Parallel Computing 39 (2013) 753–768

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# Toward balanced and sustainable job scheduling for production supercomputers

Wei Tang [a,*], Dongxu Ren [b], Zhiling Lan [b], Narayan Desai [a]

[a] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
[b] Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA

ARTICLE INFO

*Article history:*
Available online 4 September 2013

*Keywords:*
Job scheduling
Resource management
Priority balancing
Adaptive policy tuning
Workload characteristic

ABSTRACT

Job scheduling on production supercomputers is complicated by diverse demands of system administrators and amorphous characteristics of workloads. Specifically, various scheduling goals such as queuing efficiency and system utilization are usually conflicting and thus need to be balanced. Also, changing workload characteristics often impact the effectiveness of the deployed scheduling policies. Thus it is challenging to design a versatile scheduling policy that is effective in all circumstances. In this paper, we propose a novel job scheduling strategy to balance diverse scheduling goals and mitigate the impact of workload characteristics. First, we introduce metric-aware scheduling, which enables the scheduler to balance competing scheduling goals represented by different metrics such as job waiting time, fairness, and system utilization. Second, we design a scheme to dynamically adjust scheduling policies based on feedback information of monitored metrics at runtime. We evaluate our design using real workloads from supercomputer centers. The results demonstrate that our scheduling mechanism can significantly improve system performance in a balanced, sustainable fashion.



## 1. Introduction

Job scheduling is a critical task on large-scale computing platforms. The job scheduling policy directly influences the satisfaction of both users and system owners. Moreover, the success of a job scheduling policy is largely determined by the satisfaction of these stakeholders. Users are concerned with fast job turnaround and fairness, while system owners are interested in system utilization. Also, production supercomputing centers are starting to face new challenges in scheduling, such as avoiding failure interrupts and achieving energy efficiency. All these considerations, which are quantified by system metrics, are related but often conflict with one another. Even worse, the priorities differ from machine to machine and from time to time, further complicating the design of a comprehensive job scheduling policy.

Traditional scheduling policies can achieve specific scheduling goals but not balance them well. For example, using "first-come, first served" (FCFS) achieves good job fairness but results in poor response times and resource fragmentation. On the other hand, using "short-job first" (SJF) achieves best response time in theory but violates job fairness and causes job starvation. Essentially, these approaches attempt to favor a fixed combination of some priorities while ignoring others. Some traditional schedulers provide mechanisms to switch between these policies when particular boundary conditions are encountered; however, this approach provides only a coarse ability to refine goal-based priorities.

* Corresponding author. Tel.: +1 3129121207.
*E-mail addresses:* wtang@mcs.anl.gov (W. Tang), dren1@iit.edu (D. Ren), lan@iit.edu (Z. Lan), desai@mcs.anl.gov (N. Desai).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.08.007

Moreover, user satisfaction and system performance are not considered in a holistic fashion. Typically, job prioritizing and resource allocation are separated into two subsequent phases in decision making. This division greatly constrains the resource allocation process. For example, when a high-priority job suffers from insufficient resources, it reserves the resources while draining others; yet, these resources could be used to execute other low-priority jobs, thereby improving system performance. This kind of resource draining causes external fragmentation. While backfilling helps in this case, it only mitigates fragmentation already created by this division [21].

Another issue of job scheduling concerns dynamic workload. Even though we may identify a policy to achieve our integrated goals well for one workload, the policy may fail for a different workload. Although event-driven simulation can be used to evaluate the aggregate effect of a scheduling policy on a historical workload trace, it cannot provide much guidance when workload properties change dynamically at runtime.

Motivated by these issues, we propose an adaptive metric-aware job scheduling mechanism. Our objectives are twofold. First, we develop a metric-aware job scheduling mechanism to prioritize jobs and allocate resources in an integrated fashion. Here, "metric-aware" means we take the targeted performance metrics into account when configuring a specific scheduling policy. Moreover, the prioritized jobs are allowed to be altered in a limited fashion during the resource allocation phase. This approach improves flexibility in schedule creation.

Second, we introduce an adaptive tuning mechanism into job scheduling, which allows the scheduling policy to change dynamically at runtime based on workload characteristics. By monitoring the performance metrics at runtime, the scheduler can adjust its scheduling policy to favor the metrics that are less satisfied recently, thereby mitigating the impact of changing workload characteristics on the scheduling policy. For example, if the system utilization rate is below a certain threshold (e.g., a longer-term average), our scheduling system will adjust its policy to favor system utilization more than other metrics.

We implemented our design in the production resource management system called Cobalt [1]. We evaluated our design using recent real job traces from multiple production supercomputers. The experimental results show that our approach can achieve significant and sustainable performance improvement compared with traditional scheduling strategies.

The remainder of this paper is organized as follows. Section 2 discusses some related work. Section 3 reviews the motivation of our work and describes our methodology. Section 4 illustrates our experimental results. Section 5 summarizes the paper and briefly discusses future work.

## 2. Related work

The balancing of multiple scheduling objectives is supported by some production job schedulers. One example is the Maui scheduler [10], which uses a number of weighted and combined parameters to prioritize jobs. Moab [2], its commercial descendant, is extremely flexible and supports more than 250 scheduling parameters. To alleviate the tedious work of manual configuration for a Moab scheduling policy, Krishnamurthy et al. [11] provided a toolset that can help a system administrator to automatically configure a scheduling policy. Basically, the toolset uses a genetic algorithm-based scheme working with simulations from historical workloads to find an effective configuration. The open-source Cobalt resource manager [1] uses a simple utility function to prioritize jobs, which can also take into account multiple scheduling considerations. Cobalt provides an event-driven simulator to guide the design of an appropriate utility function [22]. While our approach also provides the ability to balance different scheduling goals, it does not rely on the simulation results of recent workloads. That is, the parameters of a scheduling policy can be tuned at runtime based on the feedback of monitored performance metrics, thereby adapting to different workload characteristics.

A dynamic tuning scheduling policy can be found in some existing work. Grothlags and Streit [19] and Streit [20]8 proposed a self-tuning "dynP" job scheduler that can tune queuing policy dynamically during runtime. Our work shares similar motivation but differs from those works in two ways. First, whereas the "dynP" scheduler switches policy between FCFS, SJF, and LJF (largest job first) based on the number of jobs in the queue, our scheduler supports fine-grained tuning based on more sophisticated monitoring of a number of targeted system metrics. Second, in our work, both the queuing policy and the job allocation policy can be tuned, either independently or in a two-dimensional fashion.

Feedback-based scheduling has been used in operating systems or real-time systems. Blevins and Ramamoorthy [5] proposed using feedback information to adjust the schedule in general-purpose operating systems in the form of multilevel feedback queue scheduling. Lu et al. [12] designed and evaluated a feedback control earliest-deadline-first scheduling algorithm for scheduling in real-time systems. In parallel job scheduling, however, schedulers generally use "open loop" scheduling algorithms in which policies are not adjusted based on continuous feedback. Yet, in production supercomputers the dynamics of the workload is amorphous and can influence the effectiveness of a scheduling policy. Thus, utilizing feedback information of workload change is beneficial in the selection of a scheduling policy. It is one of the motivations of our work. Our work differs from existing feedback-based scheduling efforts, however, in that we do not adjust the schedules directly but instead adjust the scheduling policy that will indirectly change the schedules.

Etsion and Tssafrir [7] reported that the prevalent default scheduler setting is FCFS and that in those management suites that also support backfilling, the governing scheme used is EASY [13]. Indeed, considerable work has been done to enhance either FCFS or EASY backfilling. Ababneh and Bani-Mohammad [4] proposed an enhancement to FCFS that uses a window of consecutive jobs from which jobs are selected for allocation and execution. Shmueli et al. [17] optimized the packing of backfilling jobs by looking ahead into the queue. Srinivasan and Feitelson [18] designed a selective reservation strategy

for backfilling jobs intended to obtain the best characteristics from both EASY backfilling and conservative backfilling. Ward et al. [26] proposed a relaxed backfilling scheme that allows backfilled jobs to moderately delay reserved jobs. Our metric-aware scheduling is also an enhancement of FCFS and backfilling. Using a balance factor we can tune FCFS toward SJF to a desired extent. The window-based allocation also provides more flexibility in job allocation and backfilling. Moreover, our policy can be dynamically tuned to adapt to changes of workload.

Fairness is an important metric in measuring a job queuing system. While it tends to be ignored by many scheduling studies, a number of existing works do focus on it. Rafaeli et al. [14] demonstrated with psychological experiments that for humans waiting in queues, the issue of fairness can be more important than the duration of the wait. Raz et al. [15] proposed a resource allocation queuing fairness measure that accounts for both relative job seniority and relative service time. Sabin et al. [16] proposed an approach to assess fairness in non-preemptive job scheduling; they defined fairness such that no later-arriving job should delay an earlier-arriving job. Yuan et al. [27] proposed a stricter fairness model in which no job is delayed by any job of lower priority and the delay caused by running time overestimation of backfilled jobs is also counted. Our work considers fairness, but our major focus is on balancing a variety of selected metrics including fairness.

