Fault-Aware, Utility-Based Job Scheduling on Blue Gene/P Systems

Wei Tang,* Zhiling Lan,* Narayan Desai,† Daniel Buettner‡

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

†Mathematics and Computer Science Division
‡Argonne Leadership Computing Facility
Argonne National Laboratory, Argonne, IL 60439, USA
† desai@mcs.anl.gov
‡ buettner@alcf.anl.gov

***Abstract*—Job scheduling on large-scale systems is an increasingly complicated affair, with numerous factors influencing scheduling policy. Addressing these concerns results in sophisticated scheduling policies that can be difficult to reason about. In this paper, we present a general utility-based scheduling framework to balance various scheduling requirements and priorities. It enables system owners to customize scheduling policies under different circumstances without changing the scheduling code. We also develop a fault-aware job allocation strategy for Blue Gene/P systems to address the increasing concern of system failures. We demonstrate the effectiveness of these facilities by means of event-driven simulations with real job traces collected from the production Blue Gene/P system at Argonne National Laboratory.**

I. Introduction

Job scheduling is a critical task on large-scale systems, where small differences in scheduling policies can result in poor use of substantial resources. However, while traditional metrics for scheduling such as utilization rates and average response times have long been used to assess scheduler policy performance, large centers now face an increasing set of considerations in scheduling. Examples of these considerations include avoiding system faults, minimizing power consumption during peak demand, and sharing I/O resources. At the same time, these individual considerations do not occur in a vacuum; systemwide metrics, such as utilization and average response time, as well as other characteristics such as fairness, remain important. A mechanism is needed to explicitly balance these considerations. Moreover, this balance is not universal; different systems have varied priorities that result in some considerations prevailing over others.

While many scheduling algorithms have been presented in the past, this paper is motivated by operational problems in job scheduling and aims to provide a general utility-based scheduling framework to balance various scheduling requirements and priorities. In our framework, scheduling policies are described as job-scoring functions called *utility functions* that can be changed on the fly. During each scheduling iteration, each job's score is evaluated, allowing the scheduler to take the appropriate action. Utility functions are implemented in Python code and can take job parameters, such as length, size, and waiting time, as well as other factors, into consideration. For example, administrators can use them to alter the balance between responsiveness and utilization rate on a dynamic basis.

Further, we have developed a fault-aware job allocation strategy for Blue Gene/P systems to address the increasing concern of system failures. Together, *the utility-based scheduling* and *the fault-aware job allocation* not only enable system owners to explicitly control scheduling behavior but also enable intelligent, fault-aware resource allocation to improve system performance. These facilities have been implemented in *Cobalt* [1], a resource management and scheduling system used at various supercomputing centers and laboratories.

The performance of scheduling policies is completely dependent of the system workload, so intuitive assessment of scheduling policies is frequently not reliable. To address this issue, we have developed *Qsim*, a simulator of queue behavior over time, based on real system workload inputs. Using this, we can explore the behavior of different scheduling policies and measure the resources lost to system failures. In other words, not only can basic functionality be tested, but likely behavior under system load can be predicted. Thus, by using Qsim to test utility functions and fault-aware allocations strategies, machine owners can build confidence in new scheduling and allocation policies prior to deployment.

We have evaluated our fault-aware, utility-based job scheduling with job logs collected from the 40-rack Blue Gene/P system called *Intrepid* at Argonne National Laboratory [2]. The results demonstrate that, as compared to the conventional scheduling policy FCFS (first-come first-serve) with backfilling, our utility functions can lower the average response time and slowdown significantly (by up to 55%). We have also examined how the utility functions achieve their scheduling goals individually, and we have provided instructive comparison among them that results in the deployment of utility functions for the real system. Further, we have verified that the fault-aware job allocation mitigates the performance loss caused by system failures (by up to 39%).

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 provides a short description of the Blue Gene/P system at Argonne and the Cobalt resource manager used on the system. Section 4 presents our new scheduling method. Section 5 addresses how to use Qsim to evaluate scheduling policies. Section 6 shows the experimental results. Section 7 presents a brief summary and our next steps.

II. Related Work

Utility functions are widely used in economics to calculate the relative values of comparable items. This approach has been applied more recently in scheduling research to model the value of particular jobs in batch systems for the submitting users. Lee and Snavely present precise and realistic utility functions for user-centric performance analysis of schedulers [12]. Vengerov et al. use utility functions to address the problem of dynamic scheduling of data-intensive multiprocessor jobs [24]. Chen presents a utility-based approach to schedule multimedia streams in peer-to-peer systems [4].

The Maui scheduler [16][11] uses a policy scheduling mechanism that is similar to the utility scheduling implemented in Cobalt. Cobalt's utility functions are implemented as simple python functions, whereas Maui uses a complex combination of policy knobs and explicit callouts. Also, Cobalt does not reserve resources in advance for jobs, instead performing explicit drains and backfilling when needed.

