Adaptive Metric-Aware Job Scheduling for Production Supercomputers

Wei Tang,* Dongxu Ren,* Zhiling Lan,* Narayan Desai†

*Department of Computer Science, Illinois Institute of Technology
Chicago, IL 60616, USA
{wtang6, dren1, lan}@iit.edu

†Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
desai@mcs.anl.gov

***Abstract*—Job scheduling is a critical and complex task on large-scale supercomputers where a scheduling policy is expected to fulfill amorphous and sometimes conflicting goals from both users and system owners. Moreover, the effectiveness of a scheduling policy is dependent on workload characteristics which vary from time to time. Thus it is challenging to design a versatile scheduling policy that is effective in all circumstances. To address this issue, we propose an adaptive metric-aware job scheduling strategy. First, we propose 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 enhance the scheduler to adaptively adjust scheduling policies based on feedback information of monitored metrics at runtime. We evaluate our design using real workloads from supercomputer centers and demonstrate that our scheduling mechanism can significantly improve system performance in a balanced, sustainable fashion.**

I. Introduction

Job scheduling is a critical task for 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. Moreover, production supercomputing centers start to face new challenges in scheduling, such as avoiding failure interrupts and energy efficiency. All these considerations, which are quantified by system metrics, are related but often conflicting with one another. Even worse, the priorities differ from machine to machine and from time to time, which further complicates 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. Some policies such as "max expansion factor first" make a compromise in terms of these goals, but in a static fashion. Essentially, these approaches attempt to use a fixed combination of different priorities, to address the problem. Many traditional schedulers provide mechanisms to switch between these policies when particular boundary conditions are encountered; however, this approach only provides a coarse ability to refine goal-based priorities.

Moreover, user satisfaction and system performance are not considered in a holistic fashion. Typically, job prioritizing and resource allocation are separated into two subsequent phrases 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; indeed, these resources could be used to to execute other low priority jobs for improving system performance. This kind of resource draining causes external fragmentation. While backfilling helps in this case, it only mitigate for fragmentation already created by this division [20].

Another issue of job scheduling is about dynamic workload. Even though we may identify a policy to achieve our integrated goals well for one workload, the policy may completely fail for a different workload. Although event-driven simulations can be used to evaluate the aggregate effect of a scheduling policy on a 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 two-fold. 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 in to the targeted performance metrics in to account when configure a specific scheduling policy. Moreover, the prioritized jobs are allowed be altered in a limited fashion during the resource allocation phase. This 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 interested 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 have implemented our design in the production resource management system called Cobalt [1]. We evaluated our

1530-2016/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPPW.2012.17

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 II discusses some related work. Section III describes our methodology. Section IV illustrates our experimental results. Section V summarizes the paper with some discussion on future work.

II. Related work

Balancing multiple scheduling objectives are supported by some production job schedulers. One example is Maui scheduler [8] which uses a number of weighted and combined parameters to prioritizing 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. [10] 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 simulation on historical workload 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 multiple scheduling considerations into account. Cobalt provides an event-driven simulator to guide the design of a appropriate utility function [21]. While our new 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.

Dynamic scheduling policy tuning can be found in some existing work. Streit et al. [18][19][7] proposed a self-tuning "dynP" job scheduler which can tune queuing policy dynamically during run time. Our work shares similar motivation but also differs from this previous work. First, 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. Further, in our work, both the queuing policy as well as 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 et al. [4] proposed to use feedback information to adjust the schedule in general-purpose operating systems in the form of multi-level feedback queue scheduling. Lu et al. [11] designed and evaluated a feedback control earliest-deadline-first scheduling algorithm for scheduling in real-time system. However, in parallel job scheduling, existing schedulers usually use "open loop" scheduling algorithms in which policies are not adjusted based on continuous feedback. In fact, in production supercomputers, the dynamics of the workload is amorphous and influential to the effectiveness of a scheduling policy. Thus utilizing feedback information of workload change is beneficial in selection of a scheduling policy. It is one of the motivations of our work. Meanwhile, our work differ from existing feedback-based scheduling work that we do not adjust the schedules directly but instead adjust the scheduling policy which will indirectly change the schedules.

