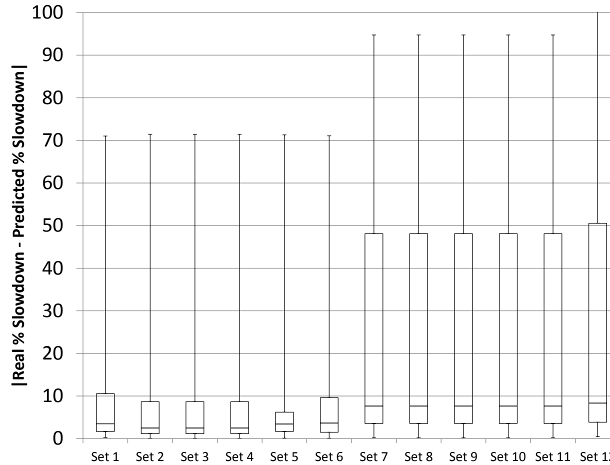

Figure 15: Prediction errors concerning the AverageLT. All the training sets shown in Table2 are considered to generate these results.

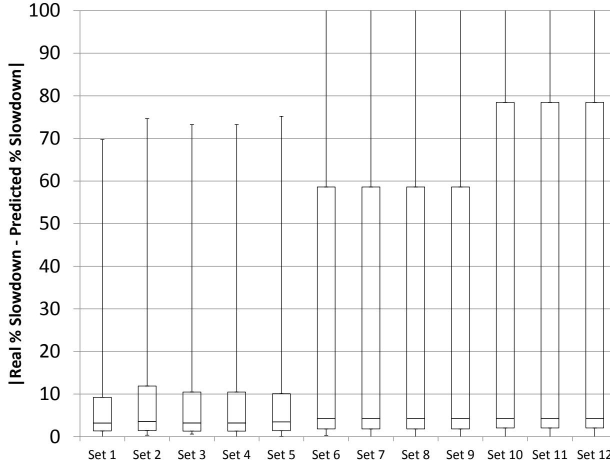

Figure 16: Prediction errors concerning the AverageStDevLT techniques. All the training sets shown in Table2 are considered to generate these results.

6.1. *Extended Look-up Table Predictions*

In Figures 15 and 16 we show the prediction accuracy provided by techniques AverageLT and AverageStDevLT respectively while Figures 17 and 18 represents results provided by the PDFLT technique and the queue model. The 12 training sets mentioned previously are considered in this campaign of experiments. For each method and training set, we show two boxes representing the second and third quartiles of the errors we get predicting the 36 considered application pairings. The line between the two boxes represents the median and the two error bars show the range covered by the errors of the first and the fourth quartile respectively.

In Figure 15 we can see the results obtained by the AverageLT technique. As it is shown in the figure, the median error is slightly below 5% for the first six training sets while the maximum error is around 71%. These values stay stable for the six initial training sets but they experiment a significant increase when set 7 is considered. The median prediction error increases from below 5% up to 8% while the maximum error raises up to 95% of prediction error. Significantly, the third quartile value suffers a large increase from around 10% up to 47% of the prediction error when comparing training sets 6 and 7. The main reasons behind this degradation in the prediction quality suffered by set 7 are the bad decisions the AverageLT model make when selecting the CompressionB benchmark configuration. The AverageLT technique selects a training set configuration included in set 7 but not in set 6, $\{1.25 \cdot 10^5, 9, 1\}$, to estimate the slowdown of applications when co-run with moderate switch usage workloads, like MILC or Lulesh. Since these applications actually use the network in a much smaller degree than CompressionB configuration $\{1.25 \cdot 10^5, 9, 1\}$, the prediction is not good. The reason for this bad choice of predicting configuration is the