## 3. Methodology

In this section we discuss the metrics we propose for our job scheduling policy. We then describe the design of the metric-aware job scheduler.

### 3.1. Goals

The effectiveness of a job scheduling policy is measured by certain metrics. A metric quantifies the job scheduler performance in terms of a single aspect. Computing centers monitor and optimize many such metrics, including utilization, response time, and fairness. Various metrics have been proposed by scheduling researchers. Generally, the metrics can be grouped into two categories: user-centric metrics, reflecting levels of user satisfaction, and system-centric metrics, measuring the system utilization and costs.

While these metrics may seem independent, they are related and often in conflict with one another. A mechanism is needed to explicitly balance these considerations. But this balance is not trivial; different systems (or the same system during different times) have varied priorities that result in some considerations prevailing over others. Therefore we propose an adaptive metric-aware job scheduling mechanism. The goal of this mechanism is twofold. First, it must connect job priorities explicitly to the impact (in metric space). Second, it must be able to tune scheduling policies based on feedback from the monitored metrics.

### 3.2. Design

Fig. 1 shows a diagram of our design, which comprises three major components: a metrics balancer, a scheduling algorithm, and a metrics monitor. The metrics balancer is used to balance different priorities or scheduling goals in composing an integrated scheduling algorithm. By running the scheduling algorithm, relevant metric values are monitored. Feedback from

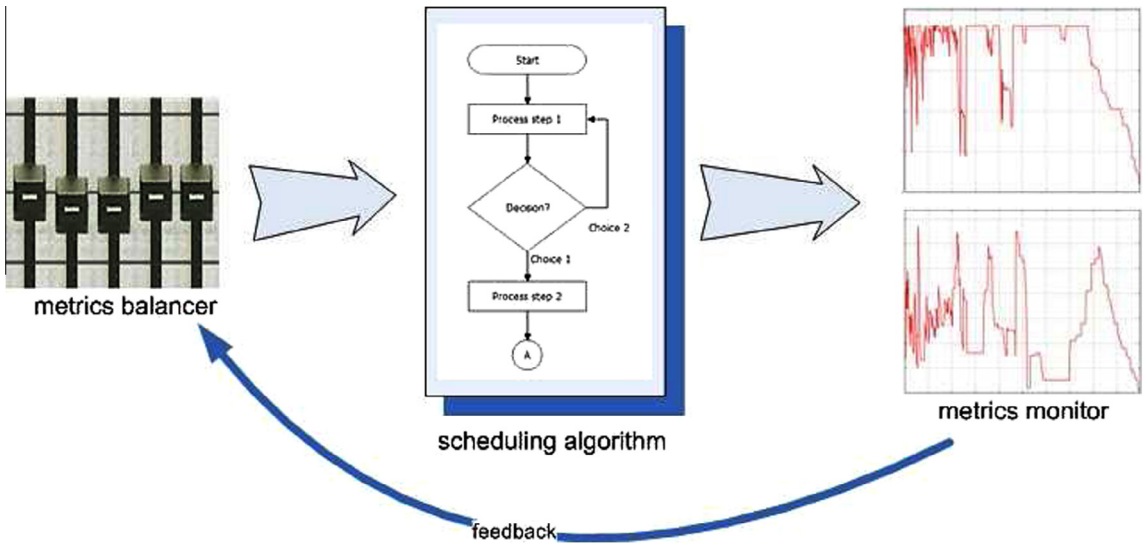



**Fig. 1.** Diagram of adaptive metric-aware job scheduling framework.

the metrics monitor is sent back to the metrics balancer so that it can adjust the weight of different factors in order to maintain the desired balance. This feedback process can be conducted manually or automatically. We call the process of how the metrics balancer impacts the scheduling algorithm "metric-aware job scheduling" and the process of how the metrics monitor influences the adjustment to the metrics balancer "adaptive policy tuning." In the following two subsections we present the detailed design of the metric-aware scheduling and adaptive policy tuning, respectively.

### 3.3. Metric-aware job scheduling

We present here the detailed design of metric-aware job scheduling. The goal is to provide a metric balancer that can determine a scheduling algorithm that balances the metrics of interest. In our current design, three metrics are to be balanced. One reflects system utilization, and other two reflect user satisfaction. Of the latter two, one is "fairness" and the other "efficiency" (i.e., fast turnaround).

User-centric metrics are influenced mainly by job queuing policies, whereas system metrics are correlated with resource allocation. Existing scheduling schemes typically allocate jobs one by one in priority order. The problem is that although this approach can achieve the best allocation for that single job, it neglects the potential impact on other jobs in the queue (one simple example is shown in Fig. 2). Thus we propose to schedule and allocate a group of jobs at one time so that the job allocation can result in more optimal system utilization. We call the number of jobs that are allocated at one time the "window size" (W), which varies from one to up to the number of queued jobs. Intuitively, the larger the window size is, the better the job allocation that can be achieved; but this approach may violate user-centric metrics especially for fairness.

To balance fairness and efficiency, we sort the queue based on a priority score considering both waiting time and running time. By tuning the "balance factor" (BF), which represents the weight of these two parameters, we can balance the priority between fairness and efficiency. The detailed scheduling steps are as follows.

Step 1: For each job *i*, calculate its score regarding waiting time, mapping the values to [0, 100]:

$$S_w = 100 \times \frac{wait_i}{wait_{max}}, \tag{1}$$

where $wait_{max}$ is the maximum waiting time of the current queue and $wait_i$ is the current waiting time of job *i*. If the maximum is 0, $S_w$ is set to 0. This case happens when a job is newly submitted to an empty queue.

Step 2: For each job *i*, calculate its score regarding requested wall time, mapping the values to [0, 100]:

$$S_r = 100 \times \frac{walltime_{max} - walltime_i}{walltime_{max} - walltime_{min}}, \tag{2}$$

where $walltime_{max}$ and $walltime_{min}$ are the maximum and minimum wall times of the current queue, respectively, and $walltime_i$ is the wall time of job *i*. If only one job is in the queue or if $walltime_{max}$ equals $walltime_{min}$, $S_r$ is set to 0.

Step 3: For each job *i*, calculate its balanced priority:

$$S_p = BF \times S_w + (1 - BF) \times S_r, \tag{3}$$

where BF is the balance factor varying from 0 to 1. BF closer to 1 means favoring fairness more; BF closer to 0 means favoring efficiency more. This value is preset but can be adjusted dynamically.

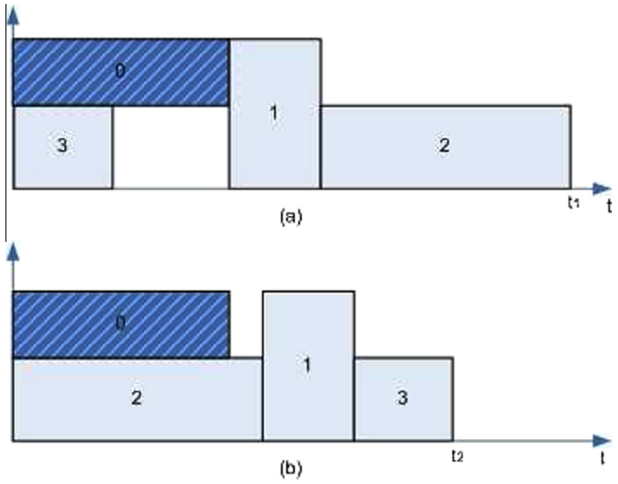



**Fig. 2.** Example showing the limitation of scheduling and allocating jobs one by one. Job 0 is running; but Jobs 1, 2, and 3 are waiting: (a) schedule and allocate job one by one in priority order; (b) schedule and allocate in a group as a whole. Apparently (b) achieves better system utilization.

Step 4: Sort all the jobs by their balanced priorities $S_p$.

Step 5: Group jobs with window size $W$. For each job window, do job allocation. The job allocation algorithm with window size $W$ runs as follows. Based on the permutation of the jobs, do greedy job allocation. If the job has enough idle nodes to run, start it; otherwise, find an earliest time that it can obtain enough nodes to reserve this job. Select one schedule with the least makespan, meaning that the jobs in the window generate a schedule with the highest utilization rate.

Step 6: Add another pass through the queue to try to backfill remained jobs, conforming with the original configuration of backfilling schemes. For EASY backfilling, the reservation of jobs in the first window will not be delayed by backfilling jobs. For conservative backfilling, no reservations can be delayed [13].