Recently, increasing attention has been paid to fault-aware scheduling in high-performance computing. In [27], Zhang et al. suggest utilizing temporal and spatial correlations among failure events for better scheduling. Oliner et al. [17] present a fault-aware job scheduling algorithm for Blue Gene/L systems by exploiting node failure probabilities. In [15], fault-aware scheduling scheme is presented for the HA-OSCAR framework. *Our work is distinguished from these studies in two aspects.* First, unlike existing studies focusing on FCFS scheduling, our work presents and evaluates utility-based scheduling, which enables system owners to customize scheduling policies based on their needs. Second, to the best of our knowledge, this is the first study on fault-aware job scheduling for Blue Gene/P systems.

In [13], we studied fault-aware rescheduling for high-performance computing. That work emphasized dynamically adjusting the placement of *active jobs* (i.e., running jobs) to avoid imminent failures discovered by on-line failure predictors. In contrast, this work aims to intelligently select and allocate *inactive jobs* (i.e., queued jobs) to execute on available nodes. An active research project in our group focuses on designing and developing effective failure diagnosis and prediction mechanisms for large-scale systems [10][9][28]. The proposed failure prediction mechanisms can be directly used by this work.

Simulation is a vital technique in both scheduling and fault tolerance research. The Maui scheduler [16] features a simulation mode for performing long-time scale analysis of scheduling policies. In [22], Tikotekar et al. present a simulation framework that evaluates different fault-tolerant mechanisms such as checkpoint/restart and process migration. Failure simulation has been used in the Cobalt project to analyze system software behavior in the presence of hardware and software faults [6]. In this paper, our simulator is closely related to the batch scheduler and has two main uses: one is to aid the devising of scheduling policies by comparing possible utility functions to those already in use; the other is to measure the impact of potential failures on the system and the effectiveness of fault-aware job allocation.

III. System Overview

Intrepid is a 557 TF, 40-rack Blue Gene/P system at Argonne. This system comprises 40,960 computing nodes with more than 160,000 cores, and associated I/O nodes, storage servers, and an I/O network. It is operated as a part of the DOE INCITE [5] program. It was ranked the third fastest overall in the June 2008 TOP500 list [19].

The IBM Blue Gene/P platform is a scalable, low-power architecture. Intrepid is currently the largest installation; however, the architecture is scalable to upwards of 80 racks, with a peak performance above 1 petaflop. In order to provide a scalable, high-performance network, Blue Gene systems use a partitioned torus network for communication. Each node is part of a unit of allocation called a midplane, which includes 512 nodes, a 3D torus to connect them, and connectivity with other midplanes. Each midplane can be used either individually or in conjunction with other midplanes that are adjacent on any of the three dimensions in the network. Partitions must be of uniform length in each dimension. Also, the wiring that connects midplanes together is also shared, further limiting system partitioning possibilities.

Cobalt is an open-source, component-based resource management suite used on a large number of Blue Gene/L and Blue Gene/P systems worldwide, including Intrepid. Cobalt comprises 12,000 lines of Python code at the current release. Cobalt components correspond to pieces of functionality in resource management systems, such as scheduling, queue management, hardware resource management, and process management. Its component architecture allows easy replacement of key software functionality. This allows Qsim, our queue simulation component, to service the queue manager component and system component interface without changing any other software components. Hence, Qsim can interface directly with an unmodified Cobalt scheduler. The Qsim architecture will be described in Section V.

Cobalt is used on Intrepid for job scheduling by using utility functions. In particular, a utility function that favors old/short jobs and attempts to avoid large job starvation (i.e., WFP3 in Table II) is currently being used for production jobs, and a utility function that provides fast turnaround for small jobs (i.e., UNICEF in Table II) is being used for development jobs on Intrepid. In Section VI-C, we will present simulation results by using these utility functions.

TABLE I
NOMENCLATURE

Symbol	Description
t_i	Estimated job running time (wall time)
q_i	Job queued (waiting) time
n_i	Number of compute nodes requested by the job
n_s	Smallest partition size in the system
S_i	Utility score

IV. SCHEDULING METHOD

User jobs are submitted via a job scheduler. For example, on Intrepid, job submission and management are handled by Cobalt. The Cobalt queue manager component maintains submitted jobs in a waiting queue. The Cobalt scheduler component is responsible for selecting queued jobs and allocating them to appropriate resources in the system. Specifically, a utility-based mechanism is used for job selection, and a fault-aware method is adopted for job allocation. Job selection is based on two values: a *utility score* and a *fallback score*. For each queued job, Cobalt calculates its utility score and fallback score according to a predefined utility function. The scheduler attempts to run the job with the highest score. If this job cannot be accommodated by any available partition, the scheduler will attempt to run the job with utility scores higher than the fallback score.

