Scheduling Beyond CPUs for HPC

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

Paul Rich, William E. Allcock, Michael E. Papka
Argonne National Laboratory
Northern Illinois University
Lemont, IL
{richp,allcock,papka}@anl.gov

Brian Austin, David Paul
Lawrence Berkeley National Laboratory
Berkeley, CA
{baustin,dpaul}@lbl.gov

ABSTRACT

High performance computing (HPC) is undergoing significant changes. The emerging HPC applications comprise both compute- and data-intensive applications. To meet the intense I/O demand from emerging data-intensive applications, burst buffers are deployed in production systems. Existing HPC schedulers are mainly CPU-centric. The extreme heterogeneity of hardware devices, combined with workload changes, forces the schedulers to consider multiple resources (e.g., burst buffers) beyond CPUs, in decision making. In this study, we present a multi-resource scheduling scheme named BBSched that schedules user jobs based on not only their CPU requirements, but also other schedulable resources such as burst buffer. BBSched formulates the scheduling problem into a multi-objective optimization (MOO) problem and rapidly solves the problem using a multi-objective genetic algorithm. The multiple solutions generated by BBSched enables system managers to explore potential tradeoffs among various resources, and therefore obtains better utilization of all the resources. The trace-driven simulations with real system workloads demonstrate that BBSched improves scheduling performance by up to 41% compared to existing methods, indicating that *explicitly optimizing multiple resources beyond CPUs is essential for HPC scheduling*.

KEYWORDS

High performance computing (HPC); multi-resource scheduling; burst buffers; multi-objective optimization

ACM Reference Format:
Yuping Fan, Zhiling Lan, Paul Rich, William E. Allcock, Michael E. Papka, and Brian Austin, David Paul. 2019. Scheduling Beyond CPUs for HPC. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325401

1 INTRODUCTION

The exponential growth in computing power has enabled high-performance computing (HPC) systems to attack scientific problems that are much larger and more complex. HPC applications have diverse resource requirements. For them, CPU is not necessarily the main resource determining the required performance, but the allocation with respect to other resources like I/O and network bandwidth becomes more critical. A typical example is *data-intensive applications*. These applications have extremely high demand for storage systems. As the growth in computing power continues to outpace the increase in network bandwidth between compute nodes and parallel file system (PFS), PFS fails to rapidly consume bursty data produced by HPC applications. As such, production supercomputers are deployed with *burst buffers* to bridge the performance gap between compute nodes and PFS. Burst buffer is an intermediate storage layer positioned between compute nodes and storage systems. It is typically built from solid-state drive (SSD), offering one to two orders of magnitude higher I/O bandwidth than PFS. Cori [2] at National Energy Research Scientific Computing Center (NERSC) and Trinity [8] at Los Alamos National Laboratory (LANL) are deployed with shared burst buffers.

As burst buffers are incorporated into HPC systems, it is crucial for HPC schedulers to schedule user jobs based on their CPU as well as burst buffer demands. Note that *this study targets at HPC schedulers that are responsible for allocating user jobs onto compute nodes and other system-level schedulable resources, e.g., burst buffers.* The terms CPU and compute node are used interchangeably in this paper. The well-known schedulers in HPC include Slurm, Moab/TORQUE, PBS, and Cobalt [1, 4, 5, 24]. Depending on the site mission, HPC facilities deploy different scheduling policies to achieve certain goals [10]. For instance, first come, first served (FCFS) with EASY backfilling is a default scheduling policy deployed at many production systems [29]. Despite the use of different scheduling policies, a common goal for HPC scheduling is *to optimize resource utilization.* Existing HPC schedulers are mainly CPU-centric. They often disregard diverse resource requirements and make scheduling decisions solely based on the application's processor footprint. Such a CPU-centric scheduling can easily result in poor application performance and waste of system resources.

Slurm is a well-known scheduler that supports burst buffer scheduling [23]. Slurm allocates the jobs from the waiting queue in sequence until either CPU or burst buffer is exhausted. We denote it as *naive method* in this study. This approach has a limited efficiency: the depletion of one resource can prevent the queued jobs from allocation, causing under-utilization of the other resource. Two optimization approaches for solving multi-resource scheduling problems may be applicable for co-scheduling CPU and burst buffer. One approach is to optimize utilization of one resource and treat other resources as constraints (denoted as *constrained method*) [31, 36, 38]. Another approach is to combine utilizations of multiple resources into one objective by a weighted sum (denoted as *weighted method*) [21, 22, 33]. Both optimization methods convert a

Table 1: An illustrative example of scheduling multiple resources using different scheduling methods.

(a) Job waiting queue

Job	Nodes	Burst Buffers (TB)
J1	80	20
J2	10	85
J3	40	5
J4	10	0
J5	20	0

(b) The scheduling decisions made by different scheduling methods

Solution	Selected Jobs	Node Utilization	Burst Buffer Utilization	Naive Method	Constrained Method	Weighted Method	Bin Packing	Pareto Set
1	J1, J4	90%	20%	✔				
2	J1, J5	100%	20%		✔	✔	✔	✔
3	J2, J3, J4, J5	80%	90%					✔

multi-resource scheduling problem into a single-objective optimization problem. Such a conversion leads to the loss of a prominent characteristic of multi-resource scheduling, i.e., trade-offs between competing resources. *Bin packing* is discussed in the literature to improve resource utilization for cluster scheduling [17, 30]. It models machines as bins and tasks as balls. Balls of different volumes are packed into bins of certain capacities iteratively and the goal is to minimize the number of bins used. This simple heuristic selects jobs in a one-by-one manner, which may miss the best job combination that maximizes resource utilization.

An Illustrative Example: Here we give a simple example to show the limitations of the existing scheduling methods for scheduling CPU and burst buffer on HPC. Consider a system with 100 nodes and 100TB of burst buffers. Five jobs are in the queue, each having different resource demands as shown in Table 1(a).

Table 1(b) compares the scheduling results of different methods. A naive method selects J1 and backfills J4 (explained in Section 2.1) for execution, resulting in node utilization of 90% and burst buffer utilization of 20%. Such a scheduling wastes 80TB of burst buffers and prevents other jobs from being scheduled. A constrained method may optimize node utilization under the constraint of the burst buffers. A weighted method may use a linear combination of node utilization with 80% weight and burst buffer utilization with 20% weight as the objective. A bin packing method may pick jobs with the maximum dot product between the demands of the job and the remaining amount of resource iteratively. The constrained, the weighted and the bin packing methods select J1 and J5 for execution, achieving node utilization of 100% and burst buffer utilization of 20% (Solution 2). While these methods improve node utilization, they still leave 80% of the burst buffers wasted.

All these methods overlook an alternative solution: the selection of J2-J5 for resource allocation by skipping J1 (Solution 3). Solution 3 achieves significantly higher burst buffer utilization, while slightly lowering node utilization as compared to Solution 2. This simple example highlights the importance of identifying a *Pareto set*[1], each solution in the Pareto set representing a tradeoff among different objectives (i.e., the selection of different resources).

In this study, we present a multi-resource scheduling scheme denoted as *BBSched* that allocates multiple resources to user jobs based on their resource demands. *Distinguishing from existing methods, BBSched aims to optimize the utilization of multiple resources by providing a Pareto set for decision making.* There are three *key obstacles* to overcome in the design of BBSched. First, the design has to be practical in the sense that it can make rapid scheduling decisions. Current HPC systems typically require a scheduler to respond in 15-30 seconds [9, 38]. Second, an efficient design has to improve system-related performance with minimal impact on site policies. Finally, HPC is dynamically evolving such that systems are constantly expanded with new resources. Hence, the scheduler is expected to be extensible to embrace emerging resources.

To tackle the above obstacles, several techniques are explored for the BBSched design. First, BBsched is developed as a plugin to existing HPC schedulers (denoted as *base schedulers*) to preserve job priority according to a site's policy. Unlike the traditional one-by-one job selection used in the conventional scheduling, BBSched leverages a window-based scheduling approach to dispatch a set of jobs from the front of job waiting queue. Such a window-based design aims to maintain the job ordering given by the base scheduler. Second, jobs are selected from the window for resource allocation, with the objective to optimize resource utilization. We formulate the multi-resource scheduling problem into a multi-objective optimization (MOO) problem. Contrary to a single-objective optimization, our MOO formulation simultaneously optimizes the utilizations of multiple resources and returns a Pareto set. The Pareto set provides a set of optimal solutions which enables system managers to make a scheduling decision by considering the tradeoffs among different optimal solutions. Considering that MOO is NP-hard [26], we explore a genetic algorithm as the MOO solver for meeting the rigid time requirement.

