# The Effect of System Utilization on Application Performance Variability

Boyang Li
Illinois Institute of Technology
Chicago, IL
bli70@hawk.iit.edu

Sudheer Chunduri
Kevin Harms
Argonne National Laboratory
Lemont, IL
sudheer@anl.gov,harms@alcf.anl.gov

Yuping Fan
Zhiling Lan
Illinois Institute of Technology
Chicago, IL
yfan22@hawk.iit.edu,lan@iit.edu

## ABSTRACT

Application performance variability caused by network contention is a major issue on dragonfly based systems. This work-in-progress study makes two contributions. First, we analyze real workload logs and conduct application experiments on the production system Theta at Argonne to evaluate application performance variability. We find a strong correlation between system utilization and performance variability where a high system utilization (e.g., above 95%) can cause up to 21% degradation in application performance. Next, driven by this key finding, we investigate a scheduling policy to mitigate workload interference by leveraging the fact that production systems often exhibit diurnal utilization behavior and not all users are in a hurry for job completion. Preliminary results show that this scheduling design is capable of improving system productivity (measured by scheduling makespan) as well as improving user-level scheduling metrics such as user wait time and job slowdown.

## KEYWORDS

performance variability; dragonfly network; system utilization; job scheduling; application experiments

**ACM Reference Format:**
Boyang Li, Sudheer Chunduri, Kevin Harms, Yuping Fan, and Zhiling Lan . 2019. The Effect of System Utilization on Application Performance Variability. In *9th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'19), June 25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3322789.3328743

## 1 INTRODUCTION

Interconnection networks play a critical role in high-performance computing systems. They serve as a "central nervous system" for data exchange between computer nodes. As the computation becomes cheaper, the network is increasingly becoming a scarce resource. Dragonfly topology provides high bandwidth and low network diameter, hence being regarded as a promising solution for building exascale systems. Nevertheless, dragonfly-based systems are vulnerable to *performance variability* due to network sharing. A recent study shows that the run-to-run variability of application runtime can be up to 2X [11]. Performance variability causes serious issues for application benchmarking and cluster scheduling. For example, HPC users are required to provide runtime estimate for their job upon job submission. When a job's execution time exceeds its runtime estimate, it gets killed by the scheduler. Due to performance variability, cautious users tend to exaggerate their job runtime estimate, which in turn leads to poor scheduling efficiency [13, 22].

Recent studies have identified that communication interference due to network contention is a dominant cause of performance variability [24]. Exploiting job scheduling to mitigate communication interference is an active research topic. Existing cluster scheduling studies mainly focus on developing an appropriate job placement policy or routing scheme in an attempt to intelligently map application processes onto compute nodes so as to alleviate communication interference [9, 23, 24]. Being complementary to the existing studies, this work seeks to examine the performance variability problem on dragonfly-based systems from a different angle.

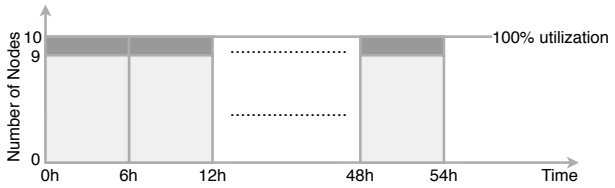
### 1.1 Paper Contributions

In this study, we first investigate how system utilization influences application runtime variability. Specifically, we analyze the real workload logs collected from the Theta [7] system to study the performance variability of production applications. Theta is a 11.69-Petaflop production system equipped with 4,392 nodes, each containing a 64 core Intel Xeon Phi processor. We observe that *there is a strong correlation between performance degradation and system utilization* where a high system utilization (e.g., above 95% for the workload in this study) can cause average application runtime increase by up to 21%. We further validate this observation by conducting application experiments (over 4000 application tests) to assess application performance variability. Our log analysis and application experiment clearly indicate that application runtime tends to increase by up to 20% under a high system utilization period. In a high system utilization environment, it is more likely for the applications to compete for network resources (e.g., link bandwidth), hence resulting in longer application runtimes.

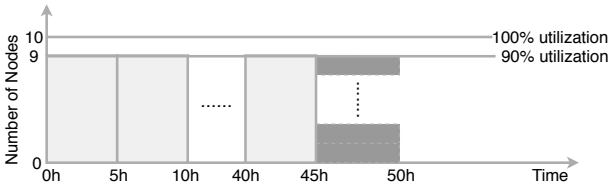
HPC systems are expensive, and the conventional scheduling focuses on achieving high system utilization (i.e., close to 100%). The hypothesis of this study is that *on shared networking systems such as dragonfly, we shall not solely target high utilization for scheduling because of the potential application performance degradation.* We provide a simple example to illustrate our argument. Suppose we have a 10-node system, and there are eighteen jobs waiting for allocation: nine 9-node jobs and nine 1-node jobs, each having a runtime estimate of 5 hours (Figure 1). The system has a shared interconnect network where performance variability is observed. We assume for each application, its runtime will be increased by


*ROSS'19, June 25, 2019, Phoenix, AZ, USA*

ACM ISBN 978-1-4503-6755-4/19/06…$15.00
https://doi.org/10.1145/3322789.3328743




**(a) A scheduling design targeting at high system utilization**




**(b) A scheduling design targeting at high system productivity (i.e., makespan)**

**Figure 1: Scheduling for utilization vs for productivity**

20% (thus becoming 6 hours) due to network sharing when system utilization is greater than a threshold (e.g., 95%). Figure 1(a) presents the case where the scheduling goal is high system utilization (i.e., 100%), whereas Figure 1(b) illustrates a different scenario where a higher system productivity (measured by scheduling makespan — the total length of the schedule to complete all the jobs) is achieved. The example clearly shows that a system of 100% utilization may not be as productive as a system of 90% utilization.