Resource allocation is based on *allocation cost*. All the candidate jobs are mapped to the available partitions one by one in a decreasing order of their utility scores. If a job cannot be accommodated by any available (idle) partition, the subsequent, lower-priority job can be considered, depending on the fallback score. For each job, if multiple partitions are available for its allocation, allocation costs on these partitions are calculated and compared. The partition with the lowest allocation cost will be chosen. If none of the jobs in the candidate list can be allocated, a backfilling is invoked to schedule other queued jobs.

Next, we will discuss the utility functions definition and the resource allocation scheme in more detail. Before that, we first summarize some nomenclature used through out the paper in Table I.

A. Utility-based Job Selection

In the job selection phase the scheduler selects jobs based on their utility scores calculated by the utility function. The utility function also returns a fallback score for each job. The fallback score is the product of the job utility score and a predefined threshold (i.e., a percentage). If the job with the highest utility score (e.g., S_h) cannot be executed, jobs whose utility scores above its fallback score are considered. For example, the threshold 0.70 means that only jobs with utility scores higher than $0.70 \cdot S_h$ will be selected for job allocation. Hence, the fallback score controls the aggressiveness of scheduling and tries to avoid the job with the highest utility score from starvation. In Cobalt, the utility function can be specified as Python functions as follows

```
def defined_utililty_func(jobinfo)
    score = [define the utility function]
    TH = [define the fallback threshold]
    return (score, score * TH)
```

The design of the utility function depends on many factors, including owner's needs and job characteristics. As a result, the job utility score can be influenced by arbitrary arguments. In our experience, the arguments can be classified into two categories: one indicates the job's *urgency*, while the other reflects the job's *importance*. Thus, we have an abstract definition of a utility function as follows

$$S_i = func(urgency, importance)$$

Here, $func$ is an arbitrary function. The function has an arbitrary set of arguments that some determines the job's urgency while others reflect the jobs importance. Job waiting time (q_i) is a job attribute directly indicating urgency. Considering the influence of job length, we usually use the ratio of job waiting time to the job wall time (q_i/t_i) to represent the urgency. This value increases as jobs wait and captures the fact that, for example, waiting an hour before running is more painful for a 10-minute job than for a 6-hour job. We usually refer to this ratio as "unitless waiting time." The arguments indicating importance can be any job attributes reflecting system owner's needs. For example, user name and project name can be used if a certain project or user is considered really important at a certain period of time. Job scale (number of computing nodes) can also be used as an argument reflecting importance, since sometimes the system owner may consider large job more important.

One example function is defined as follows,

$$S_i = (q_i/t_i)^3 \times n_i$$

The first part $(q_i/t_i)^3$ adopts a nonlinear "unitless waiting time" to represent job urgency, and the second part n_i indicates job importance regarding its scale. This function suggests that jobs get increasingly high utility scores the longer they wait, especially for shorter and larger jobs. Because of the cube operator, the weight of job urgency grows faster than that of the job importance when the waiting time is larger than the wall time. Generally speaking, this utility function attempts to achieve low average response time and avoid starvation of large jobs. The utility function is resilient to user's abuse. Since a job will be killed at wall time expiration, the user may not fail the system by requesting a much shorter wall time than needed for higher priority.

Our utility-based scheduling framework is compatible with all the existing scheduling policies since they can be described in the form of utility functions. Our design goal is not to give up existing scheduling policies but to provide more possibility and flexibility for system owners to devise scheduling policies according to their amorphous needs.

When devising utility functions, selecting arguments and appropriately combining them are challenging tasks. A feasible approach is to define a tentative function and then use a high-fidelity scheduling simulator such as Qsim to measure the impact of the scheduling policy on system workloads. The simulation results can be used to refine the function. In Section VI, real system workloads will be used to demonstrate the effectiveness of the cited example utility function and others.

B. Fault-Aware Job Allocation

Once the candidate list is determined, the scheduler moves on to map the candidate jobs to suitable resources. In order to facilitate resource allocation, *allocation cost* is used to calculate a penalty caused by placing a job on different resources. The goal is to minimize *allocation cost*. Here, *penalty* may come from two sources.

First, the Blue Gene architecture places several restrictions on node allocation. For this reason, two identically sized partitions may have very different allocation costs; one might fit will with existing partitions while another prevents still-free resources from being usable in multi-midplane partitions. Moreover, the same partition may have a different allocation cost depending on the state of existing partitions on the system. Second, any partition is subject to failure. A fault-aware scheduler can place jobs where they are least likely to fail, minimizing the likelihood of the failure of a high-priority job.

A four-step method is employed to calculate *allocation cost*.

1) Find appropriately sized partitions: First we determine the smallest partition on the system that can accommodate the job. For example, if a job requests 500 nodes, all the idle 512-node partitions are considered. If multiple candidate partitions are available, the next two steps are used to choose between them.

2) Calculate partition failure probability: Next, we estimate likelihood of failure for each candidate partition during the job execution period. Specifically, for each job i, we define $P_f(X)$ as failure probability of the partition X during the period of time between the job start and the wall time expiration. Obviously, the partition with lower value is more favorable.