We evaluate BBSched by means of extensive trace-based simulations with real workload traces collected from Cori at NERSC and Theta [7] at Argonne Leadership Computing Facility (ALCF). Additionally, we generate a series of workloads based on the real traces to stress various resource usages. The goal is to extensively evaluate BBSched under various scenarios, especially under the cases of resource confliction and saturation. A series of experiments are conducted to compare BBSched with existing methods (naive, constrained, weighted, and bin packing methods) on scheduling CPU and burst buffer. The results show that BBSched is capable of improving resource utilization by up to 20% and reducing average job wait time by up to 41%.

Furthermore, we present a case study to show BBSched can be easily extended to schedule additional resources beyond CPUs and burst buffer. The preliminary results clearly indicate that BBSched outperforms existing methods in terms of both system-level and user-level scheduling metrics. *This demonstrates that explicitly optimizing all resources is crucial to multi-resource scheduling.*

The remainder of this paper is organized as follows. We start by introducing background and related work in Section 2, including HPC scheduling, burst buffer, and multi-resource scheduling. Section 3 describes our design. The experimental results of scheduling CPU and burst buffer are presented in Section 4. A case study of incorporating more resources in BBSched is examined in Section 5. Finally, we conclude the paper in Section 6.

[1]Pareto set is a set of non-dominated solutions, being chosen as optimal, if no objective can be improved without sacrificing at least one other objective [32]. For the example shown above, the Pareto set contains Solution 2 and 3.

2 BACKGROUND AND RELATED WORK

2.1 HPC Scheduling

System-level HPC scheduling, also known as batch scheduling, is responsible for assigning jobs to resources according to site policies and resource availability. It targets on scheduling compute nodes along with other system-level resources. This is different from core-level application scheduling or task scheduling that is typically handled by operating systems. Well-known schedulers in HPC include Slurm, Moab/TORQUE, PBS, and Cobalt [1, 4, 5, 24]. When submitting a job, a user is required to provide two pieces of information: resources required by the job and runtime estimate [15]. The jobs are stored and sorted in the waiting queue based on a site's policy. In the past, a number of scheduling policies have been proposed, and one of the widely used policies is FCFS, which sorts the jobs in the order of their arrivals. At ALCF, to support the mission of running large-scale capability jobs, a utility-based scheduling policy, named WFP, is deployed which periodically calculates a priority increment for each waiting job [10, 39]. EASY backfilling is a commonly used strategy to enhance system utilization, where subsequent jobs are allowed to skip ahead under the condition that they do not delay the job at the head of the queue [29].

In this study, we denote the above schedulers that enforce job priority according to a site's policy as base schedulers. BBSched can be used along with these base schedulers, for optimizing utilization of multiple resources, without unnecessary impact of job priority posed by the base scheduler.

2.2 Burst Buffer

HPC systems are facing the challenge of ever-growing gap between compute power and I/O performance. Bridging this gap becomes increasingly critical with the increasing data-intensive applications. I/O behavior of data-intensive applications is characterized by intense bursts of data access [28]. Burst buffers, an intermediate storage layer between compute nodes and PFS, are designed to absorb bursty I/O data effectively. They are typically built from SSD, providing significantly higher bandwidth and lower latency than PFS. A burst buffer can be either attached to compute nodes as a local resource or configured as a global resource shared by compute nodes. Cori at NERSC and Trinity at LANL adopt shared burst buffers; Theta at ALCF and Summit [6] at Oak Ridge Leadership Computing Facility (OLCF) are equipped with local SSDs.

Existing studies on burst buffer scheduling are mainly at application level or at I/O server level. Little work has been done at system scheduling level. Slurm supports co-scheduling of CPU and burst buffer; however, it lacks optimization for CPU and burst buffer. This study will address the co-scheduling of CPU and burst buffer with the objective of optimizing the utilization of both resources.

2.3 Multi-Resource Scheduling

Considerable research exists in exploring optimization methods for multi-resource scheduling. Constrained optimization is a common optimization method. For example, Wallace et al. addressed a power-aware scheduling problem by optimizing node utilization with a power limit [36]; Rao et al. examined the problem of reducing the total electricity cost while guaranteeing the quality of service in datacenters [31]; Xu et al. presented an energy-aware scheduling framework which maximizes power consumption at off-peak time with the constraint of nodes [38]. Weighted sum is another widely adopted optimization method. For example, Ren et al. converted an energy-aware cluster scheduling problem to optimization of the weighted sum of energy cost and fairness [33]; Huang et al. treated the problem of multi-resource allocation in geo-distributed clusters as the minimization of the sum of the time spent on network transfer and computation [21]; Jakob et al. formulated a workflow scheduling problem in grid into optimizing a weighted sum of execution time, costs, and resource utilizations [22]. The above studies basically convert multi-objective optimization problem into a single objective optimization problem through a weighted combination or a constrained approach. In contrast to these studies, we formulate the problem as a MOO problem and use a multi-objective genetic algorithm for solving the MOO problem. Unlike the constrained or the weighted optimization that only provides a single solution optimized for a first-class objective or a weighted sum of the objectives, our method is capable of optimizing multiple objectives by providing a Pareto set for decision making.

Multi-resource cluster scheduling is an active research area that has received much attention in the past years. Cluster schedulers typically emphasize fair resource allocation [12, 16, 17, 25, 37], which distracts considerably from the goal of HPC scheduling, i.e., high resource utilization. For example, dominant resource fairness (DRF) allocates multiple resources satisfying strategy-proof, envy-freeness, sharing incentive and pareto-efficiency [16]. Pareto efficiency is defined as increasing the allocation of a user should not decrease the allocation of at least another user [16], which is different from the Pareto set targeted in this work. In practice, production cluster schedulers also prefer fairness as their primary scheduling goal [11, 20, 34, 35]. However, fairness and utilization are conflicting goals and aggressively using fair sharing can hurt cluster utilization [18]. In contrast, HPC systems are designed to run large jobs and prefer users running large jobs [10]. Therefore, fair sharing is not a concern in HPC scheduling. Some other cluster schedulers focus on individual job performance, e.g., lowering job completion times [30] and responding timely to latency-critical services [40]. Due to the heterogeneous nature of cluster machines, job completion times vary when jobs run on different machines. This assumption does not hold on HPC systems. Moreover, to ensure timely response to latency-critical services, clusters have to reserve resources for unexpected load spikes, leading to low resource utilization [40].

A few studies in multi-resource cluster scheduling evaluated the negative impact of fairness on utilization and made tradeoffs between fairness and utilization [17, 18]. Owing to substantial job arrival rate, cluster schedulers have to respond in seconds or shorter. Hence, they seek quick but greedy methods for scheduling. For instance, Grandl et al. have explored the potential of using multi-dimensional bin packing to improve resource utilization for multi-resource clusters [17]. Although bin packing algorithms are capable of making timely scheduling decisions, they allocate jobs in a one-by-one manner based on individual job information and therefore lead to less desirable performance as compared to the MOO approach adopted in this study which jointly considers resource requirements of multiple jobs. Altruistic scheduling improves resource utilization as well as meets fair-share guarantees by redistributing leftover resources, i.e., fractions of allocated resources [18]. This is inapplicable to HPC scheduling because the sizes of HPC jobs are fixed throughout their execution.

3 METHODOLOGY

In this section, we present BBSched for HPC scheduling under multiple resource constraints. As shown in Figure 1, BBSched is built as a plug-in to a base scheduler which enforces job priority according to a site's policy. BBSched consists of two components: *a window-based scheduling and a scheduling optimization scheme*. BBSched begins with a window-based scheduling with the aim to balance scheduling performance and enforcing site policies (Section 3.1). The jobs in the window are selected to execute for optimizing resource utilizations (Section 3.2). The optimization process first formulates the multi-resource scheduling problem into a multi-objective optimization (MOO) problem, which optimizes node utilization and burst buffer utilization (Section 3.2.1). Given that this MOO problem is NP-hard, we then develop an efficient meta-heuristic for problem-solving (Section 3.2.2). We discuss how to select the parameters in the solver in Section 3.2.3. The final solution is chosen from multiple solutions by considering tradeoffs among multiple resource usages (Section 3.2.4). Finally, we analyze the computational complexity of BBSched in Section 3.3.

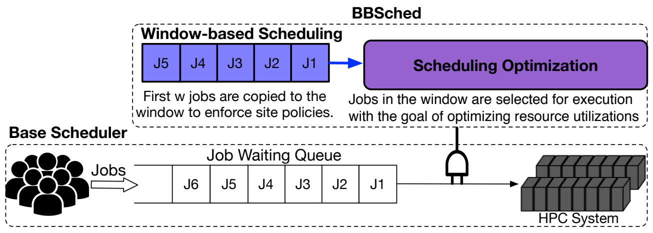

BBSched
Window-based Scheduling
J5
J4
J3
J2
J1
Scheduling Optimization
First w jobs are copied to the window to enforce site policies.
Jobs in the window are selected for execution with the goal of optimizing resource utilizations
Base Scheduler
Job Waiting Queue
Jobs
J6
J5
J4
J3
J2
J1
HPC System

Figure 1: The overview of BBSched.