It is time to rethink the design of HPC scheduling on shared dragonfly systems. Rather than solely targeting at high system utilization, other aspects like performance variability and system productivity should be considered in the scheduling design. Ideally, systems should provide real-time traffic monitoring mechanism. When a user job is submitted, the scheduler interacts with the traffic monitor to keep track of all of the traffic in the network and to detect network contention occurring at an endpoint or inside of the network fabric. It then invokes a learning model to model the behavior of complex, interacting workloads on the shared resources and to estimate potential system productivity values of when/where to allocate the user job onto the system through cost-benefit tradeoff analysis. Unfortunately, the existing Dragonfly systems do not have such a traffic monitor. In the recent announcement [2], Cray Slingshot interconnect for the next generation of supercomputers is expected to provide a very complex set of traffic monitoring and bookkeeping.

For now, the question is *whether it is possible for us to improve system productivity on a dragonfly system by leveraging the observation made in our empirical analysis*. In practice, there are ample opportunities for applications to avoid execution under heavy loads. First, on a production supercomputer, resource utilization often exhibits a fluctuating pattern where the utilization rate has a diurnal pattern throughout a day [15]. For instance, the Theta workload log shows that resource utilization fluctuates throughout the day and the fluctuation can be as high as 3X. Such a bursty behavior presents an opportunity to avoid utilization peaks without impacting the system throughput. Second, production HPC facilities typically serve users all over the world and the users of different regions have different sleep or work schedules. At any time, some users are active while others are inactive (e.g., sleep or travel). Being analogous to the electricity charge system where the electricity companies provide a lower price to attract residents to use electricity at night [26], similarly HPC facilities could offer users a discount charge on core-hours if they allow their jobs to be postponed for flexible scheduling. The delay of jobs belonging to inactive users can further create more opportunities for the scheduler to make a flexible job allocation for mitigating potential workload interference.

In the second part of this paper, we explore a scheduling design named *CEIL* for the system whose resource utilization exhibiting a diurnal pattern such as Theta. CEIL allows users to specify at the job submission whether their job is postponable or not. CEIL holds the postponable jobs temporarily and releases them during low utilization period. Moreover, when scheduling a job from the head of the waiting queue, if the allocation of the job causes the system to a high system utilization situation, CEIL will choose a smaller sized job from the waiting queue. By actively monitoring system utilization, CEIL attempts to avoid scheduling jobs when the system is heavily loaded. We evaluate CEIL through trace-based simulation with real workload traces collected from Theta at Argonne. Our preliminary results show that by leveraging the fluctuating resource usage pattern and the existence of postponable jobs, CEIL can effectively reduce high system utilization periods (measured by the metric *percentage of high utilization periods*) without sacrificing the system throughput (i.e., *scheduling makespan*). Moreover, CEIL is capable of improving the user-level metrics, namely user wait time and job bounded slowdown, by up to 35%.

CEIL can work with any of the job ordering policies for enforcing job priority according to a site's policy. It is orthogonal to existing topology-aware job placement studies [9, 23, 24]. For a given job, CEIL can be used to intelligently determine when it should be executed to avoid performance degradation, whereas topology-aware job placement scheduling can be applied for a smart job placement once the job is allocated onto the system for execution.

### 1.2 Paper Outline

The remainder of this paper is organized as follows. We first discuss related work and background of this study. The empirical analysis of system utilization on application performance variability is presented in Section 3. Section 4 present CEIL design and its trace-based simulation results. Finally, we summarize the paper in Section 5.

## 2 BACKGROUND AND RELATED WORK

### 2.1 HPC Cluster Scheduling

A cluster scheduler is responsible for allocating resources and for determining the order in which jobs are executed on a HPC system. When submitting a job, the user is required to provide two parameters of the job: number of compute nodes required for the job (i.e., job size) and job runtime estimate (i.e., walltime). The scheduler determines *when and where* to execute the jobs. Once a new job is submitted, job scheduler sorts all the jobs in the waiting queue based on a job prioritizing policy. A number of popular job prioritizing policies have been proposed, and one of the widely used

policy is FCFS [18], which sorts jobs in the order of job arrivals. Another scheduling policy called WFP [8] periodically calculates a priority increment for each job in the queue. In addition, backfilling is a commonly used approach to enhance job scheduling by improving system utilization, where subsequent jobs are moved ahead to utilize free resources [18, 21]. A widely used strategy is EASY backfilling [18] which allows short jobs to skip ahead under the condition that they do not delay the job at the head of the queue.

On Theta, Cobalt is deployed for cluster scheduling [8]. In order to achieve a number of specific site goals, a scheduling policy called *WFP* with EASY-backfilling is adopted for batch scheduling on Theta. It uses a utility "function" to prioritize jobs for scheduling.