In sum, a scheduling policy is determined by the balance factor BF and window size W. If the BF is closer to 1, the queue policy is closer to FCFS; otherwise, the policy is more like SJF. A larger window size means that more jobs in a window can be reordered, thereby enjoying more flexibility in resource allocation. If BF and W are both set to the default value 1, the scheduling policy is the most commonly used scheduling policy, namely, FCFS plus backfilling [7].

### 3.4. Adaptive policy tuning

Metric-aware scheduling allows system owners to tune scheduling policies by favoring particular metrics in a fine-grained fashion. However, the effectiveness of a scheduling policy also depends on the workload characteristics, which vary from time to time. To mitigate the impact of workload change, we introduce a adaptive policy tuning mechanism as a supplement to metric-aware scheduling. Basically, we enable the scheduler to dynamically change the scheduling policy (by tuning the balance factor and window size) based on the feedback from the monitored metrics. For example, if the monitored waiting time goal is not satisfied, the balance factor will be adjusted to a value that prioritizes the waiting time goal. This adjustment may impact the fairness metric. But when the fairness goal is insufficiently satisfied, the balance factor will be adjusted again to prioritize other goals.

In order to configure a specific adaptive policy tuning scheme, the parameters in the tuple $< T, T_i, \Delta, M, Th, E_p, E_m, C_i >$ need to be determined. $T$ is the tunable arguments to the scheduling policy. In this study, $T$ is either BF or W. $T_i$ is the initial value of $T$ that determines the initial scheduling policy. $\Delta$ defines the incremental value to tune $T$ at each time. $M$ refers to the metrics monitored at runtime, such as current queue status or system utilization. $Th$ is the threshold of the monitored metrics at which the policy tuning should be triggered. $E_p$ and $E_m$ define the events that trigger the increase and decrease of the tunable arguments, respectively. $C_i$ defines the interval between the checking points at which the metrics are checked and policies are tuned. Table 1 summarizes these parameters.

Proper parameters are relevant to the priorities of individual machines and workload characteristics. Historical statistics of interested metrics may help in choosing the proper values for each parameter. The simulation results based on recent workloads are also instructive. In the next section, we provide example configurations of the policy tuning scheme and evaluate them with experiments.

The adaptive policy-tuning mechanism is described in Algorithm 1. The process begins with initializing tunables. At each checking point the scheduler will first get the values of all monitored metrics and then compare the checked value with the preset thresholds to find out whether a policy tuning event is triggered. If yes, then the tunables are adjusted based on the event type. This metric checking logic is added before the existing job scheduling function, which does the real scheduling work.

**Algorithm 1:** Adaptive scheduling

```
T = T_i      // initialize the tunable
while True do
  if now − last_checked > C_i then      // do check
    m = get_monitored_values();      // get M values
    e = check_event(m);      // check feedback with
    if e == E_p then
      T = T + Δ;      // increase tunable by Δ
    end
    if e == E_m then
      T = T − Δ;      // decrease tunable by Δ
    end
    last_checked = now;      // reset check clock
  end
  schedule_jobs(T);      // do real scheduling stuff
  sleep(SchedInterval)
end
```

**Table 1**
Parameters to configure an adaptive scheme.

| Para | Description | Possible values |
|---|---|---|
| $T$ | tunable | BF or W |
| $T_i$ | initial value of tunable | 1 for both BF and W |
| $\Delta$ | incremental value to tune $T$ | 0.5 for BF or 1 for W |
| $M$ | monitored metrics | queue status or sys. util. |
| $Th$ | threshold of $M$ | (historical statistics) |
| $E_p$ | event triggering $T$ plus $\Delta$ | $M$ reaches $Th$ |
| $E_m$ | event triggering $T$ minus $\Delta$ | $M$ reaches $Th$ reversely |
| $C_i$ | checking interval | 30 min |

## 4. Experiments

In this section, we present a series of experiments evaluating our scheduling mechanism. We begin with a description of the experiment setup and evaluation metrics, followed by experimental results.

### 4.1. Experiment setup

We conducted simulation-based experiments on two sets of separate workloads from two kinds of machines. One workload is from the Blue Gene/P (BG/P) system (named Intrepid [6]) at the Argonne Leadership Computing Facility (ALCF); the other is from the SP2 system at the San Diego Supercomputing Center (SDSC). The former is a large-scale massively parallel processing system with 40,960 computing nodes (163,840 cores); the latter is a cluster system with 128 nodes. The ALCF BG/P workload contained 9300 jobs during the June 2010. The SDSC SP2 workload contained the first 5000 jobs (during two months) in the corresponding job log in the Parallel Workload Archive [3].

We implemented our scheduling algorithms in Cobalt's event-driven simulator [22] with different job allocation and backfilling schemes for each of the two workloads. Specifically, for the BG/P workload, we used contiguous job allocation with which only logically contiguous computing nodes can together serve a single job [9], because BG/P systems use a partitioned 3D torus network to connect computing nodes [23]. For the SP2 workload, we used noncontiguous job allocation because this type of system allows any set of jobs to be grouped together to serve a single job. We used conservative backfilling for the BG/P workload, which was actually running the Cobalt resource manager on Intrepid; and we used EASY backfilling for the SP2 workload since it is commonly used on many similar cluster systems [7]. We ran the same set of experiments on each workload in order to verify the general applicability of our approach.

### 4.2. Evaluation metrics

The following metrics were used in the performance evaluation.

- *Waiting time (wait).* A job's waiting time refers to the time period between when the job is submitted and when it is started. The average waiting time among all finished jobs in a workload is usually measured to reflect the "efficiency" of a scheduling policy. In this paper, the average waiting time is measured in minutes.
- *Average bounded slowdown.* The slowdown of a job is the ratio of the job's response time to its actual running time. Usually, a small running time bound (10 s) is imposed in order to avoid having the value skewed by extremely small jobs. The lengths of jobs in our job trace were all more than 10 s, so what we refer to as slowdown later in this paper is the same as bounded slowdown.
- *Queue depth (QD).* The queue depth is related to both job waiting and queuing efficiency. Specifically, the queue depth at a given moment is measured by the sum of the waiting times *all* the jobs in the current queue have endured up to that moment. This metric is good for real-time monitoring because it reflects the status of the queue. A high queue depth may result because a large number of jobs are waiting or some jobs are experiencing a very long wait time, or both.
- *Fairness.* To assess fairness, we assign a "fair start time" to each job at its submission. Any job started after its fair start time is considered to have been treated unfairly. The fair start time is calculated as follows. Assuming there is no later arrival jobs, we simulate scheduling under the current scheduling policy and determine when the job will be started. We count the number of unfair jobs in order to assess the overall fairness. This metric has been used, for example, by Sabin et al. [16].
- *System utilization rate.* The system utilization rate represents the ratio of the utilized (or delivered) node-hours compared with the total available node-hours in the checked period of time. Usually this metric refers to an average value over a specified period of time. Sometimes when we refer to the instant system utilization rate we count the ratio of the number of busy nodes to the total number of nodes.
- *Loss of capacity (LoC).* The LoC metric is relevant to system utilization and fragmentation. A system incurs LoC when (i) it has jobs waiting in the queue to execute and (ii) it has sufficient idle nodes, but it still cannot execute those waiting jobs because of fragmentation. Let us assume that the system has $N$ nodes and $m$ scheduling events, a situation that occurs

when a new job arrives or a running job terminates, indicated by monotonically nondecreasing times $t_i$, for $i = 1 \ldots m$. Let $n_i$ be the number of nodes left idle between the scheduling event $i$ and $i+1$. Let $\delta_i$ be 1 if there are any jobs waiting in the queue after scheduling event $i$ and at least one is smaller than the number of idle nodes $n_i$, and 0 otherwise. Then loss of capacity is defined as follows:

$$LoC = \frac{\sum_{i=1}^{m-1} n_i (t_{i+1} - t_i)\delta_i}{N \times (t_m - t_1)}. \quad (4)$$

Loss of capacity reflects the costs of resource fragmentation. This metric has been used in our previous work [23]24; a similar metric was used by Zhang et al. [28].

### 4.3. Results of metrics balancing

To show the effect of metric-aware scheduling, we run simulations with various values for balance factor and window size. Specifically, the balance factor was tuned between series [1, 0.75, 0.5, 0.25, 0]. A larger number represents a queuing policy closer to FCFS. Thus, a BF value of 1 emulates FCFS, and a BF value of 0 emulates SJF. The size of the job allocation window varied from 1 to 5. Window size 1 emulates the basic, one-by-one job allocation scheme.