3.1 Window-Based Scheduling

Rather than allocating jobs one by one from the front of the waiting queue, we adopt the window-based scheduling which allocates multiple jobs from a window at the front of the waiting queue [38]. In doing so, we balance the goals of optimizing system metrics and enforcing site policies. Note that jobs with dependencies are allowed to enter the window only if all the dependencies have been completed. This restriction keeps dependent jobs in order and preserves the priority of jobs with dependencies. Window size is a tunable parameter. The selection of window size is dependent on site policy and workload characteristics. A larger window size means more jobs are considered for optimization; however, it also means less preservation of the original job order. System managers may choose a window size according to their preference for more optimization or more preservation of job order. In addition, the window size could be dynamically adjusted in response to system status. Job queue length often changes. For instance, it is typically longer during workdays and is shorter during weekends. In this study, we use a static window size.

An issue with the window-based scheduling is *job starvation*, meaning that a job may stay in the window without being selected to execute. To prevent job starvation, we define an upper bound for the number of iterations that a job can stay in the window. Once a job passes the bound (e.g., 50), it must be selected to run.

3.2 Scheduling Optimization

3.2.1 ***Multi-Objective Optimization (MOO) Formulation:*** The optimization process determines which jobs are selected from the window to execute so that node utilization and burst buffer utilization are maximized. To achieve system performance goal, we formulate the multi-resource scheduling problem into the following MOO problem.

Suppose a system has N nodes and burst buffers of total B GB. Assume that upon a scheduling invocation, the amounts of nodes and burst buffers being used are N_{used} and B_{used} respectively. Suppose $J = \{J_1, \ldots, J_w\}$ is a set of w jobs in the scheduling window: job J_i requiring n_i nodes and b_i GB of burst buffers.

The scheduling problem can be transformed into the following MOO: *to determine a finite set of Pareto solutions* $\boldsymbol{X}$; each Pareto solution $\boldsymbol{x} \in \boldsymbol{X}$ is represented by a binary vector $\boldsymbol{x} = [x_1, \ldots, x_w]$, such that $x_i = 1$ if J_i is selected to execute and $x_i = 0$ otherwise. A Pareto solution optimizes the following two objectives:

(1) maximize node utilization: $f_1(\boldsymbol{x}) = \sum_{i=1}^{w} n_i \times x_i$

(2) maximize burst buffer utilization: $f_2(\boldsymbol{x}) = \sum_{i=1}^{w} b_i \times x_i$

Formally, the problem can be formulated as:

$$\max \quad (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}))$$

$$\text{s.t.} \quad \sum_{i=1}^{w} n_i \times x_i \le N - N_{used}, \quad x_i \in \{0,1\}$$

$$\sum_{i=1}^{w} b_i \times x_i \le B - B_{used}, \quad x_i \in \{0,1\}$$

where the constraints guarantee that assigned resources do not exceed available nodes and burst buffers in a system.

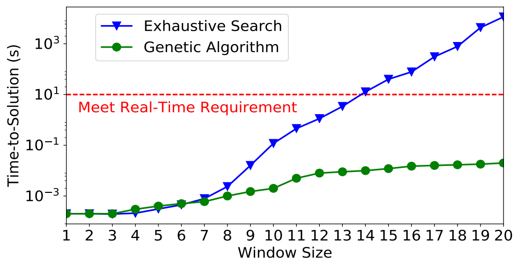
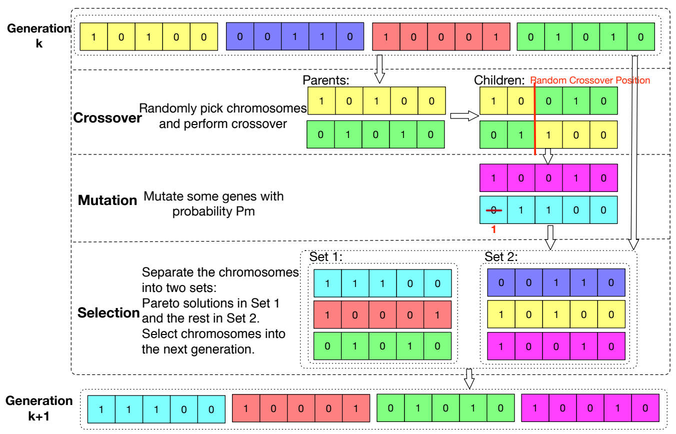


Figure 2: Impact of window sizes on average solution time. Figure 2 and 4 were conducted with first 1000 jobs from a Theta workload (see Table 2). Solutions above the red dash line do not meet the time requirement of HPC scheduling.

3.2.2 ***MOO Solver:*** The above MOO problem is NP-hard. To find all solutions, one has to exhaustively examine 2^w possible solutions and compare them to determine a Pareto set. As the window size w increases, the number of possible solutions as well as the time-to-solution increases exponentially (see Figure 2). Current HPC systems typically require a scheduler to respond in 15-30 seconds [9, 38]. To achieve fast decision making, we need a rapid solver to solve the MOO problem. In this study, we explore a multi-objective genetic algorithm [27] to solve the MOO problem. *This genetic-based algorithm approximates the true Pareto set iteratively,* so it requires much less time. It can be accelerated by leveraging parallel processing [13]. A genetic algorithm attempts to mimic natural selection: the population size is a constant P; weak chromosomes are extinct by natural selection, while strong chromosomes survive and pass their genes to future generations.



Generation k
Crossover
Randomly pick chromosomes and perform crossover
Parents:
Children:
Random Crossover Position
Mutation
Mutate some genes with probability Pm
Selection
Separate the chromosomes into two sets: Pareto solutions in Set 1 and the rest in Set 2. Select chromosomes into the next generation.
Set 1:
Set 2:
Generation k+1

Figure 3: An example of the evolution process to solve the MOO problem. MOO solver maintains a population of candidate solutions (4 chromosomes). A chromosome consists of 5 genes, where each gene represents the selection of the job at a specific location in the window and encodes as a binary number: 1 (selected) or 0 (not selected).

Figure 3 illustrates the process of solving the MOO problem. The first generation is initialized randomly. Generations are evolved iteratively via crossover, mutation, and selection operations. The crossover operation generates two children by randomly selecting two parents from the previous generation and swapping genes of parents at a random position. The mutation is used to introduce diversity and to prevent our solver from trapping in local optima. With a low probability p_m, the genes of the children are bit flipped. A high mutation rate leads to a random search and results in poor solutions. The selection operation constructs a new generation by carrying over good chromosomes to the next generation. Specifically, we separate the chromosomes into two sets: Pareto solutions in Set 1 and the rest in Set 2. A solution is chosen as a Pareto solution, if improving one of its objectives would deteriorate at least one other objective. If Set 1 has less than P chromosomes, all the chromosomes in Set 1 are passed to the next generation and then chromosomes in Set 2 (newer chromosomes have higher priorities). If Set 1 has more than P chromosomes, we select those with newer ages. Upon an evolution to new generation, the ages of chromosomes are increased by 1.

When the number of generations reaches a predefined threshold G, the above iterative process stops and the chromosomes in Set 1 in the final generation form a Pareto set.

3.2.3 ***Parameter Selection:*** The solver contains three parameters: number of generations (G), population size (P), and mutation probability (p_m). According to the literature, p_m is normally very low, i.e., less than 0.1% [14, 19]. In our experiments, varying p_m has negligible effect on approximation accuracy. A larger value of G and P means exploring a larger search space for optimization but more time to solve the problem. Therefore, the selection of G and P is a trade-off between performance and time-to-solution. The widely adopted metric *generational distance (GD)* is used to measure the accuracy of the solutions. GD is defined as:

$$GD(S) = avg_{u \in S}(min_{v \in S^*}(dist(u, v)))$$

where S is the solution set obtained by our MOO solver; S^* is the true Pareto set. GD computes the average distances between our solution and its nearest true Pareto solution. The smaller the GD is, the better the performance is.

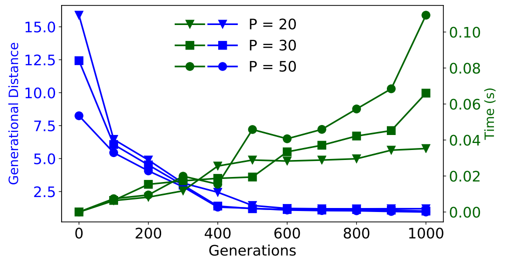


Figure 4: Impact of varying G and P parameters.

Figure 4 illustrates an example of the GD value and the time-to-solution as G and P vary. Clearly, as G increases, GD decreases and time-to-solution increases. For GD, the most significant improvement is between 0 and 500 generations and the improvement slows down after 500 generations. We also notice that increasing P leads to a decrease in GD and increase in time. This example suggests that our MOO solver is capable of achieving accurate approximations with minimal overhead (less than 0.2 second). For the workloads investigated in Section 4.1, setting $G = 500$ and $P = 20$ offers the best tradeoff between accuracy and time-to-solution.