3) Calculate allocation footprint: Identically sized partitions can have different impact on a partially allocated system; we refer to this as the allocation footprint. Part of this quantity is derived from a combination of location of the partition and the current allocation status of the machine. When fault prediction information is added, the predicted failure state of related resources, such as adjacent midplanes or midplanes that share wiring resources, must also be considered. To this end, we scale the footprint using fault predictor data. For example, blocking the use of a partition with 0.50 chance of failure costs less than blocking a similarly sized partition with an 0.05 chance of failure. We use $BLK(X)$ denote the footprint without considering failure chance, as described in Figure 1, and use $BLK_f(X)$ represent the footprint considering failure chance. For example, to block a partition with failure chance 0.2 contributes only 0.8 into the footprint value $BLK_f(X)$.

4) Calculate overall allocation cost: Finally, we calculate the aggregate allocation cost of placing the job on a partition X as follows

$$Allocation\_Cost(X) = \alpha BLK_f(X) + \beta P_f(X)$$

The parameters α and β, which adjust the relative weights of these allocation costs, can be set by system managers. In our experiments, we have set both parameters to 1.0. For example as shown in Figure 1, for a job requesting 512 nodes, partition A is the best choice since it is failure free and blocks least parent partitions. Further, we assume a later job needs to choose between partition E and G, which have the same failure probability (0.05) and will block same number of parent partitions. In this situation, the failure probabilities of their neighbor partitions make a difference. In this example, partition G is preferred because it blocks a partition ($\overline{GH}$) with higher probability while preserving the more reliable one ($\overline{EF}$).

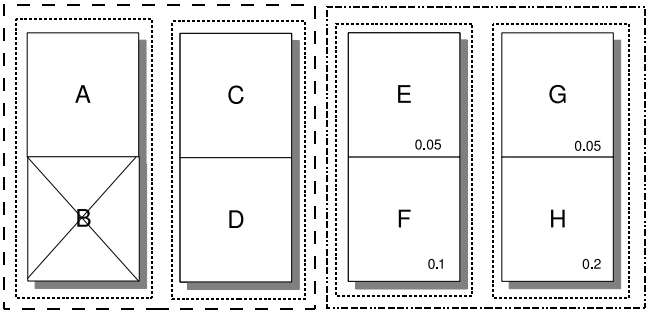

Fig. 1. Partition example in Blue Gene/P systems. Each labeled partition (A – H) contains 512 nodes. The dashed-line boxes illustrate larger partitions. Assume that the partition B is already occupied and an incoming job is requesting 512 nodes. All the partitions (A - H), except for B, are best-fit partitions for the job. Besides these 512-node partitions, larger partitions are $\overline{AB}$, $\overline{CD}$, $\overline{EF}$, $\overline{GH}$, $\overline{ABCD}$, $\overline{EFGH}$, and $\overline{ABCDEFGH}$. The occupancy of the partition B indicates that its parent partitions $\overline{AB}$, $\overline{ABCD}$, and $\overline{ABCDEFGH}$ are blocked from job allocation. $BLK(X)$ is defined as the number of available partitions blocked by placing a job on partition X. Thus, (1) $BLK(A) = 0$, since all the parent partitions of A (i.e., $\overline{AB}$, $\overline{ABCD}$, and $\overline{ABCDEFGH}$) are already blocked by the partition B, and placing the incoming job on A will not block any additional partitions, (2) $BLK(C) = BLK(D) = 1$, where the blocked partition is $\overline{CD}$, (3) $BLK(E) = BLK(F) = 2$, where the blocked partitions are $\overline{EF}$ and $\overline{EFGH}$, and (4) $BLK(G) = BLK(H) = 2$, where the blocked partitions are $\overline{GH}$ and $\overline{EFGH}$.

V. Qsim: Cobalt Simulator

Simulation is an integral part of our work for evaluating utility-based job selection and fault-aware job allocation, as well as their aggregate effect on system performance. Job execution is influenced not only by the scheduling policy used by the job scheduler but also by the users themselves. In order for a site to evaluate a new scheduling policy, it is desirable to get some idea of how the new policy will affect system performance. Historical data from a site's actual workloads

can be used to approximate what scheduling would be like under the policy being evaluated.

Motivated by this situation, we have built Qsim, an event-driven scheduling simulator for Cobalt, to evaluate the fault-aware, utility-based scheduling presented in the previous section. Using real-world workloads from Intrepid, we can examine how performance metrics such as average response time and bounded slowdown are affected by different scheduling policies. In addition, we can study the impact of hardware and software failures on system performance and how much improvement is possible by adopting fault-aware job allocation policies.

In addition to being used to compare scheduling policies, Qsim's integration with Cobalt means that changes to the production utility function need not involve untested code. That is, by virtue of having used Qsim to generate simulation data, the utility function has been evaluated on thousands of jobs. Thus, we have ample opportunity to test utility functions and correct any faults or failures in the utility function before it is deployed.