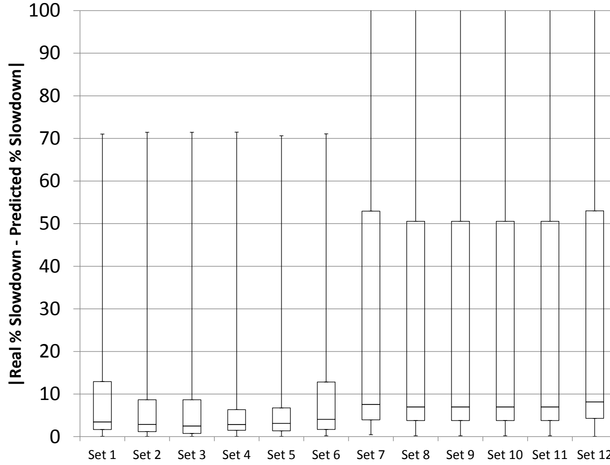

Figure 17: Prediction errors concerning the PDFLT technique. All the training sets shown in Table2 are considered to generate these results.

behavior of packets traveling through the switch under heavy network traffic. Since the distribution of packets' latencies is very irregular and noisy, its average is not a good metric to describe the real behavior observed in the network. This noisy behavior confuses the model, which ends up emulating applications with moderate switch usage, like MILC or Lulesh, with highly intensive CompressionB benchmarks configurations. Clearly, the AverageLT technique does not really get any benefit from increasing the size of the training set once noisy configurations are included. Consequently, metrics that properly filter the noisy behavior of some highly intensive traffic training configurations is required.

In Figure 16 results concerning the AverageStDevLT technique are shown. The median prediction error stays always below 5% for all the considered training sets, which means that this technique provides very good predictions for half of the 36 considered pairings no matter which training set is considered. The third quartile of the prediction errors raises from around 10% up to 58% when training set 6 is considered instead of 5. Also, the maximum prediction error increases from 75% to more than 100% when changing training set 5 for 6. This sudden decrease of the prediction accuracy comes from the irregular behavior of some new configurations included in Set 6. For example, in case of configuration $\{1.25 \cdot 10^5\, 17, 19\}$, the ratio $\frac{\sigma}{\mu}$ is equal to 2.18, while in configurations contained in set 5 this ratio is in general smaller than 1.5. These large σ values for some configurations C_i make intervals $I_{C_i} = [\mu_{C_i} - \sigma_{C_i}, \mu_{C_i} + \sigma_{C_i}]$ very large, which artificially extends the length of interval $I_B \cap I_{C_i}$, which is the metric the model uses to chose a configuration C_i to approximate application B. Again, the noise in the training sets' distribution confuses the model, which makes wrong decisions in terms of picking up a training configuration close to the application to be emulated.

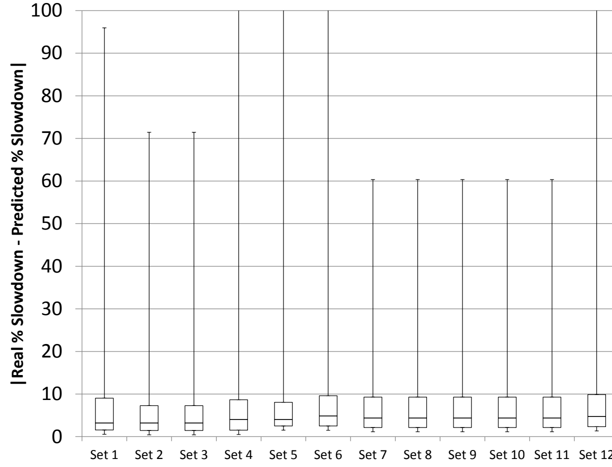

Figure 18: Prediction errors concerning the Queue Model techniques. All the training sets shown in Table2 are considered to generate these results.