3.2.4 ***Decision Making:*** The output of the solver is a Pareto set, and a decision maker needs to select one preferred solution. Different HPC facilities may have different site policies and scheduling priorities. *System managers may use a site-specific metric for selecting a preferred solution out of the Pareto set.* In this study, we use the following rule. It first chooses the solution that maximizes node utilization, and in case of multiple such solutions, it selects the solution containing the jobs at the front of the window so as to preserve the original job order. Next, it compares the solution with other Pareto solutions for tradeoff analysis. The preferred solution is replaced by another solution if the improvement on the burst buffer utilization is more than 2x of the loss of the node utilization. If more than one such solutions exist, the solution with the maximum improvement is chosen.

The decision making may be *adaptive*, such that system managers dynamically adjust their selection policy according to scheduling performance and user response. This adaptive decision making is out of the scope of this work and is a topic of our future work.

3.3 Complexity Analysis

The window-based mechanism takes a constant time $O(1)$. The scheduling optimization iterates G times. In each iteration, crossover, mutation, and selection are operated on P chromosomes. In total, the optimization requires $O(G \times P)$ operations in the worst cases. Therefore, the time complexity of our design is $O(G \times P)$. This cost can be further lowered via parallel processing of the MOO.

4 EVALUATION

In this section, we evaluate BBSched through extensive trace-based simulation using real workload traces collected from production systems. We describe the two real workload traces collected from Cori and Theta and the eight synthetic workloads with various burst buffer demands derived from the real traces (Section 4.1). We then list the system- and user-centric metrics for scheduling evaluation (Section 4.2) and the multi-resource scheduling methods for comparison (Section 4.3). Finally, we quantify BBSched's performance improvements over existing methods (Section 4.4).

Table 2: Overview of Cori and Theta workloads.

	Cori	Theta
Location	NERSC	ALCF
Scheduler	Slurm	Cobalt
System Types	Capacity computing	Capability computing
Compute Nodes	12,076 (2,388 Haswell; 9,688 KNL)	4,392 (4,392 KNL)
Aggregated Memory	1,304.5TB	913.5TB
Shared Burst Buffer	1.8PB	1.26PB (projected)
Trace Period	Apr. 2018 - Jul. 2018	Jan. 2018 - May. 2018
Number of Jobs	2,607,054	70,507
BB Data Source	Slurm log	Darshan log
BB Range	[1GB, 165TB]	[1GB, 285TB]

4.1 Workload Traces

Table 2 summarizes the real workload traces used in this study. They represent two typical HPC workloads: one for *capacity computing* and the other for *capability computing*. Note that both traces do not include job dependency information and therefore we suppose all jobs are independent in our experiment. The first workload is a four-month job log on Cori from April to July in 2018. Cori is deployed with a 1.8PB Cray Data Warp burst buffers. It uses Slurm for job scheduling. The Slurm log records a number of information per user job which include the requested number of nodes, the requested burst buffer size, job runtime estimate, job submit time, etc. Besides few extremely large requests (165TB), the burst buffer requests are in the range of [1GB, 65TB]. Among all the jobs, 0.618% of jobs request burst buffers and 0.204% of jobs request more than 1TB burst buffers. Burst buffers on Cori can be either assigned to individual jobs or assigned to users as persistent reservations. One-third of burst buffers on Cori are reserved persistently and their lifetimes are independent of jobs.

The second workload is a half-year job log on Theta from January to May in 2018. While Theta is currently not deployed with any shared burst buffer, we enhance the trace with burst buffer requests by assuming there was a shared burst buffer of 2.16PB. This assumption is based on the ratio of aggregated memory to the total burst buffer volume on Cori and the aggregated memory on Theta. The trace from Theta contains all the necessary information except for the requested burst buffer size. To address this problem, we use a corresponding *Darshan* [3] trace to extract the amount of data moved between PFS and nodes and consider this amount to be the potential burst buffer requests. In the half-year workload, 40% of jobs on Theta have Darshan I/O recording. 17.18% of the jobs have more than 1GB data transferred and we set the amount of transferred data as the corresponding job's burst buffer request. The requested burst buffer sizes are in the range of [1GB, 285TB].

One issue with both traces is that burst buffers are not heavily used. There are two possible reasons. First, some jobs have burst buffer demands but are currently not recorded in the system logs. Second, burst buffer is a relatively new resource for users. As users are getting more exposure to it, we expect significant increases in requests for burst buffers. Having this log limitation in mind, in addition to the two original workloads, we create eight synthetic workloads, four workloads (S1-S4) for each machine, by expanding the percentage of jobs requesting burst buffers to 50% (S1 and S3 workloads) and 75% (S2 and S4 workloads). In these synthetic workloads, the assigned burst buffer request is randomly selected from the original burst buffer requests in a certain range. S1 and S2 select requests from original requests greater than 5TB, while S3 and S4 choose from requests greater than 20TB. Figure 5 shows the distributions of burst buffer requests for all the ten workloads (two systems, each with five workloads). As we can see, S3 and S4 workloads have larger burst buffer requests than S1 and S2. S1 and S2 have similar distributions, but more jobs in S2 request burst buffers. The similar pattern is observed in S3 and S4.

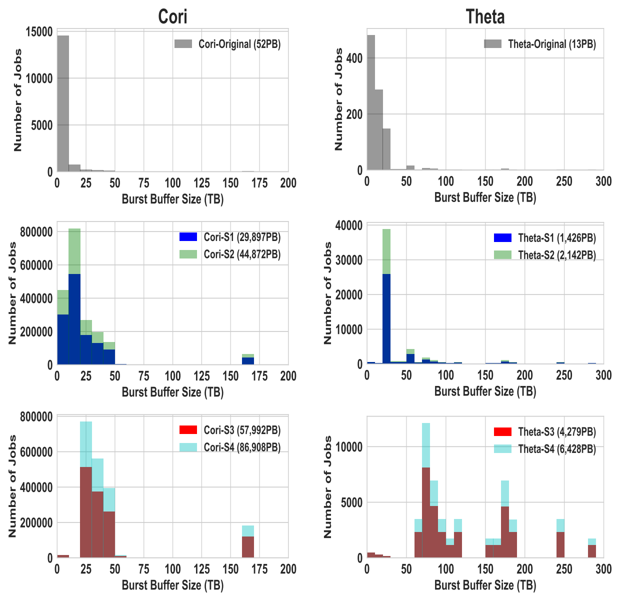


Figure 5: Histograms of burst buffer distributions on Cori (left) and Theta (right). The bin size is 10TB. The number in the parenthesis is the aggregated volume of requested burst buffers in each workload.

4.2 Evaluation Metrics

There are two classes of metrics for evaluating job scheduling: one is to evaluate system-level performance *from the perspective of system managers*, and the other is to evaluate the quality of service *from the perspective of users.* In our experiments, we use four well-established metrics:

- **Node usage** [2] measures the ratio of the used node-hours for useful job execution to the elapsed node-hours.
- **Burst buffer usage** measures the ratio of the used burst buffer hours to the elapsed burst buffer hours.

[2]Usage is a metric measuring resource utilization without considering availability.

- **Job wait time** measures the interval between job submission to job start time.
- **Job slowdown** measures the ratio of the job response time (job runtime plus wait time) to its actual runtime. It is used to gauge the responsiveness of a system. We filter out abnormal jobs in calculating average slowdown, because many abnormal jobs end abruptly at beginning of execution leading to extremely high slowdowns.

Note that the first two metrics are system-level performance metrics, whereas the last two are user-level performance metrics.

In the rest of the paper, the 1st half month data is used to "warm up" the system and the last half month data is used to "cool down" the system. We present the results in the remaining months.

4.3 Scheduling Methods

We compare eight multi-resource scheduling methods:

- **Baseline**: Baseline represents the naive method for multi-resource scheduling (e.g., the one adopted in Slurm for burst buffer scheduling).
- **Weighted**: This method aims to maximize a weighted combination of multiple objectives. The weights are site tunable parameters and system administrators can adjust them based on the importance of different resources. For this method, the weights of node utilization and burst buffer utilization are set to 50% and 50% respectively. These weights present the case where CPU and burst buffer are considered equally important.
- **Weighted_CPU**: In this weighted method, the weights of node utilization and burst buffer utilization are set to 80% and 20% respectively. These weights present the case where CPU is considered more important.
- **Weighted_BB**: In this weighted method, the weights of node utilization and burst buffer utilization are set to 20% and 80% respectively. These weights present the case where burst buffer is considered more important.
- **Constrained_CPU**: It aims to maximize node utilization under the constraints of burst buffers.
- **Constrained_BB**: It aims to maximize burst buffer utilization under the constraints of node utilization.
- **Bin_Packing**: This method is analogous to the bin packing method used in [17]. We compute alignment score (a dot product between the vector of machine's available resources and the job's requested resources) for jobs in the window and then allocate jobs with highest alignment score recursively until the machine cannot accommodate any further jobs.
- **BBSched**: The scheduling scheme presented in this work. By default, we set the window size to 20, the number of generation to 500, the population size to 20, and the mutation probability to 0.05%.