Etsion et al. [6] reported that the prevalent default scheduler setting is FCFS, and in those management suites that also support backfilling, the governing scheme used is EASY [12]. However, there exists numerous work to enhance either FCFS or EASY backfilling. Ababneh et al. [3] proposed an enhancement to FCFS which uses a window of consecutive jobs from which it selects jobs for allocation and execution. Shmueli et al. [16] optimized the packing of backfilling jobs by looking ahead into the queue. Srinivasan et al. [17] designed a selective reservation strategy for backfilling jobs which acts in the middle of EASY backfilling and conservative backfilling. Ward et al. [23] proposed a relaxed backfilling scheme that allows backfilled jobs moderately delay reserved jobs. Our metric-aware scheduling is also an enhancement for FCFS and backfilling. Using the balance factor we can tune FCFS toward SJF at a wanted extent. The window-based allocation also provided more flexibility in job allocation and backfilling. Moreover, our policy can be dynamically tuned to adapt the change of workload.

Fairness is an important metric in measuring a job queuing system. While it tends to be ignored by many scheduling studies, there are a number of existing work focusing on it. Rafaeli et al. [13] demonstrated via 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. [14] proposed a resource allocation queuing fairness measure that accounts for both relative job seniority and relative service time. Sabin et al. [15] proposed an approach to assessing fairness in non-preemptive job scheduling. Their defined fairness is that no later arriving job should delay an earlier arriving job. Yuan et al. [24] proposed a more strict fairness model that no job is delayed by any jobs 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 to balance a variety of interested metrics including fairness.

III. Methodology

A. Design

The diagram of our design is shown in Figure 1, where there are 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 will be monitored. The feedback from metrics monitor will be sent back to metrics balancer so that it can adjust the weight of different factors to maintain the desired balance. This feedback process can be conducted manually or automatically.

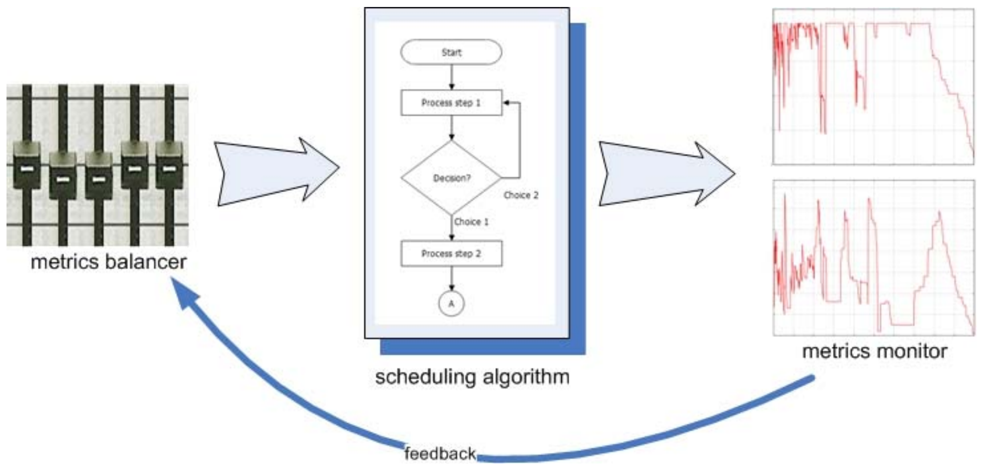


Fig. 1. Diagram of adaptive metric-aware job scheduling framework.

We call the process how the metrics balancer impacts the scheduling algorithm as "metric-aware job scheduling," and the process how the metrics monitor influences the adjustment to metrics balancer as "adaptive policy tuning." In following two sections we will present the detailed design of the metric-aware scheduling and adaptive policy tuning, respectively.

B. Metric-aware job scheduling

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

As mentioned earlier, user-centric metrics are mainly influenced by job queuing policies and system metrics are correlated with resource allocation. Existing scheduling scheme allocates jobs one by one in their priority order. The problem is that when allocating one job at one time, it can achieve best allocation for that single job but neglects the potential impact to other jobs in the queue (one simple example is shown in Figure 2). Thus we propose to schedule and allocate a group of jobs at one time so that the job allocation can result in better system utilization. We call the number of jobs that are allocated at one time as "window size (W)," which varies from one to up to the number of queued jobs. Intuitively, the larger the window size is, better job allocation can be achieved but 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 described as follows.