Figure 2 presents the major components of Qsim. Similar to Cobalt, Qsim comprises three components: a queue manager that maintains queued job lists, a system manager that maintains system hardware status, and a scheduler that makes scheduling decision. The scheduler in the figure is the unmodified Cobalt scheduler component. These components have the same internal interfaces as those used in Cobalt. The major difference is that Qsim reads job input from a historical workload file, whereas Cobalt gets jobs through real user commands. Moreover, Qsim can inject failures by parsing a synthetic failure log. Qsim is available as an open-source tool, included with the Cobalt release [1].

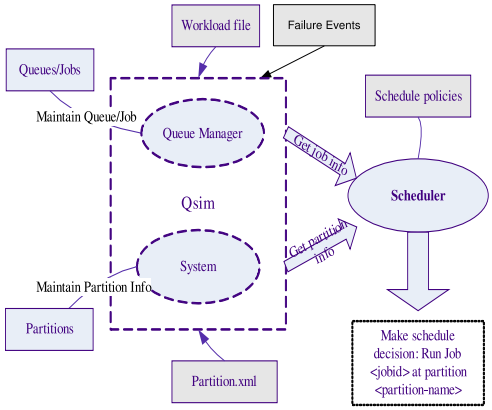

Fig. 2. Qsim components and interfaces.

Figure 3 demonstrates how Qsim works. Qsim reads job arrival times, job execution time, and system failure events as inputs. At initialization, Qsim determines a list of times where scheduling decisions may need to be made. This list initially consists of the times when new jobs arrive; however, over time, job completion times are added into this same collection.

Qsim advances the clock between the time stamps in this list. At each one, Qsim updates job and partition states based on the workload and system activity. This information is consumed by the scheduler, which determines whether any new jobs should be executed.

Upon job start, Qsim determines the completion time of the job. If Qsim is configured to simulate faults, it determines whether the job will succeed or fail. Different times stamps are inserted into the time stamp collection in these cases; succeeding jobs have their completion time inserted, while failing jobs have their failure time inserted. Failing jobs are then reinserted into the waiting queue when the failure time approaches.

Qsim logs all job events (submission, start, end, failure) into the output log, which can be used to compute the quantitative metrics such as response times and slowdown.

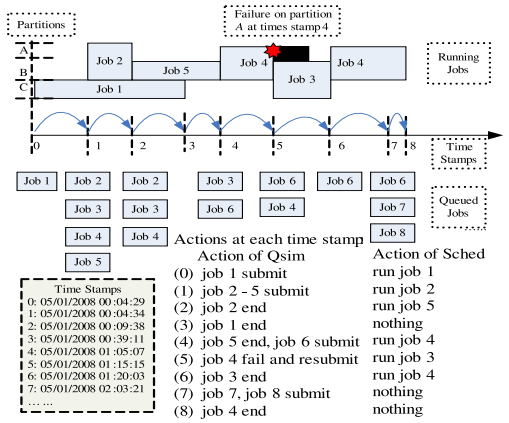

Fig. 3. Qsim simulation example.

VI. Experiments

We used Qsim to evaluate the effectiveness of the scheduling with utility functions and the fault-aware allocation. Specifically, using the real workload from 40-rack production Intrepid, we conducted simulations for a set of utility functions and examined how they influence the scheduling performance. We also evaluated the benefits of combining utility functions with fault-aware allocation.

A. Experimental Configuration

1) Utility Functions: Table II summarizes the utility functions used in the experiments. FCFS is the utility function form of first-come first-served, a commonly used scheduling policy. We propose smarter utility functions mainly based on three elements: job age (q_i), job length (t_i), and job scale (n_i). All these proposed utility functions favor older jobs and shorter jobs. Favoring old jobs is to guarantee the fairness regarding job arrival order, and favoring short jobs can achieve better average waiting/response time. The utility functions differ in their attitudes toward the job scale and the weights used to balance between the job length and the job scale. As

TABLE II
DESCRIPTION OF UTILITY FUNCTIONS

Name	Utility Function	Description
FCFS	$S_i = q_i$	first come first served
FAT	$S_i = (q_i/t_i) \cdot (n_i/n_s)^3$	favors large jobs, then old/short job
WFP1	$S_i = (q_i/t_i) \cdot n_i$	favors old/short jobs, avoiding large job starvation
WFP3	$S_i = (q_i/t_i)^3 \cdot n_i$	favors old/short jobs more, avoiding large job starvation
FCSJ	$S_i = q_i/t_i$	favors old/short jobs, regardless job scale
UNICEF	$S_i = q_i/(log_2(n_i) * t_i)$	provides fast turnaround for small jobs

shown in the table, using FAT, large jobs are favored most because a cube operator is imposed on the job scale. WFP1 and WFP3 also care about large jobs, but the weights on the job scale decrease as the cube is added on the first factor. FCSJ (i.e., first-coming short jobs) does not take the job scale into account. UNICEF, however, goes to the other extreme: it favors the small jobs. In terms of the scheduling goals, FCFS consider only the fairness regarding job arrival time; the other five not only consider the arrival fairness, but also aim to lower job average waiting time and slowdown. Meanwhile, UNICEF aims to provide fast turnaround for small jobs; FAT and WFPs attempt to avoid the starvation of large-scale jobs. Note that all of above scheduling polices are supported with the same backfilling strategy.