Figure 17 displays results involving the PDFLT technique. For training sets 1, 2, 3, 4, 5 and 6 the median error is below 5%. If training sets 7, 8, 9, 10, 11 and 12 are considered, the median error is above 5% but still below 10%. The third quartile of the prediction errors experiments a drop from 12% to 8% when comparing results concerning training sets 1 and 4. However, it increases up to 53% when the training set 7 is considered and stays above 50% for training sets 7, 8, 9, 10, 11 and 12. With respect to the maximum error, it stays around 70% for training sets from 1 to 6 and grows above 100% for training sets 7-12. The decrease of the results' accuracy is due to a very similar issue as in the case of AverageStDevLT: The increase of the standard deviation of the packets switch latencies makes the metric proposed in Section 4.1.3 to pick up a bad CompressionB configuration.

6.2. Extended Queue Model Predictions

Results concerning the Queue Model predictions using the extended training sets are shown in Figure 18. For all the considered training sets, the median error is below 3.8% and the third quartile of the prediction errors stays below 10%. The average error is 9.6%. This means that Queue Model is able to provide satisfactory predictions for 75% of the considered workloads, even with a very reduced training data set. This is consistent with the good prediction results concerning 75% of the pairings shown in Section 5.2 which are obtained from a reduced data set. The main issue is the evolution of the large error predictions obtained for the remaining 25% of the considered workloads. As seen in previous sections, the AverageLT, AverageStDevLT and PDFLT techniques do not succeed in reducing the prediction errors when the training set is increase. The results shown in Figure 18 show how the Queue Model technique does

indeed reduce the maximum error to near 60% when Training Sets 7, 8, 9, 10 or 11 are used. This is a significant reduction of the maximum error, which reaches values above 90% when training sets 1, 4, 5 and 6 are considered and above 70% for sets 2 and 3. The only drawback is the increase suffered when the largest training set, 12, is considered. This increase is brought by the fact that training set 12 considers interference experiments with bubble sizes of $2.5E2$, which bring extremely noisy and unstable active measurements.

Our results show how the AverageLT, AverageStDevLT and PDFLT approaches provide good predictions when a large number of cycles occur between message injections. These techniques based on lookup table predictions are very sensitive to the noise of the switch latency under traffic intensive workloads. In contrast, the Queue Model predictions stay consistent across different training sets, as results shown in Section 5.2 and Figure 18 clearly demonstrate. Also, the quality of queue model's predictions improve as the training set size gets increased, as results shown in Figure 18 show. This last quality highlights the importance of the Queue Model method to carry out predictions of combined workloads and also confirms that queuing theory is a good abstraction to model the network contention brought by HPC workloads.

7. Related Work

The importance of network performance optimization has motivated significant research by the performance analysis community. It can be divided into two categories: simulation and indirect measurement. The simulation approach, exemplified by tools such as SST [34], BigSim [43], Dimemas [29] or Venus [35] uses a detailed model of network hardware to account for the path of every message sent by each application node. Although these tools can accurately predict the performance of a particular application configuration on current and future network designs, they have two limitations. First, the cost of using them can be high for many realistic large-scale applications since a full analysis requires a large-scale application run followed by a detailed simulation of its communications, which is too slow to use for live application runs, although feasible for making projections to future systems. Second, each simulation is valid for only one application configuration. To predict performance for a different assignment of application tasks to nodes or different distributions of work to tasks it is necessary to perform a simulation for this specific configurations. Since the space of possible permutations grows exponentially with the number of ways to configure the allocation of work to compute nodes, the simulation approach soon grows infeasible.

The indirect measurement approach is exemplified by tracing tools such as Vampir [26] and Paraver [28] as well as performance counter-based tools such as Tau [38]. In this approach various application regions are monitored to determine its communication structure, the amount of time it spends performing various operations and the number of events such as cache misses that occur during each operation. While these measurements can be used to derive non-trivial information of HPC applications, like the internal structure of their

executions [8] or the reasons behind performance slowdowns [7, 6], they can only enable indirect inference about how the properties of a network relate to application performance.

Detailed parametric models of HPC applications performance when run on large scale architectures include network parameters like latency or bandwidth [24]. Although machine- and application-specific, such models provide useful insights. HPC applications sensitivity to network parameters like latency or bandwidth has been previously evaluated [23] and differences up to 60% have been reported between the best and the worse performing network configurations. Also, sensitivity to network parameters is reported to be strongly application dependent. Other studies analyze HPC applications sensitivity to network noise [19]. These studies conclude that network noise can bring significant performance slowdowns to basic collective operations like reductions. Non-surprisingly, network noise is reported to grow with system size.