## 2.2 Related work

The impact of communication interference on application performance variability on HPC systems has been studied for a long time. Skinner and Kramer [20] identified significant performance variability due to network resource contention. They claim that performance variability is inevitable on either fat-tree or torus networks as long as network sharing is permitted among concurrently running applications. Bhatele et al. [10] used a specific application, p3FD, to study the performance variability on different HPC production systems with torus interconnect topologies and found that performance variability on Cray XE6 system was due to the communication of competing jobs. Yang et al. [25] studied the communication behavior of three parallel applications on a torus network and analyzed interference by simulating three applications running both independently and concurrently. They found resource allocation with respect to communication pattern awareness contributes to alleviating inter-job interference.

Since the dragonfly topology was proposed, many studies have been conducted to explore the efficiency of routing, task mapping, and job placement on such networks. Jain et al. analyzed the behavior of a dragonfly network with various job placement and routing policies [16]. They demonstrated the cost and benefit for each policy with synthetic applications and traffic models. Yang et al. [24] found the performance improvement of communication intensive applications come at the expense of performance degradation of less intensive applications.

Job scheduling has a long history, and is one of the major software components in the HPC area. The well-known commercial and open source job schedulers include PBS [6], SLURM [27], Cobalt [8], etc. All the existing research on mitigating job interference with the help of job scheduling is to focus on developing an appropriate job placement policy or routing scheme which attempts to intelligently allocate job processes onto computer nodes to alleviate communication interference.

This study differs from the above studies in two aspects. First, this work provides extensive log analysis and application experiments on a production system. Our analysis shows that *a high system utilization leads to longer applicatin runtimes.* Second, being orthogonal to the existing topology-aware job placement studies, this work proposes the use of proactive scheduling strategy to avoid the allocation of user jobs under high system utilization. We believe that in order to reduce job interference on dragonfly systems, it is critical for job scheduling to include both a proactive scheduling strategy like CEIL and a topology-aware placement policy [9, 23, 24].

**Table 1: Logs of Theta at ALCF**

| Log name | Number of record items | Time period |
|---|---|---|
| Aprun log | 307303 | Jan/2018-March/2018 |
| Cobalt log | 44870 | Jan/2018-March/2018 |

# 3 EMPIRICAL ANALYSIS

In this section, we examine how application runtimes vary under different system utilization rates. We first analyze a Theta workload which consists of production jobs submitted by various users over 3-month period. The observations from this analysis are then validated using real application experiments on Theta. For each application experiment, we conduct hundreds of application runs on various times and days. In total, we perform 4000+ application runs.

## 3.1 Log analysis

For the log analysis, we use both the Cobalt and aprun logs [8, 17]. Cobalt log records job specific information such as start time, end time, project name and number of nodes requested. A Cobalt job can contain multiple aprun commands. Aprun log records the meta data related to application execution such as executable name, command line input arguments and exit code. We use the aprun log to identify the records belonging to the same application, and use the corresponding Cobalt log entries to calculate the average system utilization during an application's execuation. Table 1 summarizes the details of the logs used in this study.

In particular, we examine both logs to identify the repeated executions of the applications and then analyze how their runtimes vary under the potentially different system utilization rates during these executions. We consider two log entries belong to the same application only when all of the following information is matched: user name, project name, number of nodes (job size), name of the executable, aprun command, current working directory, and the exit code. In our analysis, we only consider the executions that exit system normally. While binning the log records based on the application executable names, we ensure the applications that have seen major source code changes are not considered. Also, we ignore the records for applications whose performance variability is larger than 3X as they potentially belong to microbenchmarks or development codes. Log records for the jobs using 16 or fewer nodes are also discarded since the size of debug queue on Theta is 16 nodes. Essentially, we make sure that the log records corresponding to real production scale applications are considered for further analysis.

From the 3-month logs, we identified 15 applications that have multiple executions. These applications are from different science domains such as Material Science, Engineering and Physics. Table 2 summarizes top five applications with high repetition frequency for various job sizes. We refer these applications as AppA, AppB, AppC, AppD, AppE to comply with the system privacy policies.

For these five applications, the log records corresponding to their multiple runs are binned into two categories: one group of runs were