In our experiments, we seek to see whether smart utility functions are better than FCFS and how these utility functions influence the performance metrics, thereby achieving their scheduling goals. Besides simulating all the utility functions under the failure-free condition, we measure some of them combined with fault-aware job allocation under failure-present conditions.

2) Workload Characteristic: We used a job trace from Intrepid after its 40 racks went into full production in January 2009. The workload covers 35 days and contains 7,630 jobs, with average and median running time 4457 and 3075 seconds, respectively. The maximum job size is 32,768 nodes and the median size is 512 nodes. We use the 40-rack partition configuration in the simulation, with a minimum partition size of 64 nodes. To provide more insight into the simulation results, we classify the jobs into various categories based on the job length (running time) and job scale (number of computing nodes). Based on job length, we have four categories: Very Short, Short, Long, and Very Long. Based on job scale, we also have four categories: Very Small, Small, Large, Very Large. For simplicity, we use one-dimensional classifying for either job length or job scale. Tables III and IV summarize the categories, together with the criteria and distribution of each category. In Table IV, considering partition restrictions, we count, for example, the size of a job requesting 1000 nodes as 1024, which is the size of the smallest partition that can accommodate the job.

3) Failure Model and Prediction Mechanism: Since natural failures are hard to trigger on demand, we emulate failure events using a Weibull distribution, which produces realistic failure behaviors based on recent studies of production systems [18][21][10]. By tuning the scale parameter, failure events can be generated in a reasonable range. Before simulation begins, we generate failure event lists for all partitions. We assume the system failure is transient. That is, when the job fails, the partition is available for allocating to other queued jobs after a reboot time (20 minutes). We also assume that when a job fails, it is automatically resubmitted to the waiting queue and will run from the original beginning when it gets the chance.

TABLE III
JOB LENGTH CATEGORY, CRITERIA, AND DISTRIBUTION

Category	Very Short	Short	Long	Very Long
Time of Running	$<10min$	$10min$–$1hr$	$1hr$–$4hr$	$>4hr$
Percentage	19.9%	46.7%	29.5%	3.9%

TABLE IV
JOB SCALE CATEGORY, CRITERIA, AND DISTRIBUTION

Category	Very Small	Small	Large	Very Large
No. of Nodes	≤ 64	$[128, 512]$	$[1024, 2048]$	≥ 4096
Percentage	14.3%	54.5%	22.5%	8.7%

Much progress has been made in failure analysis and prediction. For example, hardware sensors are commonly deployed in modern computer systems for early detection of hardware errors [3][7][8], and a variety of predictive techniques have been devised to learn fault patterns for failure prediction [23][26][25][20][14]. In a previous study, we developed a dynamic meta-learning mechanism for failure prediction [9][10]. The detailed discussion of failure prediction is beyond the scope of this paper, and interested readers can refer to our previous papers for details. Failure prediction is typically described by two metrics. $sensitivity$, defined as $\frac{T_P}{T_P+F_N}$, measures the proportion of correct failure predictions to the number of actual failures. $specificity$, defined as $\frac{T_P}{T_P+F_N}$), measures the proportion of correct nonfaulty predictions to the number of actual nonfaulty cases. Here, T_P, F_P, F_N, and T_N denote the number of true positives, false negatives, false positives, and true negatives, respectively.

Specifically, we use a Weibull distribution to model failure arrivals on each 512-node partition. At each scheduling point, based on the failure events generated, Qsim calculates the failure probability $P_f(X)$ for each partition as follows. If there is a failure in the failure list on partition X before the job's expected completion time, $P_f(X) = sensitivity$. Otherwise, $P_f(X) = 1 - specificity$. Our prediction simulation scheme is similar with the one presented in [17] except for one enhancement that we consider the existence of false alarms.

B. Evaluation Metrics

In the experiment we used the following metrics, which represent either performance or reliability,

- *Average waiting time*. The job waiting time is the time period between the job's arrival time and the time of job start.
- *Average response time (RESP)*. Job response time is also called job turnaround time, representing the time period between job's arrival and successful completion. It includes waiting time and running time of the job if no failure interrupts the job.
- *Average bounded slowdown (BSD)*. 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 seconds) is imposed to avoid the value skewed by extremely small jobs. The lengths of jobs in our job trace are all more than 10 seconds, so what we referred as slowdown in later text is the same as bounded slowdown.
- *Job failure rate (JFR)*. JFR is defined as the ratio between the number of failed jobs and the total number of jobs submitted. It reflects the percentage of jobs that are interrupted by failures and is an important indicator of system's quality of service.
- *Service unit loss rate (SULR)*. SULR is defined as the ratio of wasted service units (i.e., product of job running hours and number of computing nodes) to the entire service units in a given time span. This metric directly indicates the percentage of computing cycles lost due to failures. It is an important metric for both system managers and users.