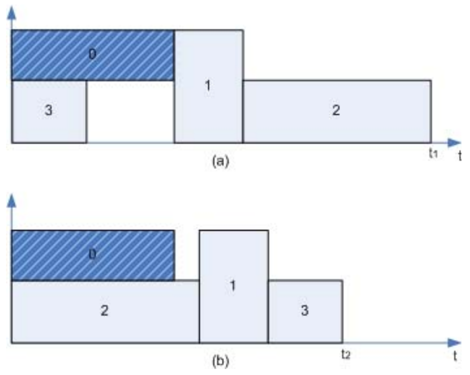


Fig. 2. An example showing the limitation of scheduling and allocate jobs one by one. Job 0 is running, Jobs 1, 2, and 3 are waiting . (a) schedule and allocate job one by one in priority order; (b) schedule and allocate in a group as a whole. Apparently (b) achieves better system utilization.

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

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

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 value 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 walltime, mapping the values to [0, 100],

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

where $walltime_{max}$, $walltime_{min}$ are the maximum and minimum walltime times of the current queue, respectively, and $walltime_i$ is the walltime of job i. If there is only one job in the queue, S_r is set to 0.

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

$$S_p = BF \times S_w + (1 - BF) \times S_r, \quad (3)$$

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

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 highest utilization rate.

Step 6: Add another pass through the queue to try to backfill remained jobs, conforming 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, all reservations cannot be delayed [12].

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 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 FCFS plus backfilling [6].

C. 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 is also dependent on the workload characteristic which varies from time to time. To mitigate the impact of workload change, we introduce an adaptive policy tuning mechanism as a supplement to metric-aware scheduling. Basically, we enable the scheduler dynamically change scheduling policy (by tuning the balance factor and window size) based on the feedback of 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 $< T, T_i, \mathit{\Delta}, M, Th, E_p, E_m, C_i >$ need to be determined. In this tuple, T is the tunable arguments to the scheduling policy. In this study, T is either BF or W described earlier. T_i is the initial value of T which determines the initial scheduling policy. $\mathit{\Delta}$ 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 thresholds 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. C_i defines the interval between checking points at which metrics are checked and policies are tuned. Table I summarizes these parameters.

TABLE I
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
$\mathit{\Delta}$	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 $\mathit{\Delta}$	M reaches Th
E_m	event triggering T minus $\mathit{\Delta}$	M reaches Th reversely
C_i	checking interval	30 minutes

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

The process of the adaptive policy tuning mechanism is described in Algorithm 1. The whole process begins with initializing tunables. At each check point the scheduler will do following. First, it gets values of all monitored metrics, then compares the checked value with the preset thresholds to find out whether policy tuning event is triggered. If yes, it then adjust tunable based on the event type. This metric checking logic is added before the existing job scheduling function.