In our experiments, each of these multi-resource scheduling methods runs along with a base scheduler. With the Cori workloads, FCFS is used as the base scheduling method. With the Theta workloads, WFP (described in Section 2.1) is used as the base scheduling method. To make fair comparisons, we use the same window size for all methods. In addition, all the methods use EASY backfilling [29] to mitigate resource fragmentation.

4.4 Results

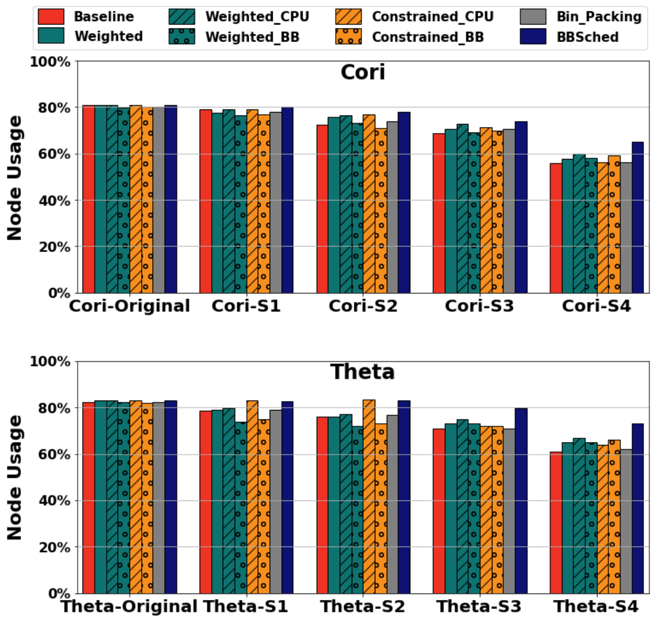


Figure 6: Comparison of node usage on Cori traces (top) and Theta traces (bottom). The performance of the methods favoring CPU resource (Weighted_CPU and Constrained_CPU) are presented by the bars with lines on them, while the performance of the methods biasing towards burst buffers (Weighted_BB and Constrained_BB) are presented by the bars with cycles on them.

Impact on Node Usage: Figure 6 compares node usage obtained by different scheduling methods. Among all the methods, BBSched yields the best node usage for 7 out of 10 workloads. The most noticeable gains happened on Theta-S4 and Cori-S4. In the other 3 out of 10 workloads, namely Cori-Original, Theta-S1, and Theta-S2, Constrained_CPU method achieves the best node usage. However, the performance difference between Constrained_CPU and BBSched is negligible. When burst buffer is abundant, Constrained_CPU method obtains good performance on node usage because it only optimizes node usage, whereas BBSched has to make trade-offs between node usage and burst buffer usage. When burst buffer becomes scarce, BBSched outperforms Constrained_CPU method in terms of node usage. This is because burst buffer shortage has become the bottleneck of allocation of CPU resource to jobs with burst buffer requests. BBSched considers both resources and therefore eases the burst buffer contention between jobs and mitigates resource wastage on CPU. We also notice that Weighted_BB and Constrained_BB have very poor performance on node usage with the worst reductions in 6.11% and 4.84% respectively compared with the baseline. Considering that Weighted_BB and Constrained_BB favor burst buffers, they prioritize jobs that can make better use of burst buffers, and as a result, waste CPU resources.

Impact on Burst Buffer Usage: Figure 7 shows burst buffer usage obtained by the eight scheduling methods. It is clear that all the methods except Constrained_CPU improve burst buffer usage. The unsatisfactory performance obtained by Constrained_CPU method is because this method puts node usage as its sole optimization objective and ignores burst buffer usage. This result clearly shows that exploiting job's complementary resource demands is crucial for multi-resource scheduling. Although BBSched, Weighted and Weighted_BB consider both node and burst buffer utilizations in scheduling, weighted methods can only find one solution, while BBSched can produce multiple solutions. Therefore, BBSched is more likely to find a better solution from multiple solutions, leading to higher resource utilization. In summary, BBSched yields the best performance on burst buffer usage for all the workloads, and the performance improvement is as high as 15.46% over the baseline.

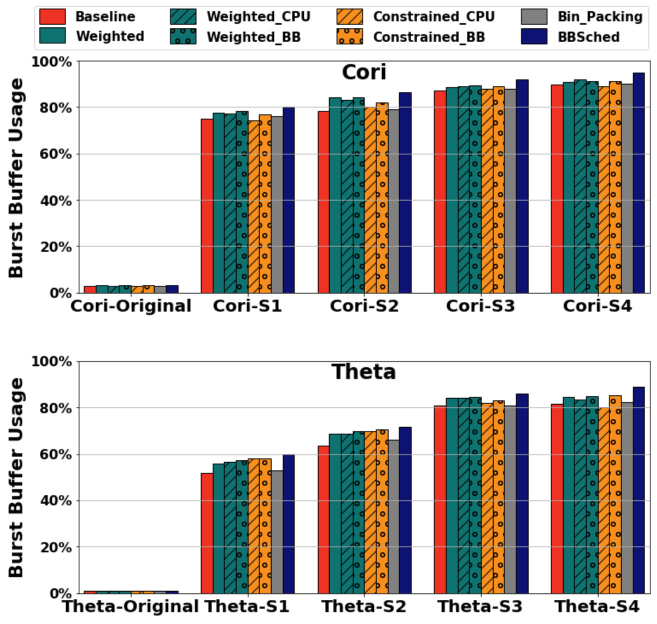


Figure 7: Comparison of burst buffer usage on Cori traces (top) and Theta traces (bottom).

Node usage and burst buffer usage are correlated. Under the scenarios that burst buffer usages are under 80% (Theta-Original, Theta-S1, Theta-S2, Cori-Original, and Cori-S1), optimization methods can keep the node usage around 80%. But in the remaining scenarios with heavy burst buffer requests, we observe the noticeable drops in node usage in comparison to the original workloads. This phenomenon indicates that heavy burst buffer requests cause wastage in node resources. Among all the optimization methods, Constrained_CPU suffers the most from the increase in burst buffer requests. The node usage drops more than 15% from Cori-S3 to Cori-S4 and from Theta-S2 to Theta-S3. We attribute this to Constrained_CPU that ignores burst buffer usage and is, therefore, incapable of ease burst buffer contention between jobs. In contrast, BBSched markedly improves node utilization on Theta-S3 (12.69%), Theta-S4 (20.03%) and Cori-S4 (16.28%) compared to the baseline, and reduces the differences in node usage among workloads. This is because when workloads are shifting from node-bound to burst-buffer-bound, plenty of nodes will be left unused, which provides more room for optimization. We also find that the biased optimization methods are effective in improving the usage of their favorite resource, but are at the risk of decreasing the usage of other resources. For example, Weighted_BB and Constrained_BB increase burst buffer usage by 10.54% and 12.09% at the cost of decreasing node usage by 6.11% and 4.84% respectively on Theta-S1. Similarly, Constrained_CPU improves node usage by 0.18%, but reduces burst buffer usage by 1.33% on Cori-S1. In contrast, unbiased optimization methods, e.g., Weighted method, are more likely to improve utilization of both resources. In addition, although Bin_Packing is capable of improving both node and burst buffer usage, the improvement is no more than 3.72%, which is significantly less than the gains of the optimization methods. This is because Bin_Packing selects jobs iteratively based on individual job information, but not attempts to find the best job combination which can optimize system performance. This demonstrates that optimization is necessary for improving system performance in multi-resource scheduling.

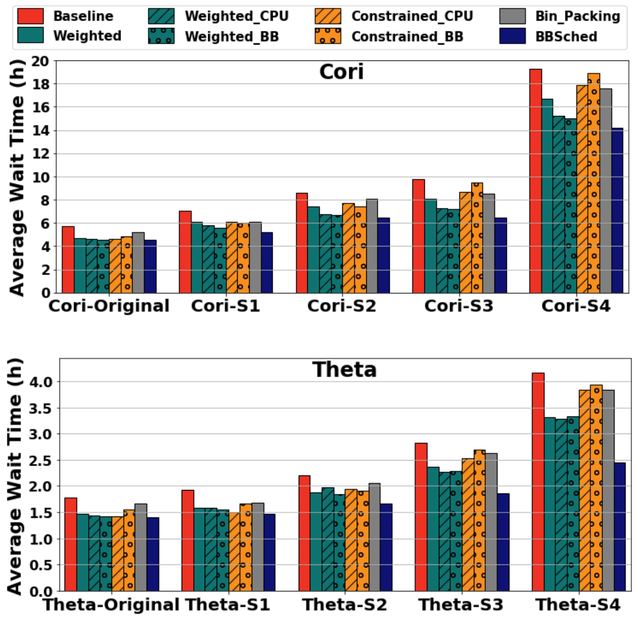


Figure 8: Comparison of average job wait time on Cori traces (top) and Theta traces (bottom). The lower the average job wait time is, the better the performance is.