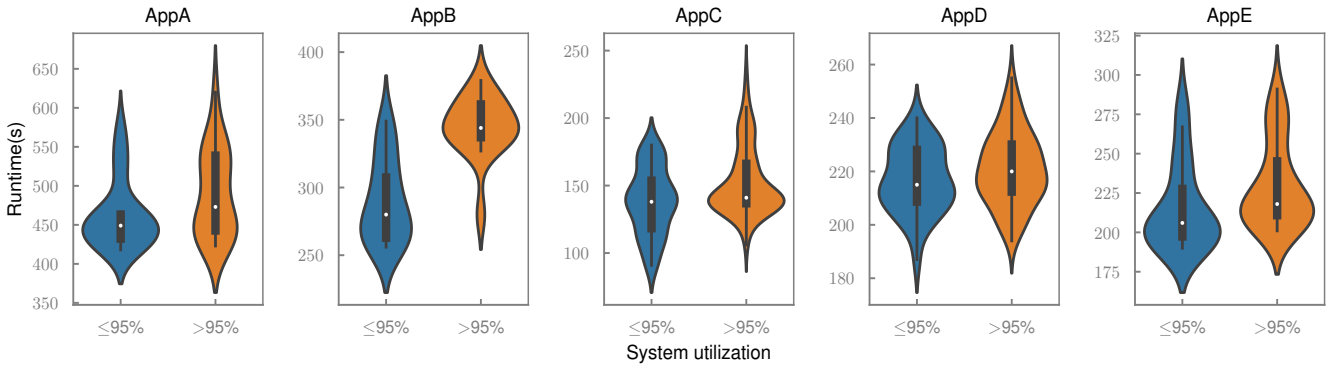



**Figure 2: Application runtimes under different system utilization rates. Here application runtimes and system utilization rates are extracted from the Theta log (Jan-March of 2018).**

**Table 2: Summary of application experiments**

| Name | Nodes | Runs | Sciece Domain | Avg. runtime(s) |
|---|---|---|---|---|
| AppA | 32 | 79 | High Energy Physics | 473.2 |
| AppB | 128 | 74 | Engineering | 310.2 |
| AppC | 256 | 194 | Material Science | 146.1 |
| AppD | 512 | 112 | Plasma Physics | 213.4 |
| AppE | 1024 | 73 | Material Science | 234.6 |

executed under >95% utilization (referred as *High*) and the other group of runs were executed under <=95% utilization (referred as *Normal*). In order to ensure better statistical significance of the following analysis, the two groups should be balanced in terms of their sample sizes. Having the 95% utilization is an appropriate boundary for ensuring this balance for this workload set.

Figure 2 presents the distribution of application runtimes under different system utilization rates. For each application, the system utilization rate is calculated as the average of system utilization rates observed during its execution.

We notice that *there is a strong correlation between high system utilization and application performance degradation.* For each application, the summary statistics (mean, median and IQR) of the runtimes under the *High* utilization are higher than the same under the *Normal* utilization. Across the applications, compared to the runs under *Normal* utilization, the runs under *High* utilization show an increase ranging from 6.7% to 21.1% in their mean runtime. *Two-sample Z-test* [12] is applied to evaluate the statistical significance of the difference between the means under *Normal* and *High* utilization. The hypothesis that the mean of the runtimes under the *High* utilization is larger than that under *Normal* utilization is accepted by two-sample Z-test with 90% confidence. Furthermore, for each application, its maximum runtime always occurred when the system utilization was higher than 95%.

Among the 15 applications, 12 applications exhibited the same statistical behavior as shown in Figure 2. However, three applications do not exhibit any clear pattern, and the possible reasons are: the application is not communication intensive, or the input files with the same name have different contents.

While the log analysis is about analyzing the jobs submitted by various users, the controlled experiments are about running the experiments ourselves. In order to validate the observations made from the log analysis study, we conduct real application experiments spanning over several months, more details are described in the following subsection.

## 3.2 Application Experiments

We use four production applications, MILC [3], Reordered MILC, Nek5000[4] and Nekbone [5] as part of the controlled experimental study.

**MILC**: This code represents Lattice Computation to study Quantum Chromo Dynamics. It implements SU3 lattice gauge theory and the primary kernel performs a communication intensive stencil operation on a 4D grid.

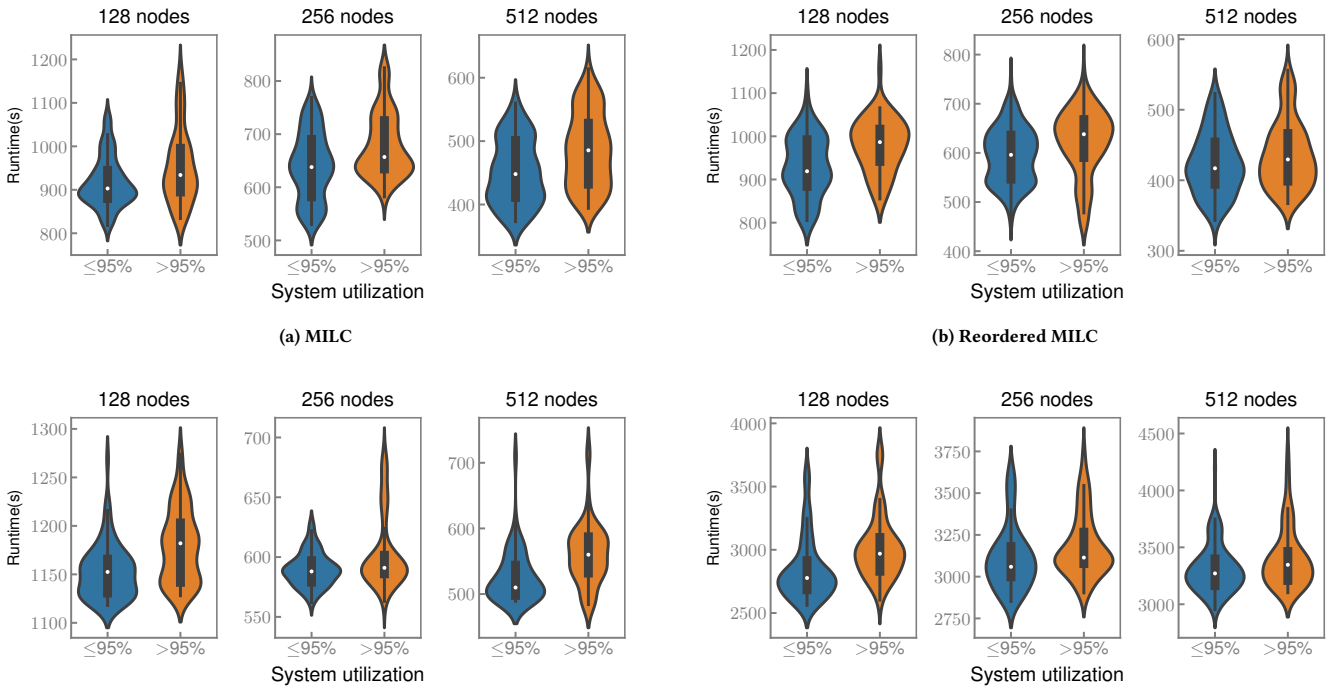
**Reordered MILC**: Reordered MILC is the same MILC application run with rank reordering. The rank reordering is done so as to ensure that intra-node communication is maximized compared to inter-node communication. Thus compared to MILC, it has more intra-node communication and less inter-node communication.