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
            $T = T + \mathit{\Delta}$ ;  // increase tunable by $\mathit{\Delta}$
        end
        if $e == E_m$ then
            $T = T - \mathit{\Delta}$ ;  // decrease tunable by $\mathit{\Delta}$
        end
        last_checked = now ;   // reset check clock
    end
    schedule_jobs(T) ; // do real scheduling stuff
    sleep($SchedInterval$)
end
```

IV. EXPERIMENT

We conducted simulation based experiments on real workloads from the production Blue Gene/P system (named Intrepid [5]) at Argonne National Laboratory, which is a large-scale MPP system with 40,960 computing nodes (163,840 cores). We implemented our scheduling algorithms in Cobalt's event-driven simulator [21]. Before presenting the experimental results, we introduce some evaluation metrics.

A. Evaluation metrics

Folloing 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.
- *Queue Depth (QD).* This metric is proposed for this work which is also related to job waiting and queuing efficiency. Specifically, the queue depth at a given moment is measured by the sum of waiting times of *all* the jobs in the current queue have endured so far to that moment. This metric is good for real-time monitoring since it reflects the status of the queue. A high queue depth may be caused by either there are a large number of jobs waiting or there are some jobs enduring very long waiting, or both.
- *Fairness.* To assess fairness, we assign a "fair star 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 conducted a simulation of scheduling under current scheduling policy and get when the job will be started. We count the number of unfair job to assess the overall fairness. This metric has been used in existing work [15].
- *System utilization rate.* This metric represents the ratio of utilized (or delivered) node-hours to total available node-hour during the checked period of time. Usually it is refer 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 [9].
- *Loss of capacity (LoC).* This 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 the system has N nodes and m scheduling events, which 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 δ_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:

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

  Loss of capacity reflects the costs of resource fragmentation. This metric has been used in our previous work [22] and a similar one has been used by Zhang et al. [25].

B. Results of metrics balancing

In order to show the effect of metric-aware scheduling, we run simulations with various values for balance factor and window size. Specifically, the balance factor is 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 job allocation window varies from 1 to 5. Window size 1 emulates the conventional, one-by-one job allocation scheme.

Figure 3 shows the results. As shown in the Figure 3(a), the average waiting time declines as the balance factor decreases. Especially, the waiting time drops significantly as BF is tuned from 1 to 0.5, and kept at the same level when BF decreases further. This suggests that prioritizing jobs considering both job age (waiting time) and job length (expected running time) can enhance scheduling efficiency dramatically compared to pure FCFS (considering job age only). Theoretically, perfect short job first should lead to best average waiting time. But the results show that as BF is tuned from 0.5 to 0, there is no clear improvement, which suggests that weighting too much on job length brings some bad effects such as large job starvation. This limits the further improvement in average waiting time. The window size does not show clear impact on average waiting time as the different lines are staggered along each other. But for FCFS, using a window size larger than 1 can achieve considerable improvement on average waiting time (by more than 10%) compared with traditional FCFS plus backfilling scheduling policy.

Figure 3(b) shows the fairness results. Clearly, the number of unfair jobs increases as the policy approaches SJF. A general trend appears 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. Figure 3(c) shows the results in terms of loss of capacity. Since this metric is influenced by window size more than balance factor, we put the window size to x-axis and the balance factor to legend in order to show the trends. We can see that when BF is no less than 0.5, the loss of capacity has the decreasing trend as the window size increases, meaning that enlarging job allocation window can help to utilize computing nodes more efficiently. This effect is not shown when the queuing policy gets closer to SJF; for example, see the lines where BF=0.25 or BF=0. In these set of data, we can learn that we can improve "efficiency" (either for queuing or utilizing resources) by moderately sacrificing "fairness," vice versa.