Fig. 3 shows the results of the BG/P workload. As shown in Fig. 3(a), the average waiting time declines as the balance factor decreases. In particular, the waiting time drops significantly as BF is tuned from 1 to 0.5 and then stays at the same level when BF decreases further. These results suggest that prioritizing jobs considering both job age (waiting time) and job length (expected running time) can enhance scheduling efficiency dramatically, compared with pure FCFS (considering job age only). Theoretically, a perfect short-job-first scheme should lead to the best average waiting time. But the results show that as BF is tuned from 0.5 to 0, there is no clear improvement, thus suggesting that putting too much weight on job length brings some bad effects, such as large job starvation. This result limits further improvement in the average waiting time.

Fig. 3(b) shows the results for average slowdown, which is generally consistent with the average waiting time: the slowdown is decreasing as BF is tuned toward 0. The window size does not show a clear impact on the average waiting time and slowdown. But for FCFS, using a window size larger than 1 can considerably improve the average waiting time compared with that of the traditional FCFS plus backfilling scheduling policy.

Fig. 3(c) shows the fairness results for the BG/P workload. Clearly, the number of unfair jobs increases as the policy approaches SJF. A general trend appears showing that a larger window size also decreases fairness. This is expected because a window larger than 1 allows reordering of sorted jobs in the same window.

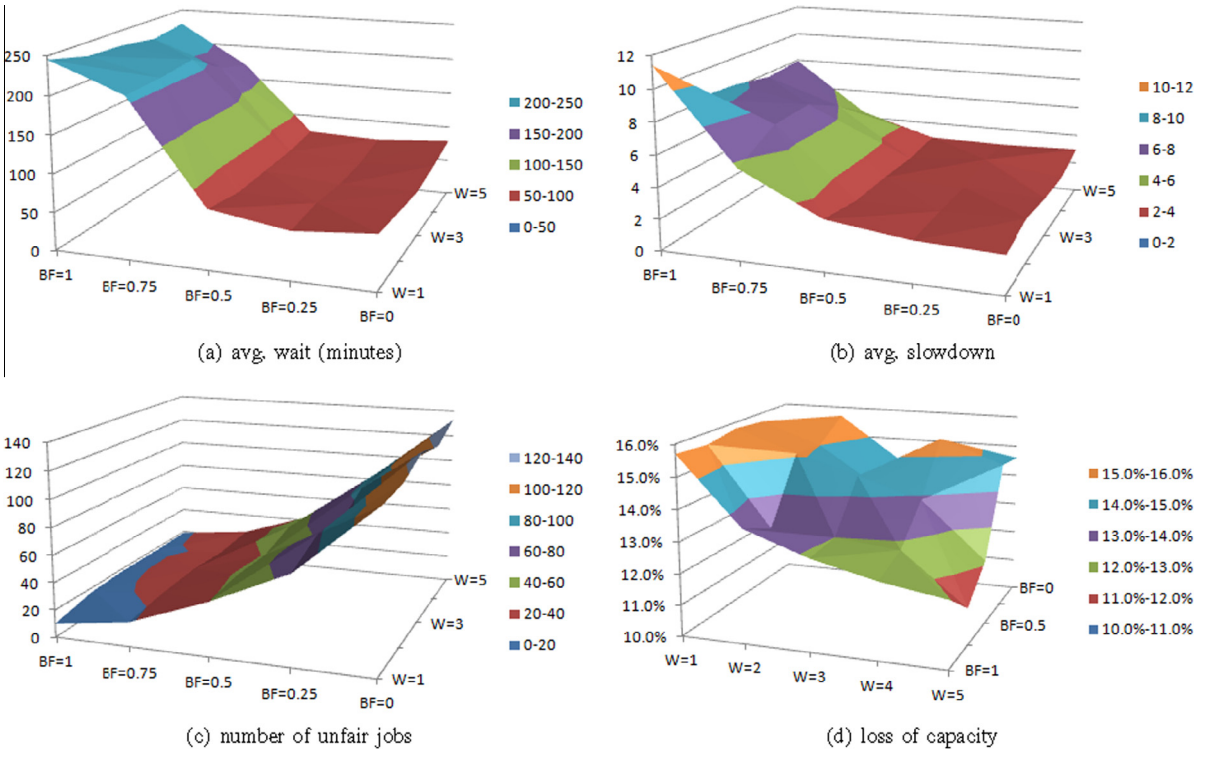



**Fig. 3.** Results of metrics balancing (BG/P).

Fig. 3(d) shows the results in terms of loss of capacity. Since loss of capacity is influenced by window size more than by the balance factor, we put the window size on the x-axis and the balance factor in the legend in order to show the trends. We can see that when BF is no less than 0.5, the loss of capacity shows a decreasing trend as the window size increases, meaning that enlarging the job allocation window can help utilize computing nodes more efficiently. This effect is not shown when the queuing policy gets closer to SJF, as shown, for example, in the lines where BF = 0.25 or BF = 0. From these sets of data, we learn that we can improve "efficiency" (either for queuing or utilizing resources) by moderately sacrificing "fairness", or vice versa.

Fig. 4 shows similar results for the SP2 workload. Generally, the trends are consistent with those for the BG/P workload. The average wait times and slowdown decrease when the queuing policy gets closer to SJF (Fig. 4(a) and (b)). Similarly, the window size provides a slight help in decreasing the average waiting. As shown in the figure, window size 4 achieves the best average waiting time, whereas window size 1 has the worst. Fig. 4(c) indicates that fairness declines when BF decreases. With BF less than 0.5, this effect is striking. Fig. 4(d) shows that the loss of capacity decreases as the window size gets larger.

### 4.4. Results of adaptive tuning

We next examine the effects of adaptive policy tuning. We begin with trying to balance queuing efficiency and fairness by tuning only the balance factor. In this set of experiments we fix the window size to the default number 1.

#### 4.4.1. Monitor of queue depth

Fig. 5 shows the variation of the queue depth, measured by instant aggregate waiting time among all queuing jobs, over time. Fig. 5a shows the data in normal scale, and Fig. 5b presents the data on a logarithm scale, helping to zoom in on the data characteristics.

Four lines are plotted in each figure. Three of them represent static metric-aware scheduling with BF = 1, 0.75, and 0.5, respectively; the fourth line represents BF with adaptive tuning. We do not examine the cases with BF smaller than 0.5 here because previous experiments show poor performance with such settings. Each plot represents the queue depth, that is, the sum of the waiting times of all waiting jobs at that instant of time. The value is checked every 30 min. Thus, the plots are made every 0.5 h on the *x*-axis, which is labeled with the elapsed hours from time zero (the first job submitted). Note that although we have run the experiments with all available workloads, we plot only the first 200 h here for visual clarity because we intend to study the instant influences from tuning the scheduling policies. The overall performance improvement is presented later.