Other work has been more focused on exploring the usefulness of new network topologies to achieve significant reductions in both network cost and network power, while still providing a balance of high global and high local bandwidth [41]. Previous work [33, 12] considered how power may reduce interconnection networks' capacity. There have been several prior efforts to reduce power in interconnection networks with a particular focus on reducing power on infrequently used links. Both [25, 39] propose techniques to power down certain links in response to traffic behavior. The techniques presented in this paper can be used to evaluate this kind of power optimizations for interconnecting networks. Some work that has focused on the characterization of link utilization in large systems [31].

8. Conclusion

In this paper we have shown the usefulness of proactive measurements to analyze applications' consumption of switch resources and to predict performance degradations when those resources are shared with other workloads. This a very important problem since high performance computing infrastructures typically run several applications on the same time, all of them sharing the network. Our technique uses two interferences that inject extra network workload. The first determines the fraction of the network that is utilized by a software component (an application or an individual task) to figure out the existence and severity of network contention. The second aggressively injects network packets while a software component runs to evaluate its performance on networks with less capacity or when it shares network resources with other software components.

We then combine the information from the two types of experiment to predict the performance slowdown experienced by multiple software components (e.g. multiple processes of a single MPI application) when they share a single network. We have also validated our approach by comparing the predictions we get through our modeling and measurement techniques with real measurements obtained when two applications run together on the same switch. By using a queuing theory based approach we have been able to achieve excellent accuracy

in almost all the considered workloads. Also, our methodology is general in the sense that can be deployed in any kind of HPC infrastructure that uses any kind of interconnecting network to handle communication between computing nodes.

The sensitivity of the prediction techniques presented in this paper against the size of the training set is also evaluated. The Queue Model technique is demonstrated to provide accurate predictions considering very restricted training sets and also to improve its predictions when these sets are increased. That facts confirms the suitability of queuing theory to model network contention brought by HPC workloads. In contrast, the other three techniques AverageLT, AverageStDevLT and PDFLT do not experiment significant benefits when the training set gets increased, which suggests that approaches exclusively based on massive amounts of data do not always have good prediction qualities if they do not contain any physical notion of the problem being targeted.

[1] Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory.

[2] *Scientific Application Performance on Candidate PetaScale Platforms*. IEEE Computer Society, 2007.

[3] G. Bauer, S. Gottlieb, and T. Hoefler. Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3_rmd. In *CCGRID*, pages 652–659, 2012.

[4] A. Bhatele, N. Jain, Y. Livnat, V. Pascucci, and P. T. Bremer. Analyzing network health and congestion in dragonfly-based supercomputers. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 93–102, May 2016.

[5] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 41:1–41:12, New York, NY, USA, 2013. ACM.

[6] M. Casas, R. Badia, and J. Labarta. Prediction of behavior of mpi applications. In *Cluster Computing, 2008 IEEE International Conference on*, pages 242–251, Sept 2008.

[7] M. Casas, R. M. Badia, and J. Labarta. Automatic analysis of speedup of MPI applications. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 349–358, New York, NY, USA, 2008. ACM.

[8] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of mpi applications. *Int. J. High Perform. Comput. Appl.*, 24(3):335–360, Aug. 2010.

[9] M. Casas and G. Bronevetsky. Active measurement of memory resource consumption. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 995–1004, May 2014.

[10] M. Casas and G. Bronevetsky. Active measurement of the impact of network switch utilization on application performance. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 165–174, May 2014.

[11] J. Cetnar, J. Wallenius, and W. Gudowski. MCB: A Continuous Energy Monte-Carlo Burnup Simulation Code. In *Actinide and Fission Product Partitioning and Transmutation*, 1999.