C. Results of adaptive tuning

Now we will examine the effect of adaptive policy tuning. We begin with trying to balance queuing efficiency and fairness by tuning the balance factor BF only. In this set of experiments we fix the window size to the default number one.

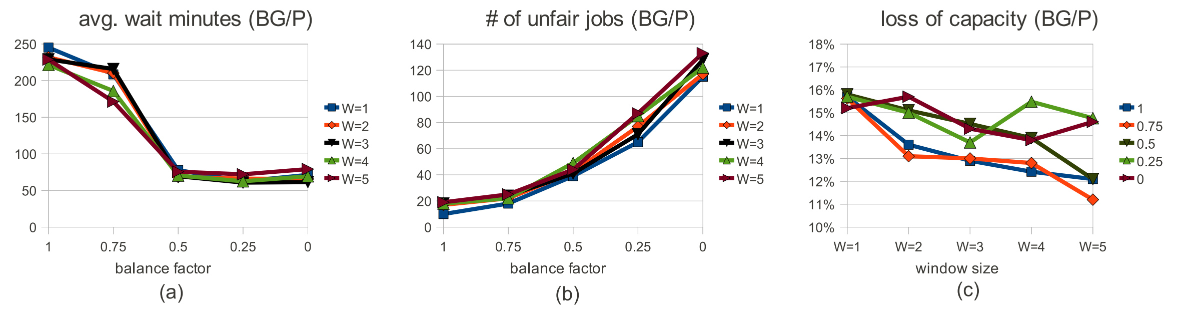
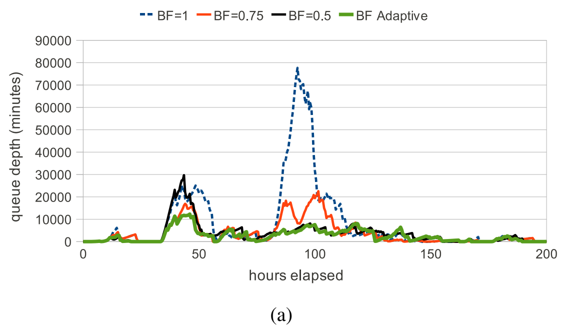


Fig. 3. The effect of using balance factor and window size.



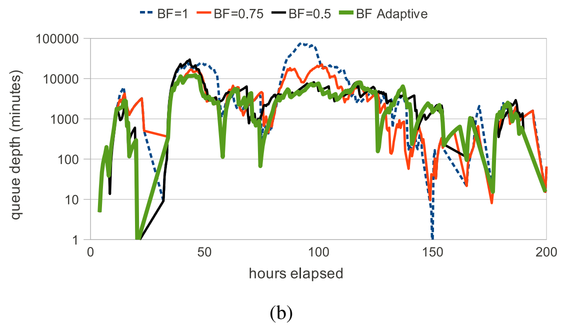


Fig. 4. Results of adaptively tuning balance factor.

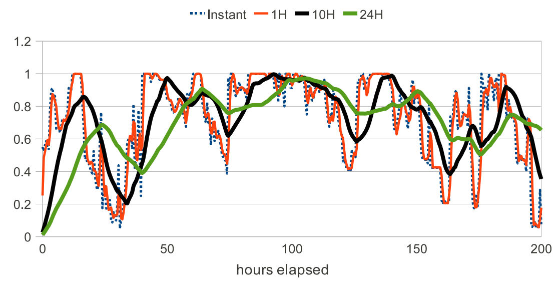
1) Monitor of queue depth: Figure 4 shows the variation of the queue depth, measured by instant aggregate waiting time among all queuing jobs, over time. Figure 4(a) shows the data in normal scale and Figure 4(b) presents the logarithm scale on the same data sets helping to zoom the data characteristic under small scale.

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

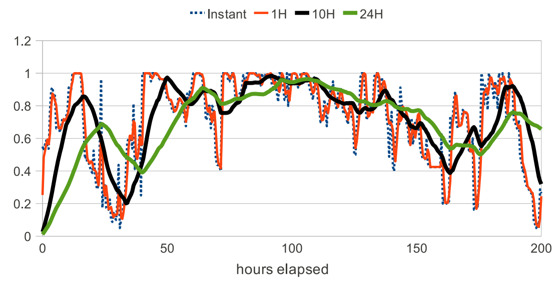
As shown in the figures, BF=1 is generally on the top of all other lines, meaning FCFS has the deepest queue, especially with a waiting job burst round 100th hours from the time zero. The policy with BF=0.75 does better than FCFS in dealing with the job burst; the maximum queue depth is only $1/4$ of the FCFS one (around the 100th hour). The policy with BF=0.5 does even better; the maximum depth is only less than $1/8$ of the FCFS one. However, when the queue is not deep, the FCFS does not bad, sometimes even better than the other two cases (see Figure 4(b) at elapsed hours around 150). This fact further encourage us to adaptively tune balance factor with the rationale that using a high BF when the queue is not deep and tuning the BF to a smaller value when the queue is deep.

The fourth line (legend "adaptive") represents the results from adaptive tuning the balance factor during runtime. Here the tuning point is when the queue depth reaches 1000 minutes. 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. past one month or one week.) Specifically, when the queue depth is under 1000 minutes, the BF is set to 1; otherwise, the BF is set to 0.5. The experimental results are as expected that 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. This suggests that adaptive method takes advantage of different policies at corresponding queue status, which is desired by our original design goal.

2) Monitor of system utilization: This part of experimental results demonstrates the impact on system utilization rate by adjusting window size only. For simplicity, we only examine window size 1 and 4 which represent the base setting and an enlarged window allowing job reordering, respectively. Thus, the parameter T_i is 1, Δ is 4. Since the average utilization rate is decided by the total node-hour of all jobs and the makespan, so different simulations will get the same average utilization rate if the workload does not saturate the machine. Thus, in order to see the impact on system utilization by window size, we monitor the instant utilization rate and some short-term

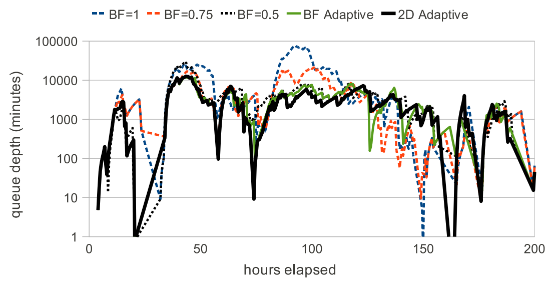


(a) base

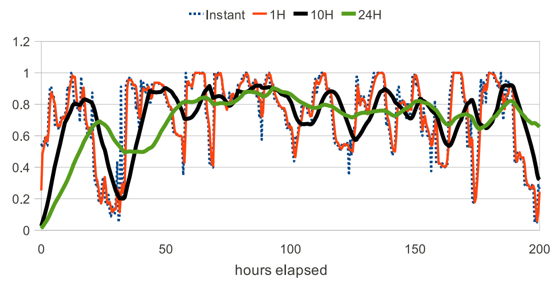


(b) with adaptive tuning

Fig. 5. Monitoring of system utilization.



(a) queue depth



(b) system utilizatin rate

Fig. 6. Results of 2D policy tuning.

averages. As in each subfigure, 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" the average in the past 10 hours and 24 hour, respectively. The values are checked at every 30 minutes. The x-axis represents the elapsed time in hour from time zero. Similar to some of the previous figures, we only present the first 200 hours' plots for visual clarity.

Now we will examine the effect of tuning window size. The configured triggering event is as follows: when 10H line is above the 24H line, W is set to 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, so that we need to trigger corresponding act to lift the trend. Here, our action is to enlarge the window size. Figure 5(a) shows the base results without adaptive tuning. Figure 5(b) shows the results with adaptive tuning. As show in the figure, we get a higher 24H line during the stable time (especially from hour 50 to 150) compared to the cases without adaptive tuning. This suggests that using adaptive window size tuning can stabilize system load.