**Nek5000**: Nek5000 is an open-source software that can be used to simulate laminar, transitional, and turbulent incompressible or low Machnumber flows with heat transfer and species transport.

**Nekbone**: Nekbone is a mini-app derived from the Nek5000 CFD code which is a high order, incompressible NavierStokes CFD solver based on the spectral element method.

We run each application multiple times as listed in Table 3 on different days and times. For each application, three different job sizes (128 nodes, 256 nodes, 512 nodes) are used. Altogether, we perform over 4000 application tests for this study. We collected corresponding Cobalt logs for calculating the average system utilization during these application runs.

Figure 3 shows distributions of the runtimes under *Normal* and *High* system utilization scenarios. The median of the runtimes for runs under *High* utilization is larger compared to the same for runs under *Normal* utilization. The mean of the runtimes for runs under *High* utilization is also higher, ranging from 5.2% to 12.7%, than the mean for runs under *Normal* utilization. *The hypothesis that the mean of the runtimes under High utilization is larger than that*




(a) MILC

(b) Reordered MILC

(c) Nek5000

(d) Nekbone

**Figure 3: Application runtimes under different system utilization rates. These results are obtained from actual application experiments performed on Theta. For each application, three different job sizes (128, 256, 512 nodes) are used.**

**Table 3: Description of Application Experiments**

| Application name | Number of nodes | Number of runs |
|---|---|---|
| MILC | 128 | 502 |
| | 256 | 520 |
| | 512 | 440 |
| Reordered MILC | 128 | 241 |
| | 256 | 509 |
| | 512 | 560 |
| Nek5000 | 128 | 156 |
| | 256 | 205 |
| | 512 | 120 |
| NEKBONE | 128 | 365 |
| | 256 | 319 |
| | 512 | 259 |

*under Normal utilization is accepted by the two-sample Z-test with 90% confidence, thus confirming the statistical significance of the analysis.* In addition, the maximum runtime occurs only during *High* utilization.

Furthermore, the difference between maximum runtimes under *High* utilization and *Normal* utilization is larger for MILC than for Reordered MILC. Compared to MILC (Figure 3(a)), Reordered MILC (Figure 3(b)) uses less inter-node communication thus potentially has lower chances of performance degradation. This confirms that performance degradation under *High* system utilization environment is primarily attributable to the communication interference.

These application experiments based analysis validates the key findings obtained from our log analysis, that is, *a high system utilization can lead to a longer application runtime and the worst application performance always occurs when the system utilization is greater than 95%.*

The aforementioned log analysis and application experiments clearly demonstrate that applications likely suffer significant performance degradation when system utilization is *High* (e.g., above 95% on Theta). Hence, it is prudent for the job scheduler to *proactively* avoid the allocation of user jobs onto the system under *High* utilization.

## 4 SCHEDULING FOR PRODUCTIVITY

In this section, we investigate a scheduling policy named CEIL. CEIL is designed to *proactively* avoid the allocation of user jobs onto the system under high utilization. The goal is to improve system productivity measured by scheduling makespan (defined as the total length of time to complete all the jobs), rather than system utilization.

### 4.1 CEIL Description

The design of CEIL is based on two typical operating features of HPC systems: (1) not all users are in a hurry for their job completion; (2) the resource utilization on the system fluctuates throughout the day. Thus, we introduce the notion of postponable jobs where these

jobs can be postponed and possibly get scheduled to run under relatively low utilization periods.