Average waiting time and response time are actually linearly related. Thus we use only waiting time under failure-free conditions since it is more illustrative when comparing the value on different job length categories. And we use only response time under failure-present condition because it covers the time loss in the case of failure.

C. Results

1) Effect of Utility Functions: We first conducted simulations with different utility functions under failure-free condition. Figure 4 illustrates the overall results. The x-axis indicates different utility functions. The y-axis shows the average performance results among all the jobs. As shown in the figure, the average waiting times range from 3560 to 7882 seconds and the average slowdowns are in the range 4.38–9.7. For both metrics, FCFS performs the worst. This means that the smart utility functions can improve system performance against FCFS. The relative performance gains on average waiting time are between 13.4% (WFP1) and 54.8% (FCSJ), with a median value of 25.7% (WFP3); the relative gains on average BSD are between 11.4% (WFP1) and 54.8% (FCSJ), with a median value 36.1% (WFP3).

Because of the skewed distribution, merely measuring the *average* values does not suffice in presenting the scheduling performance. As a supplement, Figure 5 presents more detailed

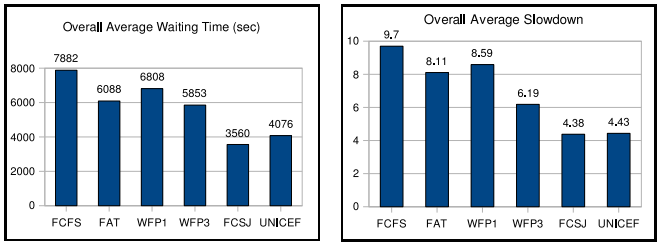

Fig. 4. Overall performance comparison of all utility functions.

distribution of the performance values. As shown in the figure, under every scheduling policy, large amount of jobs can start immediately at submission (i.e., waiting time is zero and slowdown is 1). There are also very few jobs enduring very long waiting; they either have waiting time larger than 10 hours or slowdown larger than 100. The proportion of zero-waiting jobs is nearly 50% for FCFS and around 55% for other utility functions. The long waiting jobs are less than 5% for each scheduling policy, and the FCFS also has a comparatively larger number. The distributions demonstrate in another perspective that the smart utility functions are better than FCFS in eliminating the number of long-waiting jobs and increasing the number of zero-waiting jobs.

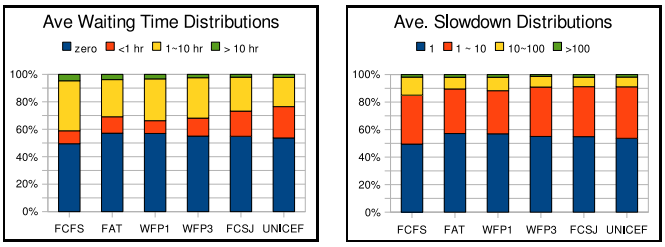

Fig. 5. Performance value distributions. The y-axis shows the percentage of the number of jobs with performance value in a certain range to the total number of jobs.

Next we explored how the utility function influences the performance. Figure 6 illustrates the performance variation by different job categories, under various scheduling policies.

The results show that smart utility functions achieve better performance than FCFS only for relatively short jobs. As shown in Figure 6 (a and b), for short and very short jobs the smart utility functions are noticeably better than FCFS in lowering the average waiting time and slowdown. For long and very long jobs, this is not the case; on the contrary, the FCFS is relatively good regarding the average waiting time of very long jobs. Thus, the overall waiting time and slowdown improvement of smart utility functions against FCFS comes from the improvement on comparatively short jobs.

The results also indicate that the average slowdown of shorter jobs is significantly larger than that of longer ones. As shown in Figure 6 (b), For very short jobs the average slowdown is between 11.5 and 30, dramatically larger than that of other categories, in 2.9–6.5, 1.9–2.3, and 1.2–1.3, re-

spectively. In other words, no matter what scheduling policy is used, comparatively short jobs dominate the average slowdown metric. Hence, achieving lower average slowdown requires providing fast turnaround for short jobs.

We noted that UNICEF achieves its scheduling goal for benefiting small jobs but does harm for very large jobs. As shown in Figure 6 (c and d), the average waiting times increase strictly (1038, 3420, 4535, and 11928) as the jobs get larger. A similar trend is shown for average slowdown. Notably, by using UNICEF, the average waiting time for very small jobs (1038 s) is remarkably less than all the others, lower by 85% than that of FCFS and by 88% than that of the worst case (WFP1). UNICEF effectively provides small jobs fast turnaround. This benefit, however, comes from the sacrifice of large jobs; indeed, for very large jobs, UNICEF performs the worst among all.