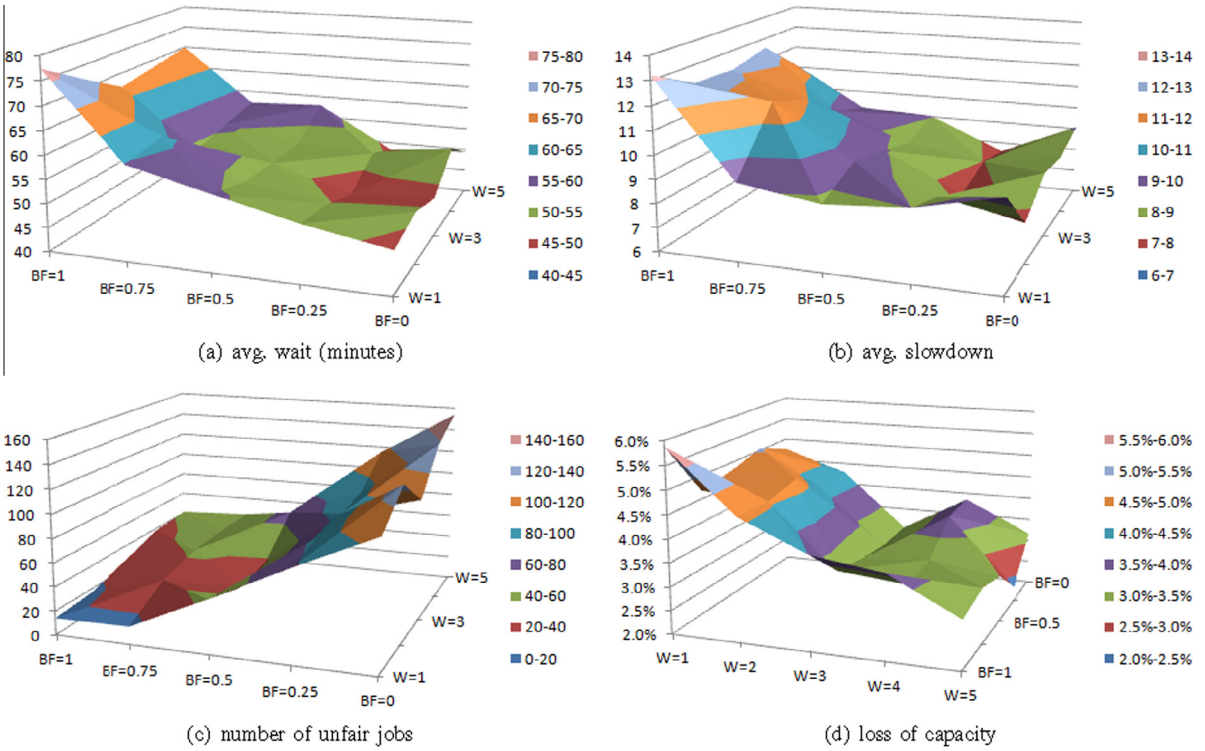



**Fig. 4.** Results of metrics balancing (SP2).

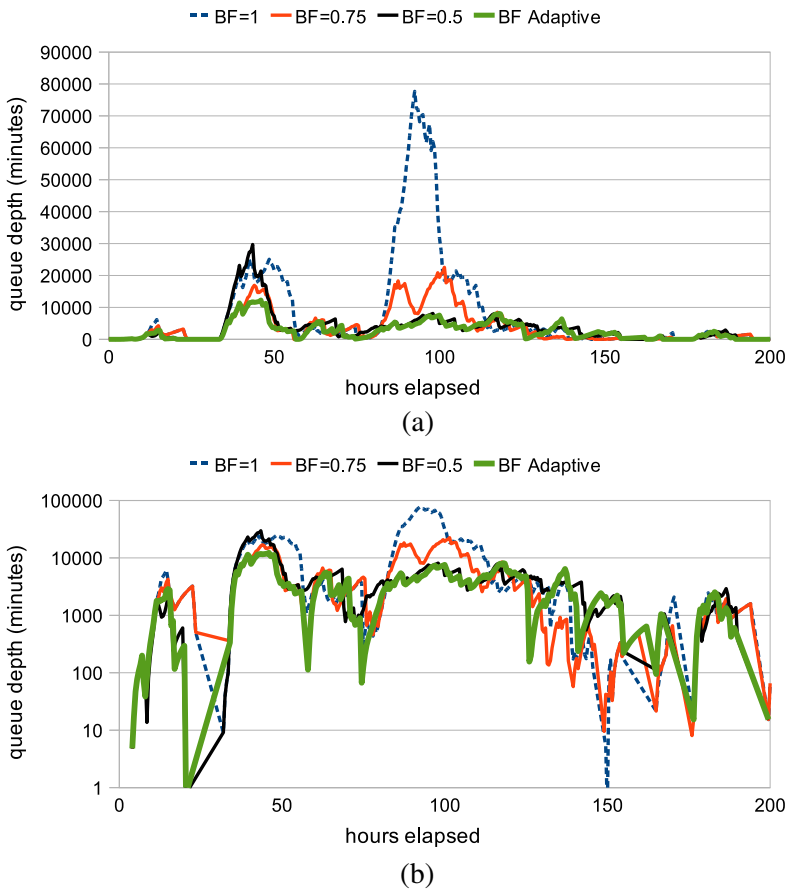



**Fig. 5.** Results of adaptively tuning the balance factor (BG/P).

As shown in the figures, BF = 1 is generally on the top of all the other lines, meaning that FCFS has the deepest queue, especially with a waiting job burst round 100 h from time zero. The policy with BF = 0.75 does better than FCFS in dealing with the job burst; the maximum queue depth is only 1/4 of that of FCFS (around the 100th hour). The policy with BF = 0.5 does even better; the maximum depth is less than 1/8 that of the FCFS. When the queue is not deep, however, FCFS does not do badly, sometimes performing even better than the other two cases; see Fig. 5b at elapsed hours around 150. This fact further encourages us to adaptively tune the balance factor such that a high BF is used when the queue is not deep and BF is tuned to a smaller value when the queue is deep.

The fourth line (legend "BF Adaptive") represents the results from adaptive tuning of the balance factor during runtime. Here the tuning point is when the queue depth reaches 1000 min. This is set based on the whole month's average. (In practice, this threshold can be set by an average queue depth number of a recent period of time, e.g., the past month or week.) Specifically, when the queue depth is under 1000 min, the BF is set to 1; otherwise, the BF is set to 0.5. The experimental results are as expected: the overall queue depth of adaptive tuning is not only much better than FCFS but also slightly better than the case with BF set to 0.5. These results suggest that the adaptive method takes advantage of different policies at corresponding queue status, as is desired by our original design goal.

Fig. 6 shows similar results for the SP2 workload. The threshold is set to 400 min, which is rounded by the whole month average. On this smaller-scale system compared with the Intrepid BG/P, the absolute queue depth numbers are much smaller. But a similar trend can be seen in that most of the time the queue is not deep and there is a job burst from time to time. Similarly, FCFS results in the biggest burst, and policy with BF = 0.5 or BF Adaptive can mitigate the job burst substantially. In this figure the lines of BF Adaptive and BF = 0.5 overlap extensively; their overall performance difference is shown later in the paper.

*4.4.2. Monitoring of system utilization*

We next discuss experimental results demonstrating the impact on system utilization rate by adjusting the window size only. For simplicity, we examine only window sizes 1 and 4, which represent the base setting and an enlarged window allowing job reordering, respectively. Thus, the parameter $T_i$ is 1, and $\Delta$ is 4. Since the average utilization rate is decided by the total node-hours of all jobs and the makespan, different simulations will have the same average utilization rate if

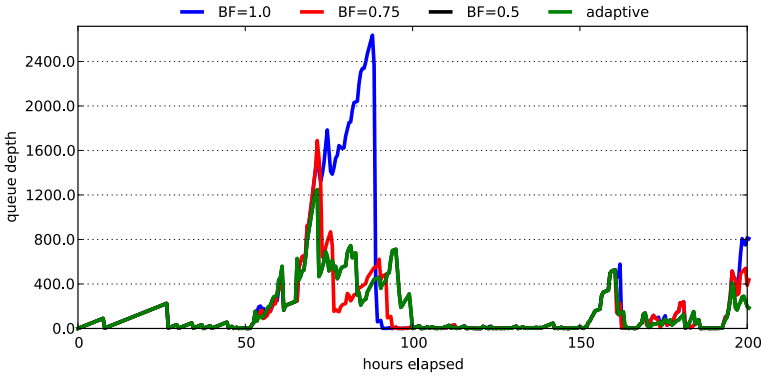



**Fig. 6.** Results of adaptively tuning the balance factor (SP2).

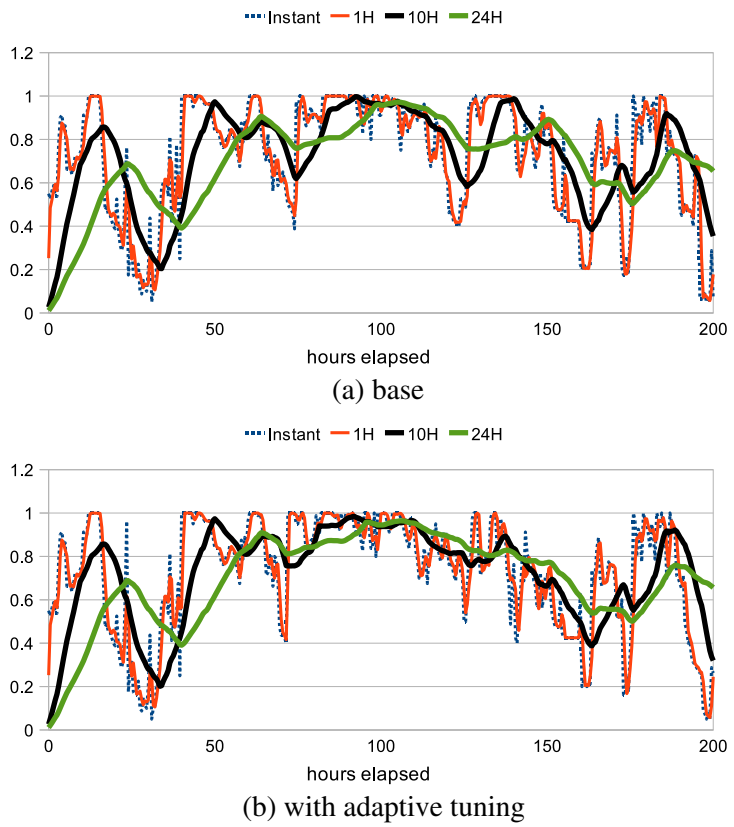
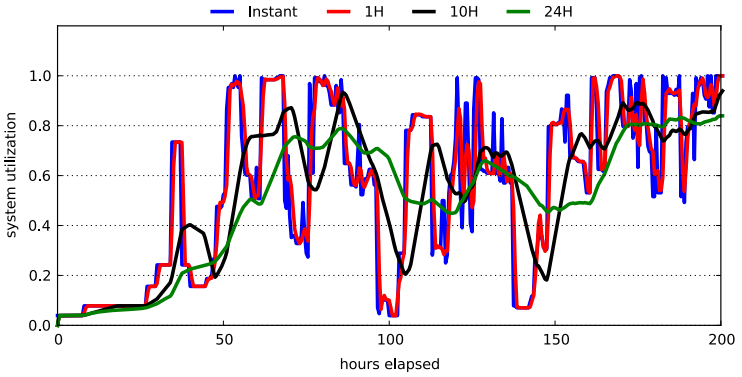