Impact on Job Wait Time: In Figure 8, we compare average wait time of the eight scheduling methods. It is clear that all methods improve average job wait time in comparison to the baseline. BBSched achieves the most significant reductions on average job wait time, by up to 33.44% on Cori and up to 41% on Theta compared with the baseline. The second best method, Weighted_BB, improves average job wait time by less than 26% on both machines. We also notice that average job wait time increases dramatically as the burst buffer requests increases. For example, when using the baseline method, the average job wait time on Cori-Original is less than 6 hours, compared to 19 hours on Cori-S4. Additionally, the surge of burst buffer requests provides more opportunities for the optimization methods to reduce job wait time. For instance, BBSched reduces the average job wait time by 21.30% on Theta-Original, while it reduces the average job wait time by 41% on Theta-S4.

To understand the origin of the gains, Figure 9 shows the breakdown of average job wait time by job sizes on Theta-S4. We observe that the most significant gain comes from small jobs. BBSched reduces average wait time by 48.29% on 1-8 node jobs, compared to

31.59% of reduction on 1024-4392 node jobs. The baseline method on Theta (WFP) prefers large jobs. However, small jobs on Theta wait less time than large jobs owing to EASY backfilling. The great wait time reductions on small jobs suggest that the optimization methods are more effective than EASY backfilling in avoiding resource fragmentation in multi-resource scheduling. We observe the similar results on all other workloads, and thus we only present the representative results on Theta-S4.

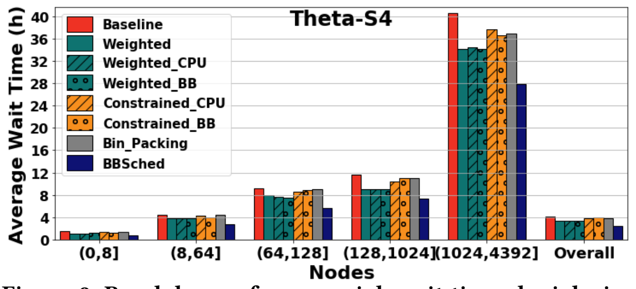


Figure 9: Breakdown of average job wait times by job sizes on Theta-S4.

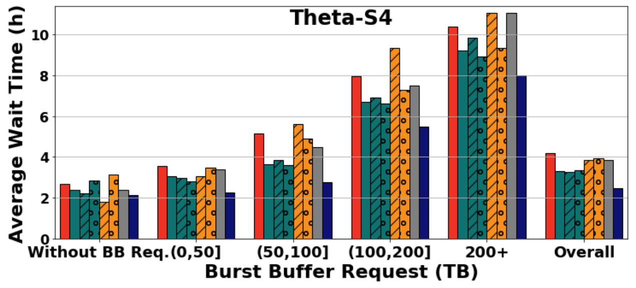


Figure 10: Breakdown of average job wait times by burst buffer requests on Theta-S4.

To show the impact of burst buffer requests, Figure 10 presents the breakdown of average wait time by burst buffer requests on Theta-S4. Clearly, jobs with burst buffer requests wait longer times than jobs without burst buffer requests. For example, when using the baseline method, jobs with more than 200TB burst buffer requests wait, on average, 10.25 hours as opposed to 2.5 hours of jobs without burst buffer requests. We also observe that BBSched and weighted methods make more significant reductions on average jobs wait time of jobs with burst buffer requests. This is because they are designed to optimize both node and burst buffer utilization and therefore jobs with both node and burst buffer requests benefit more from these methods. In contrast, Constrained_CPU fails to improve average wait time of jobs with burst buffer requests, because it only focuses on optimization CPU usage. For example, although Constrained_CPU decreases the average wait time of jobs without burst buffer requests by 32.43%, it increases the average wait time of jobs with 100-200TB burst buffer requests by 17.21%. Under heavy burst buffer requests, such as Theta-S4, Constrained_CPU optimizes node utilization by allocating jobs without burst buffer requests. But once jobs with burst buffer requests go over the threshold in the window, they are forced to run leaving a fraction of the nodes unused. Such inefficient job selections lead to poor performance on both node and burst buffer utilizations and job wait times. Constrained_BB, on the hand, jeopardizes jobs without burst buffer requests to improve wait times of jobs with burst buffer requests.

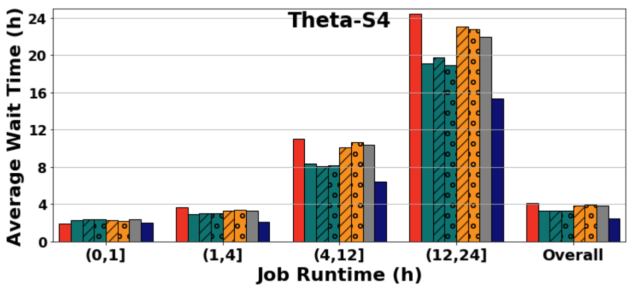


Figure 11: Breakdown of average job wait times by job runtimes on Theta-S4.

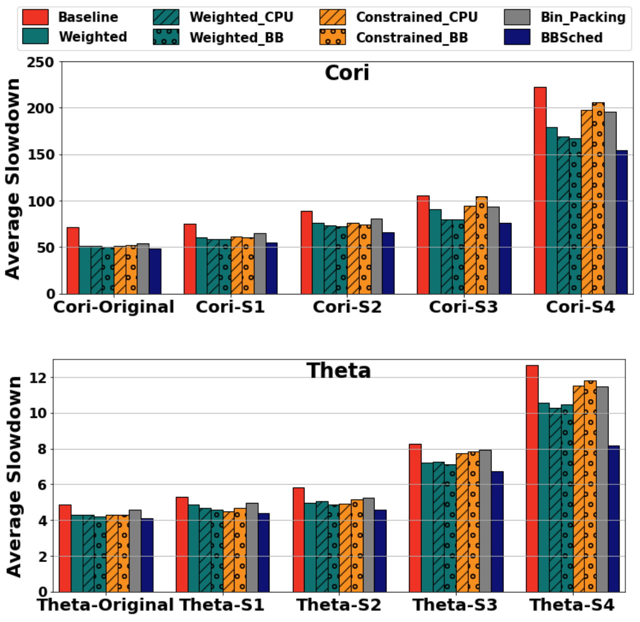


Figure 12: Comparison of average slowdown on Cori traces (top) and Theta traces (bottom). The lower the average slowdown is, the better the performance is.

If we look at the breakdown of average job wait time by job runtimes in Figure 11, we find that jobs wait times increase significantly with job runtimes. The reasons why short jobs wait less than long jobs are the baseline policy (WFP) and EASY backfilling. In WFP, shorter jobs get higher priorities to run. In EASY backfilling, jobs can be backfilled if it does not delay the first job in the queue. Hence, short jobs are more likely to be backfilled than long jobs. It is interesting to notice that all optimization methods reduce average wait time of long jobs, but increase the average wait time of short jobs. This is because the optimization methods only take job's resource requirement into consideration, which does not include job runtimes. As the optimization methods improves resource usage, less idle resources are left for backfilling. As a result, compared with the baseline, the optimization methods lead to longer wait times for short jobs owing to fewer opportunities for backfilling and shorter wait times for long jobs due to improvement in resource usage.