3) Two-dimensional policy tuning: Besides we tune balance factor and window size separately, we can also tune them collectively. We call the latter kind of tuning as two-dimensional policy tuning. Specifically, the balance factor and windows are both initialized to 1 and each of them follows their respective tuning strategy used in earlier experiments. The simulation results are shown in Figure 6.

Figure 6(a) shows the queue depth change under 2D policy tuning compared with original strategies. We can see 2D adaptive tuning does even better than BF-only tuning. Not only 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 hour 150 and 200.

Figure 6(b) shows the system utilization rate under 2D policy tuning. We observe that in this figure both 10H and 24H lines are more stable than in previous figures. That is, system utilization rates are stabilized by 2D policy tuning. This indicates that 2D policy tuning helps to achieve a stable system load, which indicates job scheduling policies that are more robust to changing workload characteristics and variable goals over time. Moreover, evenly distributed system load helps to spread out demands on system infrastructure, both software and hardware.

4) Overall improvement of adaptive tuning: Figures 4 through Figure 6(b) presented monitored instant metric values in order 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. Following table data present the overall improvement delivered by adaptive tuning on the metrics we mentioned earlier compared with several representative configurations without adaptive tuning. Table II shows results.

The first column in each figure 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 2nd to 4th cases represent some enhanced cases that either BF or W larger than 1. The last three 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 table.

As shown in the table, we can learn that all the enhanced cases are good for minimizing average waiting time of the base case. The main contribution comes from using BF value 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 the improvement is moderate.

All the enhanced cases hurt fairness by increasing the number of unfair jobs, only differ in the extent. Generally, using adaptive tuning can help to limit the unfair job increases to a relatively low level. That's why 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 are mostly 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 2D adaptive schemes outperform others in an integrated fashion, which improves 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.

TABLE II
IMPROVEMENT OF ADAPTIVE TUNING

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

D. Scheduling cost

Compared with traditional FCFS plus backfilling scheduling, our approach admittedly incurs more overhead in running scheduling algorithm. We intended to test the performance cost of our algorithm to demonstrate its practicalness. The test results show that introducing balance factor results in little overhead. The most overhead is caused by increasing window size. Thus, we ran experiments to test the average running time per scheduling iteration for each window size. The scheduling algorithms are implemented in Python and the tests are run on a Linux desktop machine with an Intel 2.4GHz CPU.

Table III shows the test results. From the data, we can see the running time per iteration increases substantially as window size gets larger. Considering the scheduling iteration is triggered about every 10 seconds in real system (e.g. Cobalt uses the 10-second setting), a scheduling iteration

second is affordable. For large-scale system like BG/P using contiguous job allocation [22], our approach can work soundly configured with a window size no larger than 5, which can already deliver considerable improvement on resource allocation.

TABLE III
RUNTIME PER SCHEDULING ITERATION (SEC)

window size	time per iteration
W=1	0.021
W=2	0.034
W=3	0.069
W=4	0.117
W=5	0.584