CEIL can work with any base scheduler that enforce job priority according to a site's policy. It allows users to specify at job submission whether their job is postponable or not. Users can also specify an expected job completion time for a postponable job. Besides the conventional Waiting Queue as is adopted in the existing schedulers, there is an additional queue called Postpone Queue. When jobs are submitted, the jobs which can be postponed are put in the Postpone Queue temporarily. The conditions of releasing jobs from Postpone Queue to Waiting Queue are described in the following subsection. Jobs in the Waiting Queue are ordered according to the site's policy. Only the jobs in the Waiting Queue can be scheduled for execution. CEIL selects the job from the head of the Waiting Queue and checks whether the allocation of this job would increase utilization beyond a threshold (e.g, 95%). If the check fails, CEIL allocates the resources for the job. However, there is an exception to this check for the jobs requesting the entire machine or occupying 95% or more of the nodes. In such cases, CEIL will schedule them without applying the threshold limitation. During the backfilling process, CEIL backfills the ready jobs from the Waiting Queue before looking at the Postpone Queue. CEIL also avoids backfilling jobs which would increase utilization beyond the threshold.

Given that the scheduler only allocates resources to the jobs in the Waiting Queue, jobs are not kept permanently in the Postpone Queue. Thus, when one of the following conditions is satisfied, the jobs in the Postpone Queue are moved to the Waiting Queue.

**Empty Waiting Queue**: When the Waiting Queue is empty, jobs in the Postpone Queue are moved to the Waiting Queue so that the job scheduler can allocate available resources to these jobs for improving the system utilization.

**Low utilization**: When the utilization is low and there are still jobs in the Waiting Queue, it means the sizes of the jobs in the Waiting Queue are too large to be "backfilled". When these big jobs accumulate in the Waiting Queue, there is a possibility that the jobs in the Postpone Queue will not complete before the deadline, i.e., the specified expected job completion time. Hence, a threshold of 60% utilization is used to specify the lower bound of system utilization. If the system utilization is lower than this threshold, jobs in the Postpone Queue are moved to the Waiting Queue.

**Approaching user's expected job completion time**: When the user's expected job completion time for a job in the Postpone Queue is approaching nearer, it should be moved from the Postpone Queue to the Waiting Queue. In this study, a threshold of *(job completion time - job runtime estimate - 3hour)* is used to determine the latest timestamp when a postponable job should be moved to the Waiting Queue.

## 4.2 Experimental Configuration

CEIL is built on top of a base scheduler where the base scheduler enforces job priority and CEIL is integrated to improve system productivity by leveraging postponable jobs and fluctuating system usage. In the rest of the paper, we use *WFP* to denote the original scheduling policy deployed on Theta, whereas *CEIL* to denote the integration of CEIL with the base scheduler. All the scheduling policies use EASY Backfilling [18] to mitigate resource fragmentation.

**Table 4: Theta workload traces**

| Time period | Nodes | Users | Projects | Jobs |
|---|---|---|---|---|
| 07/01/2017-/07/31/2017 | 3624 | 148 | 41 | 7632 |
| 01/01/2018-/01/31/2018 | 4392 | 132 | 75 | 16184 |

**Table 5: Six workloads with various configurations**

| Workload | Trace | Postponable jobs% |
|---|---|---|
| Workload 1 | Theta in 07/2017 | 30% |
| Workload 2 | | 50% |
| Workload 3 | | 70% |
| Workload 4 | Theta in 01/2018 | 30% |
| Workload 5 | | 50% |
| Workload 6 | | 70% |

Event-driven, trace-based simulation with real workload traces is widely used for evaluating job scheduling efficiency in HPC. In our experiments, we use two one-month real workload traces from Theta, and the open-source scheduling simulator CQSim [1] is used for trace-based simulations.

Table 4 summarizes the Theta workload traces used in our trace-based simulation. We use two separate one-month logs to represent different stages of Theta, where the July 2017 workload represents an early production phase and the Jan 2018 workload represents a mature and stable phase. The July trace contains 7, 632 jobs, whereas the January trace contains 16, 184 jobs. We randomly mark some jobs in the log as the postponable jobs to synthesize a new workload. For each trace, we generate three workloads with different percentages of postponable jobs. In total, we evaluate six workloads which are summarized in Table 5. As some jobs are marked as postponable, we adopt a mechanism similar to that described in the deadline-based study [19] to specify the deadlines for the postponable jobs. The deadline for each postponable job is set as $max(24hr, runtime\ estimate * 10)$ [19]. According to the log analysis and application experiments listed in Section III, the application runtime is increased by 5.2%-21.1% when the system utilization is above 95% on Theta. Hence, for a given application, if it is allocated and executed when the system utilization is greater than 95%, we increase its runtime by a random value between 5.2% and 21.1%.

Four widely used metrics are used to evaluate the different scheduling methods:

- *Makespan* is defined as the total length of the schedule to complete all the jobs. It is used to measure scheduling throughput and system productivity.
- *Percentage of high utilization periods* is defined as the proportion of the time when the system utilization is higher than 95% in this study.
- *User wait time* measures the time period between a job's expected end time and its actual end time. For a postponable job, its expected end time is submitted by its user. For other jobs, job's expected end time is the job's requested runtime after job submission, indicating the user wants their job to be executed immediately— right after its submission. User wait time is the same as the conventional job wait time for regular

**Table 6: Comparison of system-level scheduling metrics.**

| Workload | Scheduling policy | Makespan(s) | Percentage of high utilization periods |
|---|---|---|---|
| Workload 1,2,3 | WFP | 2608532 | 21.81% |
| | CEIL | 2608497 | 0.06% |
| Workload 4,5,6 | WFP | 2684287 | 45.20% |
| | CEIL | 2684202 | 0.09% |