Impact on Slowdown: Figure 12 compares average slowdown of the scheduling methods. We find that the trends on average slowdown are similar to that on average wait time. Additionally, the performance of average slowdown is related to resource usage. The average slowdowns of Theta-S4 and Cori-S4 are evidently higher

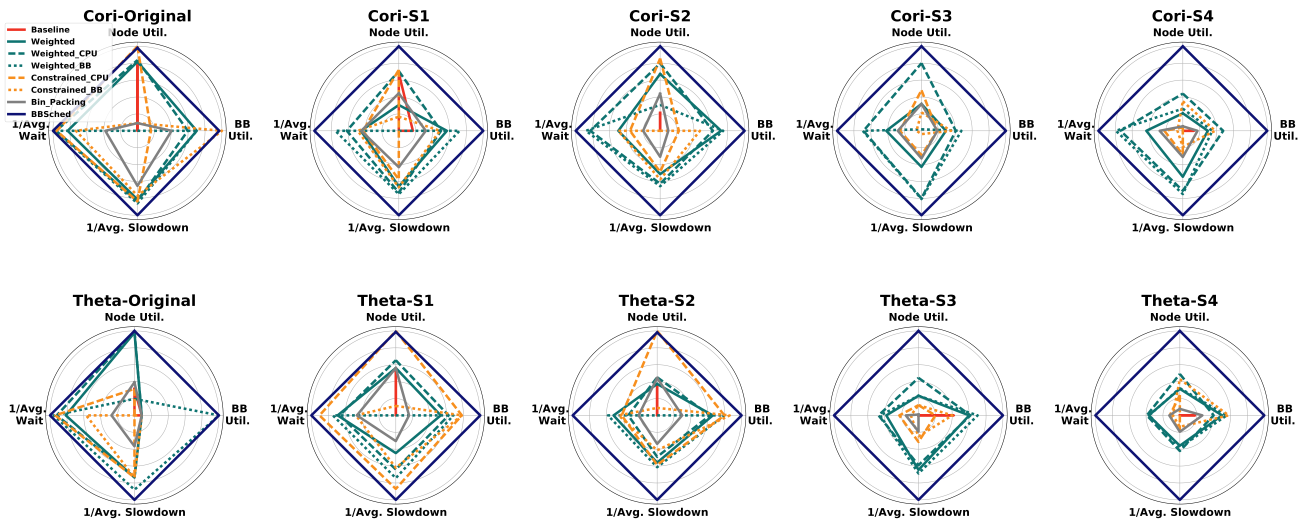


Figure 13: Overall scheduling performance comparison using Kiviat graphs: Cori traces (top) and Theta traces (bottom). The larger the area is, the better the overall performance is.

than other workloads. These workloads are also characterized by low node usage and high burst buffer usage. In these scenarios, even though more jobs are waiting to be scheduled, the severe burst buffer contention prevents jobs with burst buffer requests from being allocated, leaving a fraction of the nodes unused.

Holistic Performance Comparison: To provide a holistic view of the performance of different methods, we present the scheduling results using Kiviat graph (see Figure 13). We use the reciprocal of average job wait time and the reciprocal of average slowdown in the plots. All metrics are normalized to the range of 0 to 1. 1 means a method achieves the best performance among all methods and 0 means a method obtains the worst performance. For all metrics, the larger the area is, the better the overall performance is.

Clearly, BBSched achieves the best and the most balanced performance, as it improves all the metrics significantly. Weighted methods and constrained methods make improvement on some metrics. Their overall performance, however, is unbalanced and is much lower than BBSched. Besides the baseline method, Bin_Packing obtains the poorest performance, as it is an aggressive approach which makes scheduling decision based on isolated job information rather than information of multiple jobs. We also find that as the intensity of burst buffer requests increases, the areas of all methods except BBSched are shrinking. This suggests that BBSched can make notable performance improvement even under heavy burst buffer requests.

Sensitivity Analysis: Several parameters are used in the BBSched design. The selection of the number of generations (G) and population size (P) is discussed in Section 3.2. The window size is another parameter in BBSched. Tuning the window size enables us to balance scheduling performance and preservation of the original job order. A larger window size leads to better resource utilization at the expense of higher computation overhead. Although window size does not directly affect computational complexity of BBSched (described in Section 3.3), a larger window size expands the search space and thus need more generations and larger population sizes

Table 3: BBSched performance under different window sizes. There are two numbers per cell: the top is for Cori-S4 and the bottom is for Theta-S4.

Metrics \ Window Size	10	20	50
CPU Usage	60.18% 67.12%	64.90% 73.29%	65.06% 74.34%
Burst Buffer Usage	92.53% 84.23%	94.74% 89.54%	94.65% 89.63%
Average Job Wait Time (s)	55,732 10,402	51,028 8,847	50,871 8,792
Average Slowdown	162.37 8.93	154.43 8.16	153.20 8.08

to achieve acceptable approximation. Table 3 shows the sensitivity study on Cori-S4 and Theta-S4. As we can see, the most significant improvement is obtained when the window size is between 10 and 20. The improvement slows down with further increase in window size. Note that the Cori and the Theta workloads represent two types of HPC computing, namely capacity computing and capability computing. Considering that a larger window size can cause more disturbance to the original job order as well as higher computation overhead, we believe a window size of around 20 is an appropriate option for typical HPC workloads.

Scheduling Overheads: For methods other than BBSched, scheduling overhead depends on window size (w). As we increase w, the scheduling overhead increases. For BBSched, the number of generation (G) is the main factor that affects scheduling time.

All experiments were performed on Intel Core i5 3.4GHz PC with 4 GB of RAM. It is not surprised that, besides the baseline, Bin_Packing uses the least time (0.1s when w is 50) in making scheduling decision. This is because Bin_Packing is a greedy method rather than an optimization method. Although other methods take more time, they still satisfy the time requirement of HPC scheduling (15-30 seconds). For example, if we set G to 2000 and w to 50 in BBSched, the average scheduling time is less than 2 seconds.

5 INCORPORATING MORE RESOURCES

The design of BBSched is generally applicable to schedule other shared or local resources. Local SSD is a representative local resource in HPC. Both Theta at ALCF and Summit at OLCF are equipped with local SSDs. On Theta, each node is equipped with a 128 GB local SSD and some will be gradually replaced by 256GB SSDs in the near future. In this section, we present a case study to illustrate that BBSched can be easily extended to incorporate additional schedulable resources.

Problem Formulation: Suppose a system has N nodes and B GB burst buffer. Each node is equipped with either 128GB SSD or 256GB SSD. $J = \{J_1, \ldots, J_w\}$ is a set of w jobs in the scheduling window: job J_i requiring n_i nodes, b_i GB of shared burst buffer and s_i GB of local SSD per node. For the j-th node assigned to job J_i, its actual local SSD volume l_{ij} should be equal to or greater than the requested amount s_i. The difference between assigned SSD volume and requested SSD volume is considered as wasted local SSD volume. In addition to the two objectives in Section 3.2, we augment the MOO formulation with two additional objectives:

(3) maximize local SSD utilization: $f_3(\boldsymbol{x}) = \sum_{i=1}^{w} s_i \times n_i \times x_i$

(4) minimize wasted local SSD: $f_4(\boldsymbol{x}) = -\sum_{i=1}^{w} \left(\sum_{j=1}^{n_i} (l_{ij} - s_i) \right) \times x_i$

The multi-resouce scheduling problem can be formulated as:

$$\begin{aligned}
\max \quad & (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x}), f_4(\boldsymbol{x})) \\
\text{s.t.} \quad & \sum_{i=1}^{w} n_i \times x_i \le N - N_{used}, \quad x_i \in \{0,1\} \\
& \sum_{i=1}^{w} b_i \times x_i \le B - B_{used}, \quad x_i \in \{0,1\} \\
& s_i \le l_{ij}, \quad l_{ij} \in \{128, 256\}
\end{aligned}$$

In the fourth objective, we minimize the wasted local SSD with the constraint that the assigned local SSD volume l_{ij} is not less than the requested amount s_i.

MOO Solver: The MOO solver basically remains the same and the only change is the rule of selecting the preferred solution in the decision maker. To accommodate one more resource, we adopt the following rule. First, we choose the solution with the maximum node utilization; next, we replace the preferred solution by another solution if the sum of the improvement in burst buffer utilization, local SSD utilization, and percentage of reduction in wasted local SSD of another solution is more than 4x of the loss of the node utilization. If we find more than one such solution, we choose the solution with the maximum sum of improvement.

Workload Traces: In terms of the hardware configuration, we assume 50% of nodes in the system are equipped with 128 GB local SSDs, the rest of the nodes are equipped with 256 GB local SSDs. We generate three workloads (S5-S7) on top of Cori-S2 and Theta-S2 by creating job's local SSD requests. In S5, 80% of jobs have 0-128GB local SSD requests, and 20% of jobs have 129-256GB local SSD requests. In S6, 50% of jobs have 0-128GB local SSD requests, and 50% of jobs have 129-256GB local SSD requests. In S7, 20% of jobs have 0-128GB local SSD requests, and 80% of jobs have 129-256GB local SSD requests. Jobs with more than 128GB local SSD requests have to be allocated to nodes with 256GB SSD. Jobs with no more than 128GB local SSD requests can be either allocated to nodes with 128GB or 256GB SSD. When assigning nodes to jobs with 0-128GB local SSD requests, nodes with 128GB SSD are preferred over 256GB SSD in order to mitigate wastage in local SSD.