V. 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 satisfy the performance objectives of different users while simultaneously making efficient use of system resources. In this paper we proposed a novel job scheduling framework that provides 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 of monitored metrics thereby adapting to varying workload characteristics.

We evaluated our approach with simulation, using real workloads from the large-scale Blue Gene/P system at Argonne. The experimental results show that using proper balance factor and job allocation window can achieve significant performance improvement compared with the existing prevailing FCFS with backfilling policy. With adaptively policy tuning, the targeted metrics are further improved in a 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.

This is our first step toward building an adaptive metric-aware job scheduling framework. In the future, we plan to extend the range of balanced metrics, especially adding some new metrics in the system cost category. For example, energy efficiency 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.

VI. ACKNOWLEDGMENTS

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 DOE Contract DE-AC02-06CH11357.

REFERENCES

[1] Cobalt resource mananger. *http://trac.mcs.anl.gov/projects/cobalt*.

[2] Moab HPC suite. *http://www.adaptivecomputing.com/products/moab-hpc-suite-basic.php*.

[3] I. Ababneh and S. Bani-Mohammad. A new window-based job scheduling scheme for 2D mesh multicomputers. *Simulation Modelling Practice and Theory*, 19(1):482–493, 2011.

[4] P. Blevins and C. Ramamoorthy. Aspects of a dynamically adaptive operating system. *IEEE Trans. on Computers*, C-25(7):713–725, 1976.

[5] N. Desai, R. Bradshaw, C. Lueninghoener, A. Cherry, S. Coghlan, and W. Scullin. Petascale system management experiences. In *Proc. USENIX Large Installation System Administration Conference (LISA)*, 2008.

[6] Y. Etsion and D. Tsafrir. A short survey of commercial cluster batch scheduler. In *Technical report 2005-13, the Hebrew University of Jerusalem*, 2005.

[7] S. Grothklags and 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.

[8] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing, LNCS 2221*, pages 87–102, 2001.

[9] J. P. Jones and B. Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In *Proc. of the Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1999.

[10] D. Krishnamurthy, M. Alemzadeh, and M. Moussavi. Towards automated HPC scheduler configuration tuning. *Concurrency and Computation: Practice and Experience*, 23(15):1723–1748, 2011.

[11] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[12] A. Mu'alem and 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):529–543, 2001.

[13] A. Rafaeli, G. Barron, and K. Haber. The effects of queue structure on attitudes. *Journal of Service Research*, 5(2):125–139, 2002.

[14] D. Raz, H. Levy, and 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.

[15] G. Sabin, G. Kochhar, and P. Sadayappan. Job fairness in non-preemptive job scheduling. In *Proc. of International Conference on Parallel Processing (ICPP)*, 2004.

[16] E. Shmueli and D. G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *J. Parallel Distrib. Comput.*, 65(9):1090–1107, 2005.

[17] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *Job Scheduling Strategies for Parallel Processing, LNCS 2357*, pages 55–71, 2002.

[18] A. Streit. A self-tuning job scheduler family with dynamic policy switching. In *Job Scheduling Strategies for Parallel Processing, LNCS 2357*, pages 1–23, 2002.

[19] 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.

[20] W. Tang, N. Desai, D. Buettner, and Z. Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[21] W. Tang, Z. Lan, N. Desai, and D. Buettner. Fault-aware, utility-based job scheduling on Blue Gene/P systems. In *IEEE International Conference on Cluster Computing (Cluster)*, 2009.

[22] W. Tang, Z. Lan, N. Desai, D. Buettner, and Y. Yu. Reducing fragmentation on torus-connected supercomputers. In *Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[23] W. A. Ward, C. L. Mahood, and J. E. West. Scheduling jobs on parallel systems using a relaxed backfill strategy. In *Job Scheduling Strategies for Parallel Processing, LNCS 2357*, pages 88–102, 2002.

[24] Y. Yuan, G. Yang, Y. Wu, and 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.

[25] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving parallel job scheduling by combining gang scheduling and backfilling techniques. In *Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2000.