[12] B. Dickov, M. Pericas, P. Carpenter, N. Navarro, and E. Ayguade. Software-managed power reduction in infiniband links. *2014 43nd International Conference on Parallel Processing (ICPP)*, pages 311–320, 2014.

[13] R. D. Falgout and U. M. Yang. hypre: a library of high performance preconditioners. In *Preconditioners, Lecture Notes in Computer Science*, pages 632–641, 2002.

[14] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[15] R. E. Grant, K. T. Pedretti, and A. Gentile. Overtime: A tool for analyzing performance variation due to network interference. In *Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '15, pages 4:1–4:10, New York, NY, USA, 2015. ACM.

[16] T. Groves, R. E. Grant, and D. Arnold. NiMC: Characterizing and eliminating network-induced memory contention. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 253–262, May 2016.

[17] T. Groves, R. E. Grant, S. Hemmer, S. Hammond, M. Levenhagen, and D. C. Arnold. (sai) stalled, active and idle: Characterizing power and performance of large-scale dragonfly networks. *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 00:50–59, 2016.

[18] J. Haigh. *Probability Models*. Springer, 2002.

[19] T. Hoefler, T. Schneider, and A. Lumsdaine. The impact of network noise at large-scale communication performance. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009.

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

[21] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[22] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 51:1–51:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[23] D. Kerbyson. A look at application performance sensitivity to the bandwidth and latency of infiniband networks. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7 pp.–, April 2006.

[24] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 37–37, New York, NY, USA, 2001. ACM.

[25] E. J. Kim, K. H. Yum, G. M. Link, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, M. Yousif, and C. R. Das. Energy optimization techniques in cluster interconnects. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 459–464, New York, NY, USA, 2003. ACM.

[26] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Parallel Tools Workshop*, pages 139–155. Springer, 2008.

[27] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA IPTO, September 2008.

[28] J. Labarta. New Analysis Techniques in the CEPBA-Tools Environment. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Parallel Tools Workshop*, pages 125–143. Springer, 2009.

[29] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A Parallel Program Development Environment, 1996.

[30] R. A. Lebensohn, A. K. Kanjarla, and P. Eisenlohr. An elasto-viscoplastic formulation based on fast fourier transforms for the prediction of micromechanical fields in polycrystalline materials. *International Journal of Plasticity*, 3233(0):59 – 69, 2012.

[31] E. A. Len, I. Karlin, A. Bhatele, S. H. Langer, C. Chambreau, L. H. Howell, T. D'Hooge, and M. L. Leininger. Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 909–920, Nov 2016.

[32] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*, volume 5. 1989.

[33] G. Patel, S. Chai, S. Yalamanchili, and D. Schimmel. Power constrained design of multiprocessor interconnection networks. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 408–416, 1997.

[34] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, Mar. 2011.

[35] G. Rodriguez, R. Beivide, C. Minkenberg, J. Labarta, and M. Valero. Exploring Pattern-aware Routing in Feneralized Fat Rree Networks. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 276–285, New York, NY, USA, 2009. ACM.

[36] J. C. Sancho, D. J. Kerbyson, and M. Lang. *Characterizing the Impact of Using Spare-Cores on Application Performance*, pages 74–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[37] V. Sarkar. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, DARPA IPTO, September 2009.

[38] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[39] V. Soteriou and L.-S. Peh. Design-space exploration of power-aware on/off interconnection networks. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 510–517, 2004.

[40] V. Sundarapandian. *Probability, Statistics and Queueing Theory*. PHI Learning, 2009.

[41] K. D. Underwood and E. Borch. Exploiting Communication and Packing Locality for Cost-Effective Large Scale Networks. 2012.

[42] M. Valero, M. Moreto, M. Casas, E. Ayguadé, and J. Labarta. Runtime-aware architectures: A first approach. *International Journal on Supercomputing Frontiers and Innovations*, 1(1):29–44, 2014.

[43] G. Zheng, G. Kakulapati, and L. V. Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *IPDPS*. IEEE Computer Society, 2004.