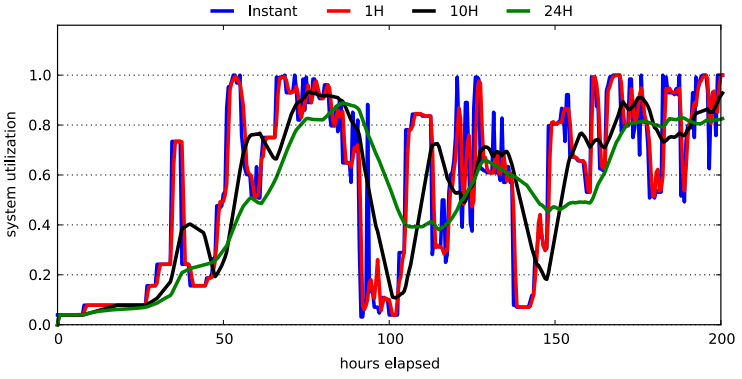

**Fig. 7.** Monitoring of system utilization (BG/P).

the workload does not saturate the machine. Thus, to see the impact on system utilization by window size, we monitor the instant utilization rate and some short-term averages. In each subfigure of Figs. 7 and 8, there are four plotted lines. A point on the "instant" line represents the instant system utilization rate. A point on the "1H" line represents the average system utilization rate during the past one hour. Similarly, the "10H" and "24H" are the average in the past 10 h and 24 h, respectively. We use 1H, 10H, and 24H to represent the "short-term," "mid-term," and "long-term" averages, respectively. The values are checked every 30 min. The *x*-axis represents the elapsed time in hours from time zero. We present only the first 200 h for visual clarity.

We tune the window size as follows. When the 10H line is above the 24H line, W is set to the base value 1; otherwise, W is set to a larger window (i.e., 4 in this experiment). The rationale is similar to the monitoring of a stock price: when a short-term average is below a long-term average, the price (or utilization rate here) is considered in a decline trend, and one needs to trigger a corresponding act to lift the trend. Here, our action is to enlarge the window size. Fig. 7a shows the base results




(a) base




(b) with adaptive tuning

**Fig. 8.** Monitoring of system utilization (SP2).

without adaptive tuning and Fig. 7b the results with adaptive tuning for BG/P. We get a higher 24H line during the stable time (especially from hour 50 to 150) compared with the cases without adaptive tuning. These results suggest that using adaptive window size tuning can stabilize the system load.

Fig. 8 shows similar experimental results for the SP2 workload with the same tuning scheme.

The quantified overall improvement for both the BG/P and SP2 workloads reflected by reduced loss of capacity is presented later in the paper.

*4.4.3. Two-dimensional policy tuning*

Besides tuning the balance factor and window size separately, we can also tune them simultaneously. We call this latter kind of tuning "two-dimensional policy tuning." Specifically, the balance factor and windows are both initialized to 1, and each follows its respective tuning strategy used in earlier experiments. The simulation results for BG/P are shown in Fig. 9.

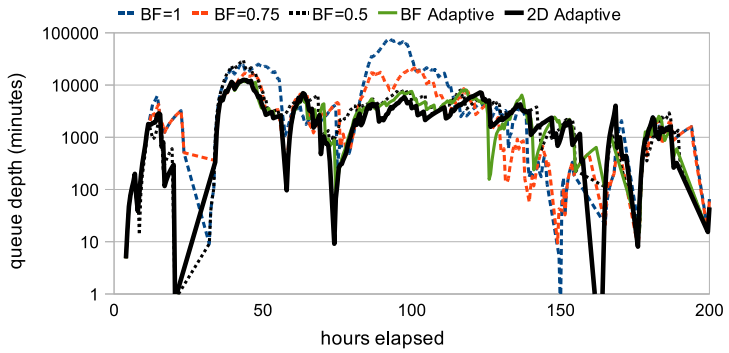
Fig. 9a shows the change of queue depth under 2D policy tuning compared with the original strategies. We can see that 2D adaptive tuning does even better than BF-only tuning. Not only does it perform well in avoiding queue depth bursts, but also it performs very well when the queue is not deep. For example, it outperforms all other cases between hours 150 and 200.

Fig. 9b shows the system utilization rate under the 2D policy tuning. We observe that in this figure both the 10H and 24H lines are more stable than in previous figures. That is, the system utilization rates are stabilized by 2D policy tuning. Moreover, evenly distributed system load helps spread out demands on the system infrastructure, both software and hardware.

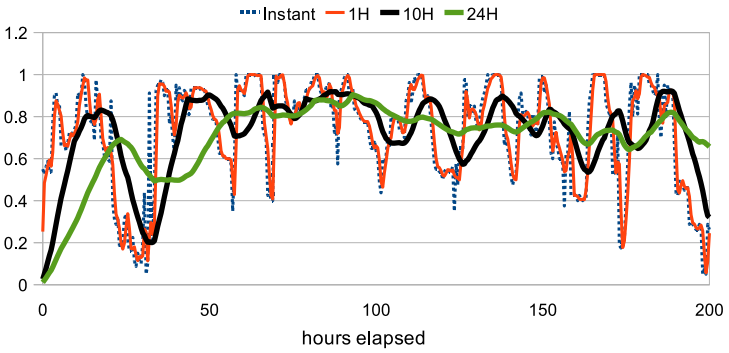
Fig. 10 shows similar results for the SP2 workload.

*4.4.4. Overall improvement of adaptive tuning*

Figs. 5–10 present monitored instant metric values to give a sense of how adaptive policy tuning schemes affect the metrics we are interested in. However, they cannot provide quantified values to measure the overall improvement brought by adaptive tuning. Table 2 and 3 present the overall improvement delivered by adaptive tuning on the metrics we mentioned earlier compared with several representative configurations without adaptive tuning. Table 2 shows results for the BG/P workload and Table 3 for the SP2 workload.




(a) queue depth




(b) system utilization rate

**Fig. 9.** Results of 2D policy tuning (BG/P).

The first column in each table lists a set of simulation cases with different configuration. The first case (BF = 1/W = 1) emulates basic FCFS with backfilling and one-by-one job allocation. The second to fourth cases represent some enhanced cases that either BF or W larger than 1. The last three cases are enhanced cases each representing one type of policy tuning scheme. The experimental results for average waiting, number of unfair jobs, and loss of capacity are shown in the tables.

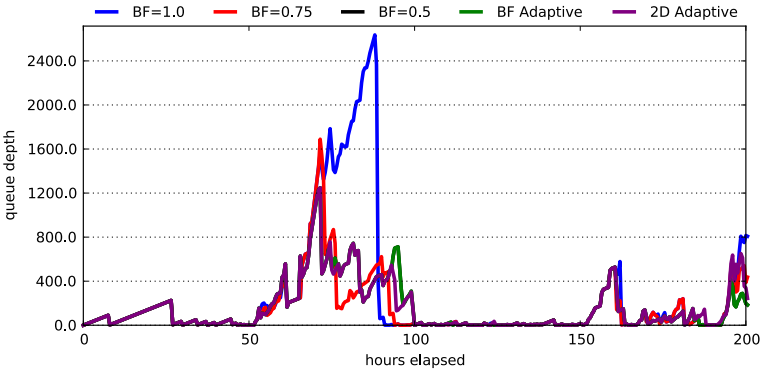
From the tables, we can see that all the enhanced cases are good for minimizing the average waiting time of the base case. The main contribution comes from using a BF value of 0.5, either fixing it to 0.5 or dynamically tuning it to 0.5. Note that increasing W only does help reducing average waiting but that the improvement is moderate.

All the enhanced cases hurt fairness by increasing the number of unfair jobs, differing only in the extent. Generally, using adaptive tuning can help limit the unfair job increases to a relatively low level. Hence, adaptive schemes are superior to the fourth case (BF = 0.5/W = 4) even though the latter has the minimum average waiting time (it also has the highest number of unfair jobs). Loss of capacity schemes are generally improved by the enhanced cases, with one or two exceptions for each workload. Again the adaptive schemes are also good for this metric.