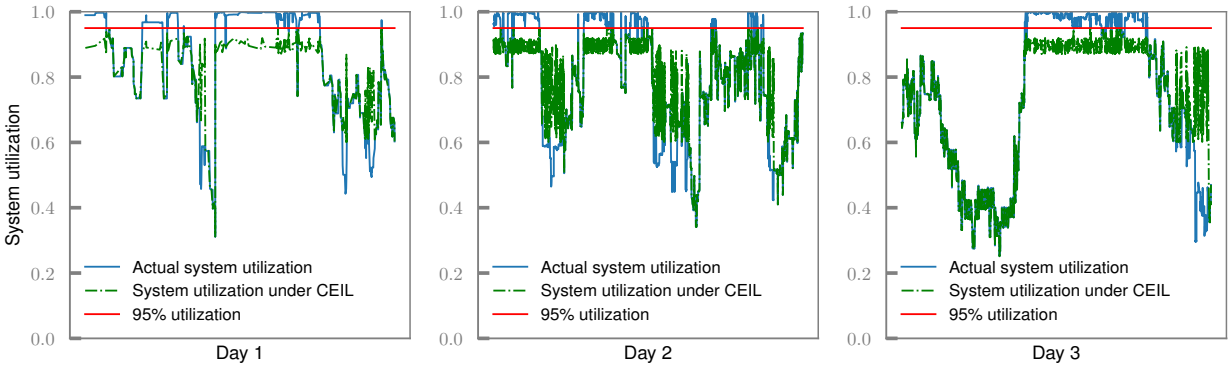
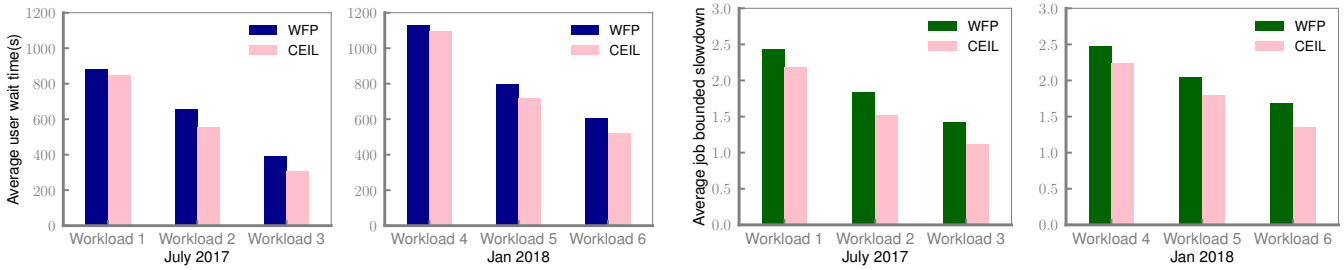



**Figure 4: Comparison of system utilization after applying CEIL and actual system utilization on three randomly selected days. The blue line shows the actual system utilization on Theta. The red line is corresponding to 95% utilization. The green line shows system utilization after applying CEIL with 50% postponable jobs (Workload 5).**




**(a) Average user wait time**

**(b) Average job bounded slow down**

**Figure 5: Comparison of CEIL and WFP**

jobs; however, it is slightly different from job wait time for postponable jobs. The difference lies in the part that user wait time doesn't consider the time period for which user is not actively waiting for job completion. As an example, if a user submits a postponable job with a deadline of 8 hours, as long as the job completes within eight hours, user wait time is zero.

- *job slowdown* is defined as the ratio of job response time (user wait time plus job runtime) to the job runtime. Usually, a small time bound (60 seconds) is used to avoid the value skewed by extremely small jobs [14]. This metric captures the factor that a long job can endure longer waiting time than a short job.

Note that the first two metrics are used to measure system-level scheduling performance, whereas the other two metrics are used to measure user satisfaction.

## 4.3 Preliminary Results

Table 6 presents system-level scheduling metrics: *makespan* and *percentage of high utilization periods*. Across all the six workloads tested in this study, CEIL is capable of maintaining the percentage of high utilization periods under 0.1% — significantly reduced from the original percentages of 20%-45%. For each workload, there is a very small percentage of high system utilization periods. This is due to the fact that Theta targets capability computing and its workload contains several capability jobs (e.g., whole-machine jobs).

Figure 4 shows comparison of system utilization after applying CEIL and actual system utilization on three randomly selected days. The figure clearly shows that CEIL is capable of "smoothing" resource utilization curve by delaying some postponable jobs from high utilization periods to low utilization periods. The resource utilization on a system typically fluctuates throughout the day. CEIL

leverages this utilization feature for reducing high system utilization without impacting the system throughput. Although CEIL can significantly reduce the percentage of high utilization periods, it does not impact system throughput (measured by *makespan*) for all the workloads tested in this study.