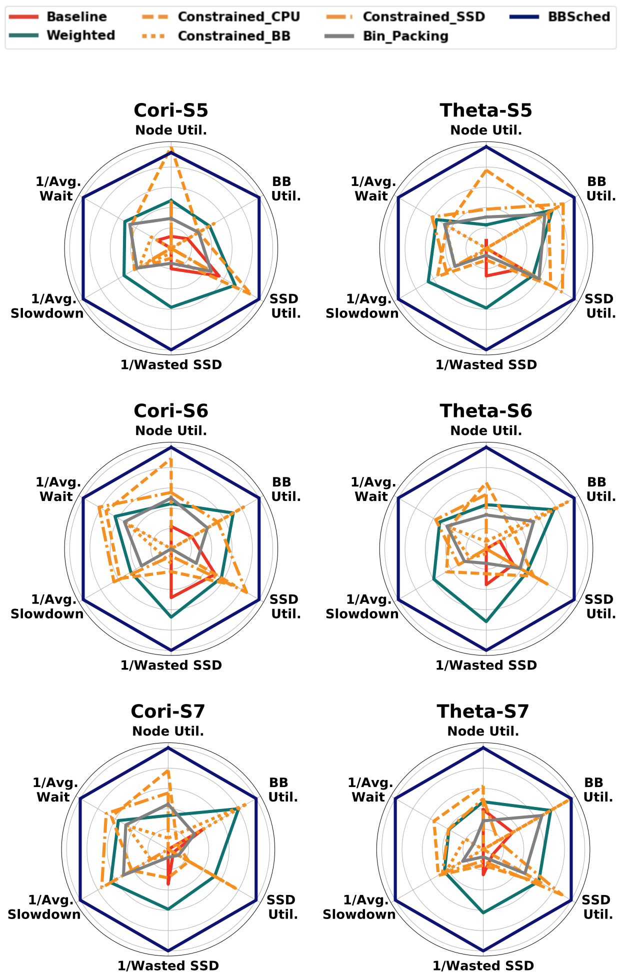


Figure 14: Overall scheduling performance comparison using Kiviat graphs: Cori (left) and Theta (right). Note that as compared to Figure 13, the Kiviat graphs contain two additional metrics, that is, SSD utilization and the reciprocal of wasted SSD.

Scheduling Methods: We compare seven scheduling methods, i.e., Baseline, Weighted, Constrained_CPU, Constrained_BB, Constrained_SSD, Bin_Packing, and BBSched. Weighted method aims to maximize the equally weighted sum of node, burst buffer, local SSD utilization, and negative percentage of wasted SSD. Constrained_SSD method aims to maximize local SSD utilization under the constraints of the other resources. The rest of the methods use the same strategies as described in Section 4.3.

Results: Figure 14 shows a holistic view of scheduling performance. We observe that BBSched achieves the best overall performance on all workloads. After BBSched, Constrained_CPU and Constrained_SSD methods have relatively good performance on both node utilization and local SSD utilization. This indicates that node utilization and local SSD utilization are correlated. Improving the utilization of one type of the resource increases the utilization of another resource. However, high local SSD utilization does not necessarily mean low wasted local SSD resource. The constrained methods and Bin_Packing method waste local SSD resource more than the baseline, because they aggressively allocate nodes with 256GB SSD to jobs with 0-128GB SSD requests. Constrained_BB obtains high burst buffer buffer utilization but low node and SSD utilization. Both Weighted and BBSched methods yield balanced performance. However, the improvement of Weighted method is noticeably lower than BBSched.

6 CONCLUSION

In this paper, we have presented BBSched, a multi-resource scheduling scheme for HPC. In our design, the multi-resource scheduling problem is formulated into a MOO problem and is rapidly solved by a multi-objective genetic algorithm. BBSched generates a Pareto set for decision making, which enables system managers to explore potential tradeoffs among multiple resources for better utilization of multiple schedulable resources.

We have compared BBSched with existing methods using two real workloads and eight synthetic workloads. The extensive trace-based simulations demonstrate BBSched outperforms the existing methods in terms of both system-level and user-level metrics. Specifically, BBSched is capable of improving the overall performance by 41% over naive method, 33% over bin packing method, 35% over constrained methods, and 20% over weighted methods. *This indicates that considering all resources in an explicit optimization is essential for HPC systems with multiple schedulable resources.* Moreover, we have presented a case study to show that BBSched can be extended to incorporate other resources (e.g., local SSDs in this case study). Given the promising results demonstrated in this study, our future work is to deploy and test BBSched on production systems.

ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation grants CNS-1717763, CCF-1422009, CCF-1618776, and the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, DE-AC02-05CH11231.

REFERENCES

[1] Cobalt. https://www.alcf.anl.gov/cobalt-scheduler
[2] Cori. http://www.nersc.gov/users/computational-systems/cori/
[3] Darshan - HPC I/O Characterization Tool. http://www.mcs.anl.gov/research/projects/darshan/.
[4] Moab. http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/
[5] PBS Professional. http://www.pbsworks.com/
[6] Summit. https://www.olcf.ornl.gov/summit/
[7] Theta. https://www.alcf.anl.gov/theta
[8] Trinity. http://www.lanl.gov/projects/trinity/
[9] A. Verma and L. Pedrosa and M. Korupolu and D. Oppenheimer and E. Tune and J. Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *EuroSys*.
[10] W. Allcock, P. Rich, Y. Fan, and Z. Lan. 2017. Experience and Practice of Batch Scheduling on Leadership Supercomputers at Argonne. In *JSSPP*.
[11] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*.
[12] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*.
[13] K. Deb. 2001. Multi-Objective Optimization Using Evolutionary Algorithms. In *John Wiley and Sons, Inc.*
[14] A.E. Eiben and S.K. Smit. 2011. Parameter Tuning for Configuring and Analyzing Evolutionary Algorithms. In *Swarm and Evolutionary Computation.* 19–31.
[15] Y. Fan, P. Rich, W. Allcock, M. Papka, and Z. Lan. 2017. Trade-Off Between Prediction Accuracy and Underestimation Rate in Job Runtime Estimates. In *CLUSTER.* 530–540.
[16] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*.
[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM*.
[18] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*.
[19] R. L. Haupt. 2000. Optimum Population Size and Mutation Rate for a Simple Real Genetic Algorithm that Optimizes Array Factors. In *APS*.
[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*.
[21] C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang. 2018. Wide-Area Analytics with Multiple Resources. In *EuroSys*.
[22] W. Jakob, S. Strack, A. Quinte, G. Benge, K Stucky, and W. Süß. 2013. Fast Rescheduling of Multiple Workflows to Constrained Heterogeneous Resources Using Multi-Criteria Memetic Computing. In *Algorithms*.
[23] M. Jette and T. Wickberg. 2015. Slurm Burst Buffer Support. In *Slurm User Group Meeting*.
[24] M. A. Jette, A. B. Yoo, and M. Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *JSSPP*. 44–60.
[25] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. 2012. Multi-Resource Allocation: Fairness-Efficiency Tradeoffs in a Unifying Framework. In *INFOCOM*.
[26] K. Bringmann and T. Friedrich. 2012. Approximating the Least Hypervolume Contributor: NP-hard in General, but Fast in Practice. In *Theoretical Computer Science.* 104–116.
[27] A. Konak, D. Coit, and A. Smith. 2006. Multi-Objective Optimization using Genetic Algorithms: A Tutorial. In *Reliability Engineering and System Safety*.
[28] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *MSST*. 1–11.
[29] A. W. Mu'alem and D. G. Feitelson. 2001. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *TPDS*.
[30] M. NoroozOliaee, B. Hamdaoui, M. Guizani, and M. Ghorbel. 2014. Online Multi-Resource Scheduling for Minimum Task Completion Time in Cloud Servers. In *INFOCOM WKSHPS*.
[31] L. Rao, X. Liu, L. Xie, and W. Liu. 2010. Minimizing Electricity Cost: Optimization of Distributed Internet Data Centers in a Multi-Electricity-Market Environment. In *INFOCOM*. 1–9.
[32] M. J. Reddy and D. N. Kumar. 2007. An Efficient Multi-Objective Optimization Algorithm based on Swarm Intelligence for Engineering Design. In *Engineering Optimization.* 49–68.
[33] S. Ren, Y. He, and F. Xu. 2012. Provably-Efficient Job Scheduling for Energy and Fairness in Geographically Distributed Data Centers. In *ICDCS*.
[34] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *EuroSys*.
[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*.
[36] S. Wallace, X. Yang, V. Vishwanath, W. Allcock, S. Coghlan, M. Papka, and Z. Lan. 2016. A Data Driven Scheduling Approach for Power Management on HPC Systems. In *SC*.
[37] W. Wang, B. Liang, and B. Li. 2015. Multi-Resource Fair Allocation in Heterogeneous Cloud Computing Systems. In *TPDS*.
[38] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka. 2013. Integrating Dynamic Pricing of Electricity into Energy Aware Scheduling for HPC Systems. In *SC*.
[39] L. Yu, Z. Zhou, Y. Fan, M. E. Papka, and Z. Lan. 2018. System-wide Trade-off Modeling of Performance, Power, and Resilience on Petascale Systems. In *The Journal of Supercomputing*.
[40] Y. Zhang, G. Prekas, G. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *OSDI*.