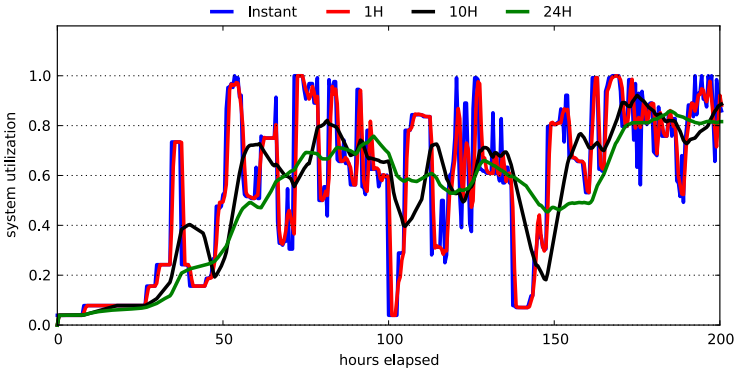
Overall, we can see that 2D adaptive schemes outperform others in an integrated fashion. Particularly for BG/P workloads, 2D adaptive schemes improve the average waiting time and loss of capacity by 71% and 23%, respectively, only doubling the unfair jobs compared with the base configuration. The fourth case (BF = 0.5/W = 4) achieves similar improvement but results in four times more unfair jobs.

To better illustrate the improvement of new schedule schemes compared with traditional schemes, we used the data shown in Table 2 and 3 to create 3D radar charts, which visualize the improvement from different aspects and as a whole; see Fig. 11. To this end, each metric is normalized to a score in the range of 0–100. Specifically, the maximum value of each metric is set to 100, and other values are normalized based on the ratio to the maximum value. Thus, for each metric, the lower the score is, the better. As a whole, the smaller the triangle is, the better the corresponding scheme performs. We can see that the adaptive schemes (solid lines) are generally within the range of statical schemes (dashed lines). with some exceptions.

To further quantify the relative gains, we calculate the relative gain of each scheduling scheme compared with traditional FCFS plus backfilling (BF = 1, W = 1). The calculation is based on the normalized score used in the radar charts. The relative gain of scheme B to scheme A is calculated as $G = 100\% * (S_A - S_B)/S_A$, where $S_A$ and $S_B$ represent scores of A and B. Fig. 12 shows the results. Each schedule scheme is represented by a bar group, each containing several bars. Each bar represents a relative gain calculated as a weighted average among three metrics: $w_1 * G_{wait} + w_2 * G_{unfair} + w_3 * G_{LoC}$, where $w$ is the




(a) queue depth




(b) system utilization rate

**Fig. 10.** Results of 2D policy tuning (SP2).

**Table 2**
Improvement of adaptive tuning (BG/P).

| Configuration | Avg. wait (min) | Unfair # | LoC (%) |
|---|---|---|---|
| BF = 1/W = 1 | 245.2 | 10 | 15.7 |
| BF = 1/W = 4 | 221.6 | 18 | 12.4 |
| BF = 0.5/W = 1 | 77.9 | 39 | 15.8 |
| BF = 0.5/W = 4 | 70.4 | 49 | 13.9 |
| BF Adapt. | 74.1 | 21 | 12.8 |
| W Adapt. | 198.1 | 16 | 11.9 |
| 2D Adapt. | 71.3 | 19 | 12.1 |

**Table 3**
Improvement of adaptive tuning (SP2).

| Configuration | Avg. wait (min) | Unfair (#) | LoC (%) |
|---|---|---|---|
| BF = 1/W = 1 | 77.4 | 14 | 5.88 |
| BF = 1/W = 4 | 68.4 | 38 | 5.38 |
| BF = 0.5/W = 1 | 55.8 | 45 | 4.82 |
| BF = 0.5/W = 4 | 52.5 | 67 | 3.21 |
| BF Adapt. | 55.2 | 43 | 4.29 |
| W Adapt. | 64.3 | 26 | 3.62 |
| 2D Adapt. | 53.8 | 33 | 4.15 |

weights and $w_1 + w_2 + w_3 = 1$. We examine several weighting combination: EQUAL, WAIT+, UNFAIR+, and LOC+. EQUAL means the three metrics are equally weighted ($w_1 = w_2 = w_3 = 1/3$). WAIT + means we weight the metric average waiting time more than the other two, setting $w_1 = 0.5$ and $w_2 = w_3 = 0.25$. Similarly, the UNFAIR + and LoC + also represent

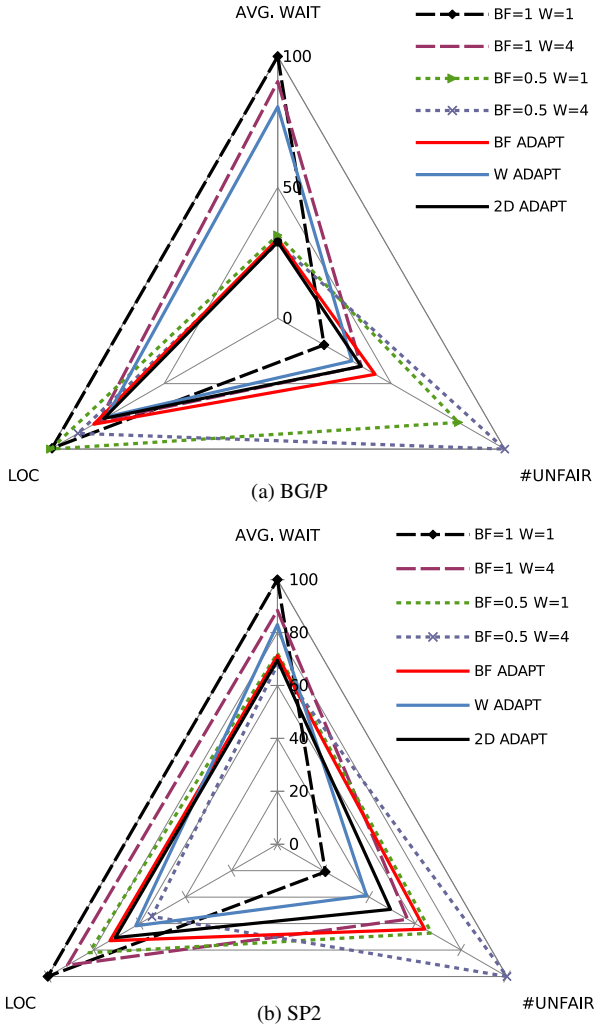



**Fig. 11.** Radar charts showing relative improvement.

weighting the respective metrics more than the other two. The bar with legend "average" represents the average value of the other four bars in that group.

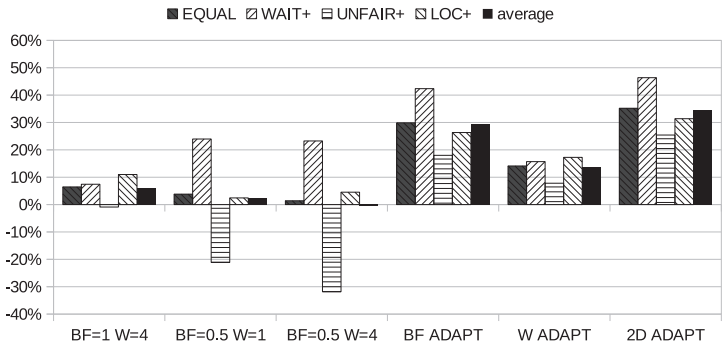
From Fig. 12a we can see that for the BG/P workload the adaptive scheduling schemes, especially BF-adpat and 2D-adapt, perform significantly better than ones without adaptive policy tuning. In particular, 2D-adapt achieves an average improvement of nearly 35%. These results indicate that adaptive policy tuning can help balance different metrics in order to achieve an overall good performance. The statical configurations also has moderate improvement especially when the average waiting time is weighted more than other metrics. The negative bars indicate that when fairness is weighted more than others, the statical schemes are worse than FCFS/backfill. However, adaptive policy tuning has no such issue.

Fig. 12b shows similar trend for the SP2 workload. W-adpat and 2D-adapt perform best: the average gain is up to 15%.

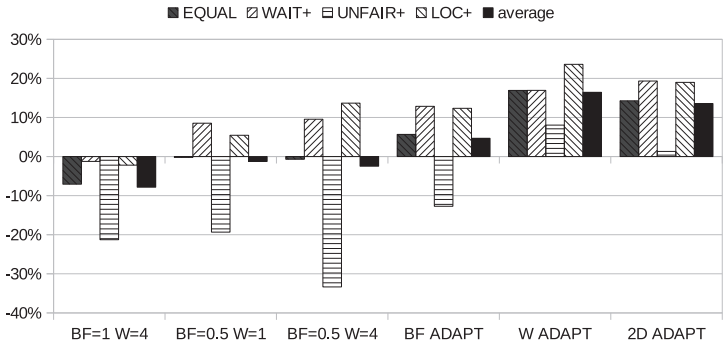
### 4.5. Scheduling costs

Compared with traditional FCFS plus backfilling scheduling, our approach admittedly incurs more overhead. The test results show that introducing the balance factor results in no overhead. Instead, the most overhead is caused by increasing the window size. Thus, we ran experiments to test the average running time per scheduling iteration for each window size. The scheduling algorithms were implemented in Python, and the tests are run on a Linux desktop machine with an Intel 2.4 GHz CPU.