Figure 5 presents the average *user wait time* and *bounded slowdown*. CEIL can effectively reduce average *user wait time* by 12.5%-35.3%. Job bounded slowdown is reduced by 7.4%−20.2%. In general, the reduction is higher in the workload 1-3 than that in the workload 4-6. The reason is that workload 1-3 are from July of 2017 when Theta was in the early production period. As such, we notice the workloads in July of 2017 have more low utilization periods than the workloads in Jan of 2018. Theta is a production system serving the users all over the world, and at any time, not all users are in a hurry for their job completion. CEIL takes advantage of this observation: it smartly delays some postponable jobs without impacting user's satisfaction of the scheduling service; meanwhile, it reduces *user wait time* for the unpostponable jobs.

## 5 CONCLUSIONS

In summary, we have investigated whether there is a strong correlation between application runtime and system utilization. Our extensive log analysis and real application experiments indicate that a high system utilization can cause application runtime increase by 5.2% to 21.2%. Next, we have investigated a scheduling strategy CEIL to proactively avoid job allocation under high system utilization. Preliminary results have demonstrated that CEIL can effectively smoothen system utilization by leveraging the fluctuating resource demand pattern and the existence of postponable jobs. As a result, CEIL is capable of improving both system-level metrics (makespan and percentage of high utilization periods) and user-level metrics (user wait time and bounded slowdown).

There are some limitations in this work. First, the selection of 95% as the high utilization is specific to the Theta workload. Determining the tipping point via more experimental and analytical study remains as a future work. Second, CEIL explores the fact that some users are willing to delay the execution of their jobs and system utilization often exhibits diurnal pattern. CEIL is not suitable for the systems which are always heavily utilized. We hope that this work-in-progress study will motivate the community to rethink the design of HPC scheduling for tackling the job interference problem on shared dragonfly systems.

## 6 ACKNOWLEDGEMENT

This work is supported in part by US National Science Foundation grants CNS-1717763, CCF-1422009, CCF-1618776. This research used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357.

## REFERENCES

[1] CQSim. https://github.com/SPEAR-IIT/CQSim
[2] CRAY SLINGSHOTS BACK INTO HPC INTERCONNECTS WITH SHASTA SYSTEMS. https://www.nextplatform.com/2018/10/30/cray-slingshots-back-into-hpc-interconnects-with-shasta-systems
[3] MIMD Lattice Computation (MILC) Collaboration. http://physics.indiana.edu/~sg/milc.html
[4] NEK5000: a fast and scalable high-order solver for computational fluid dynamics. https://nek5000.mcs.anl.gov/
[5] NEKBONE: Apps for Thermal Hydraulics. https://cesar.mcs.anl.gov/content/software/thermal_hydraulics
[6] PBS Professional. http://www.pbsworks.com/
[7] Theta. https://www.alcf.anl.gov/theta
[8] William Allcock, Paul Rich, Yuping Fan, and Zhiling Lan. 2017. Experience and Practice of Batch Scheduling on Leadership Supercomputers at Argonne. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*.
[9] Abhinav Bhatele, Nikhil Jain, Yarden Livnat, Valerio Pascucci, and Peer-Timo Bremer. 2016. Analyzing network health and congestion in dragonfly-based supercomputers. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 93–102.
[10] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. 2013. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 41.
[11] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-run variability on Xeon Phi based Cray XC systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 52.
[12] Wayne W Daniel and Chad L Cross. 2012. *Biostatistics: a Foundation for Analysis in the health sciences*. Wiley Global Education.
[13] Yuping Fan, Paul Rich, William E Allcock, Michael E Papka, and Zhiling Lan. 2017. Trade-Off Between Prediction Accuracy and Underestimation Rate in Job Runtime Estimates. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 530–540.
[14] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. 1997. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1–34.
[15] Dror G Feitelson and Edi Shmueli. 2009. A case for conservative workload modeling: Parallel job scheduling with daily cycles of activity. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*. IEEE, 1–8.
[16] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J Wright, and Laxmikant V Kale. 2014. Maximizing throughput on a dragonfly network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 336–347.
[17] Michael Karo, Richard Lagerstrom, Marlys Kohnke, and Carl Albing. 2006. The application level placement scheduler. *Cray User Group* (2006), 1–7.
[18] Ahuva W. Mu'alem and Dror G. Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems* 12, 6 (2001), 529–543.
[19] Tchimou N'takpé and Frédéric Suter. 2017. Don't Hurry be Happy: a Deadline-based Backlling Approach. In *21st Workshop on Job Scheduling Strategies for Parallel Processing*.
[20] David Skinner and William Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 137–149.
[21] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P Sadayappan. 2002. Characterization of backfilling strategies for parallel job scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE, 514–519.
[22] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. 2010. Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–11.
[23] Xin Wang, Misbah Mubarak, Xu Yang, Robert B Ross, and Zhiling Lan. 2018. Trade-Off Study of Localizing Communication and Balancing Network Traffic on a Dragonfly System. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1113–1122.
[24] Xu Yang, John Jenkins, Misbah Mubarak, Robert B Ross, and Zhiling Lan. 2016. Watch out for the bully! job interference study on dragonfly network. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 750–760.
[25] Xu Yang, John Jenkins, Misbah Mubarak, Xin Wang, Robert B Ross, and Zhiling Lan. 2016. Study of intra-and interjob interference on torus networks. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 239–246.
[26] Xu Yang, Zhou Zhou, Sean Wallace, Zhiling Lan, Wei Tang, Susan Coghlan, and Michael E Papka. 2013. Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 60.
[27] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.