Table 4 shows the test results. From the data, we can see that the running time per iteration increases substantially as the window size gets larger. We also note that the absolute value for the BG/P workload is much larger than for the SP2. The reason is mainly that our BG/P scheduler uses contiguous job allocation on a very large node set (40,960 nodes), which is




(a) BG/P




(b) SP2

**Fig. 12.** Quantified relative gains.

**Table 4**
Runtime per scheduling iteration (sec).

| Window size | BG/P | SP2 |
|---|---|---|
| W = 1 | 0.021 | 0.00876 |
| W = 2 | 0.034 | 0.00933 |
| W = 3 | 0.069 | 0.00979 |
| W = 4 | 0.117 | 0.0109 |
| W = 5 | 0.584 | 0.0147 |

very time-consuming. Since the scheduling iteration is triggered about every 10 s in real systems (e.g., Cobalt uses the 10-s setting), a scheduling iteration less than 1 s is affordable. In sum, for cluster systems using noncontiguous job allocation, the overhead from our approach can be ignored, and the window size can be even larger than the number we have tested. For large-scale systems such as BG/P using contiguous job allocation, our approach can work well with a window size up to 5, improving the resource allocation considerably.

## 5. Summary

The job scheduler is an important component of a high-performance supercomputer system. The job scheduler is responsible for prioritizing queued jobs in a manner that allows the system to satisfy the performance objectives of different users while simultaneously making efficient use of system resources. In this paper we presented a novel job scheduling framework with two important features. First, we provided a metric-aware job scheduling mechanism that enables the scheduler to balance goals based on targeted system metrics such as job waiting time, fairness, and system utilization. Second, we extended the system to support dynamic scheduling policy tuning based on feedback information from monitored metrics, thereby adapting to varying workload characteristics.

We evaluated our approach by simulating real workloads from both a large-scale Blue Gene/P system (at Argonne) and a medium-scale SP2 system (at SDSC). The experimental results show that using a proper balance factor and job allocation window can achieve significant performance improvement compared with the prevailing FCFS plus backfilling policy. With

adaptive policy tuning, the targeted metrics are further improved in a balanced and sustainable fashion because the scheduling policies are adapted to varying workloads. In particular, the two-dimensional policy tuning is demonstrated to be the most effective approach in our experiments.

In the future, we plan to extend our work in a few directions. First, we plan to extend the range of balanced metrics, especially adding some metrics in the system cost category. For example, energy consumption and reliability are two major issue in building extreme scale system where related metrics can be considered as system costs to be balanced in the entire scheduling process. These facilities can be used by our previous work regarding power-aware job scheduling [29] and failure-aware job scheduling [22]25. Meanwhile, we plan to use feedback-control models and multi-objective optimization to enhance current resource management mechanism. Also, we plan to optimize the algorithm used in the window-based job selection so that we can enlarge the window size to achieve more enhancement.

## Acknowledgements

This work is supported in part by National Science Foundation grants CNS-0720549 and CCF-0702737. The work at Argonne National Laboratory is supported by the U.S. Department of Energy under Contract DE-AC02–06CH11357.

## References

[1] Cobalt resource manager, http://trac.mcs.anl.gov/projects/cobalt.
[2] Moab HPC suite, http://www.adaptivecomputing.com/products/moab-hpc-suite-basic.php.
[3] Parallel workload archive, http://www.cs.huji.ac.il/labs/parallel/workload.
[4] I. Ababneh, S. Bani-Mohammad, A new window-based job scheduling scheme for 2D mesh multicomputers, Simulation Modelling Practice and Theory 19 (1) (2011) 482–493.
[5] P. Blevins, C. Ramamoorthy, Aspects of a dynamically adaptive operating system, IEEE Transactions on Computers C25 (7) (1976) 713–725.
[6] N. Desai, R. Bradshaw, C. Lueninghoener, A. Cherry, S. Coghlan, W. Scullin, Petascale system management experiences, in: Proc. USENIX. Large Installation System Administration Conference (LISA), 2008.
[7] Y. Etsion, D. Tsafrir, A short survey of commercial cluster batch scheduler, Technical report 2005–13, The Hebrew University of Jerusalem, 2005.
[8] S. Grothklags, A. Streit, On the comparison of cplex-computed job schedules with the self-tuning dynp job scheduler, in: Proc. of IEEE International Parallel & Distributed Processing Symposium, 2004.
[9] L. Goh, B. Veeravalli, Design and performance evaluation of combined first-fit task allocation and migration strategies in mesh multiprocessor systems, Parallel Computing 34 (9) (2008) 508–520.
[10] D. Jackson, Q. Snell, M. Clement, Core algorithms of the Maui scheduler, Job Scheduling Strategies for Parallel Processing, vol. 2221, LNCS, 2001, pp. 87–102.
[11] D. Krishnamurthy, M. Alemzadeh, M. Moussavi, Towards automated HPC scheduler configuration tuning, Concurrency and Computation: Practice and Experience 23 (15) (2011) 1723–1748.
[12] C. Lu, J.A. Stankovic, G. Tao, S.H. Son, Design and evaluation of a feedback control EDF scheduling algorithm, in: Proc. of IEEE Real-Time Systems Symposium (RTSS), 1999.
[13] A. Mu'alem, D. Feitelson, 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.
[14] A. Rafaeli, G. Barron, K. Haber, The effects of queue structure on attitudes, Journal of Service Research 5 (2) (2002) 125–139.
[15] D. Raz, H. Levy, B. Avi-Itzhak, A resource-allocation queueing fairness measure, in: Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2004.
[16] G. Sabin, G. Kochhar, P. Sadayappan, Job fairness in non-preemptive job scheduling, in: Proc. of International Conference on Parallel Processing (ICPP), 2004.
[17] E. Shmueli, D.G. Feitelson, Backfilling with lookahead to optimize the packing of parallel jobs, Journal of Parallel and Distributed Computers 65 (9) (2005) 1090–1107.
[18] S. Srinivasan, R. Kettimuthu, V. Subramani, P. Sadayappan, Selective reservation strategies for backfill job scheduling, Job Scheduling Strategies for Parallel Processing, vol. 2357, LNCS, 2002, pp. 55–71.
[19] A. Streit, A self-tuning job scheduler family with dynamic policy switching, Job Scheduling Strategies for Parallel Processing, vol. 2357, LNCS, 2002, pp. 1–23.
[20] A. Streit, Evaluation of an unfair decider mechanism for the self-tuning dynp job scheduler, in: Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2004.
[21] W. Tang, N. Desai, D. Buettner, Z. Lan, Job scheduling with adjusted runtime estimates on production supercomputers, Journal of Parallel & Distributed Computing (JPDC) 73 (7) (2013) 926–938.
[22] W. Tang, Z. Lan, N. Desai, D. Buettner, Fault-aware, utility-based job scheduling on Blue Gene/P systems, in: IEEE International Conference on Cluster Computing (Cluster), 2009.
[23] W. Tang, Z. Lan, N. Desai, D. Buettner, Y. Yu, Reducing fragmentation on torus-connected supercomputers, in: Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2011.
[24] W. Tang, D. Ren, Z. Lan, N. Desai, Adaptive metric-aware job scheduling for production supercomputers, in: Proc. of International Conference on Parallel Processing Workshops (ICPPW), 2012.
[25] W. Tang, Z. Lan, N. Desai, D. Buettner, Automatic and coordinated job recovery for high performance computing, in: Proc. of IEEE Workshop on Many-Task Computing on Grids and Supercomputers, 2010.
[26] W.A. Ward, C.L. Mahood, J.E. West, Scheduling jobs on parallel systems using a relaxed backfill strategy, Job Scheduling Strategies for Parallel Processing, vol. 2357, LNCS, 2002, pp. 88–102.
[27] Y. Yuan, G. Yang, Y. Wu, W. Zheng, PV-EASY: a strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling, in: Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010.
[28] Y. Zhang, H. Franke, J. Moreira, A. Sivasubramaniam, Improving parallel job scheduling by combining gang scheduling and backfilling techniques, in: Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2000.
[29] Z. Zhou, Z. Lan, W. Tang, N. Desai, Reducing energy costs for IBM Blue Gene/P via power-aware job scheduling, in: Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2013.