FAT achieves its preference for large-scale jobs but its overall performance is not good. As shown in Figure 6 (c and d), FAT achieves lower average waiting time and slowdown for larger jobs than smaller jobs. The average waiting times are strictly decreased (8576, 6124, 5193, and 4115) as the jobs get larger in category. The average slowdowns have a similar trend. Compared with other utility functions, FAT achieves the best performance for very large jobs. But aside from that, the overall performance of FAT is not good. Therefore, FAT is not considered as a practical utility function.

FCSJ performs better than WFP1 in most cases, except for very large jobs. This means that by avoiding starvation of large jobs, WFP1 sacrifices some interests of small jobs, even some short jobs. On the contrary, FCSJ focus on only job length, regardless of the job scale, so it can achieve overall good performance more easily. WFP3 compromises between those two; it achieves better overall performance than WFP1 and is also better than FCSJ for very large jobs. Considering that large-scale jobs are usually important jobs in our system, we tend to give them more priority. Therefore, although FCSJ performs better in most cases, we prefer WFP3 when selecting the default utility function to deploy on-site.

We conclude that utility functions are effective in achieving their scheduling goals and concerns. Specifically, smart utility functions achieve overall better performance than FCFS. FAT and UNICEF achieve their concerns on job scale. FCSJ performs the best for most job categories but is beaten by WFP1 and WFP3 for large-scale jobs. WFP3 achieves overall good performance also for large-scale jobs. Considering the importance of large-sale jobs, we deploy WFP3 as the default utility function on Intrepid. UNICEF is also considered as an alternative under some special conditions when small jobs (e.g., development jobs) should have fast respond.

2) Benefit of Fault-Aware Job Allocation: We also measured the impact of failure on systems and evaluated the effectiveness of the fault-aware job allocation. By tuning Weibull parameters, we generated a set of failure event sequences corresponding to different system failure rates. By setting the Weibull shape parameter as 1.0, and tuning the Weibull scale parameter from 1×10^6 to 6×10^6, the system-wide MTBF

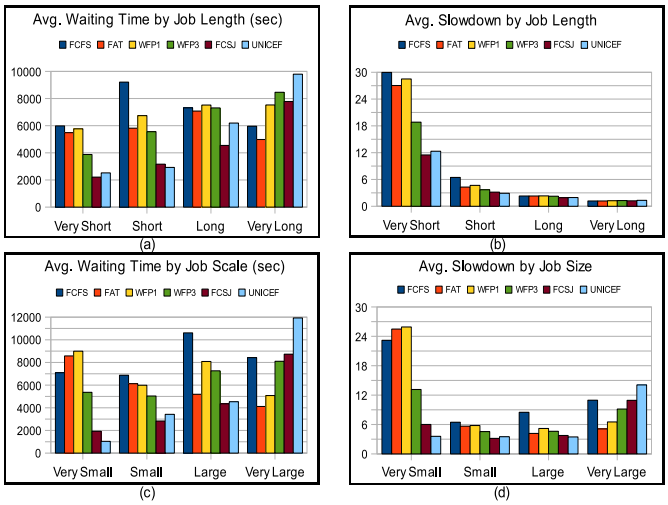

Fig. 6. Performance value by job categories. (a)(b) categorize jobs by length and (c)(d) categorize jobs by job scale, based on the criteria described in Table III and IV. For each category, the performance values of all the utility functions are compared.

is tuned in the range of 4 to 20 hours. Because the 40-rack Intrepid is new in production, we do not yet have steady-state production failure rates for the system. So, for this work, we used moderate to high failure rates for a system of this size based on several recent studies of productions systems [18][21][10]. We choose the $sensitivity$ and $specificity$ both as 0.6, a moderate value in previous study [9][10].

With each failure sequence, we run simulations with fault-aware allocation and without fault-aware allocation to evaluate their impacts. For convenience, we denote the allocation policies without fault awareness as "ordinary allocation." We first use our default scheduling utility function WFP3. Figure 7 illustrates the simulation results. In each chart, the upper lines represent the performance value of the scheduling with ordinary allocation while the lower lines indicate the case with fault-aware allocation. The performance lines fluctuate up as the failure rate gets higher, meaning that system failures degrade system performance. The generally lower "fault-aware" lines suggest that fault-aware allocation mitigates the failure impact on system performance.

Specifically, at the highest failure rate (MTBF=4 hr), the average response time of ordinary allocation goes up to 11790, degraded by 14.35% as against failure-free case; the fault-aware allocation decreases the maximum response time and the degrading percentage to 11003 and 6.72%, respectively. Average slowdown has the same trend. For ordinary allocation, the value goes up to 9.67, while the fault-aware allocation maintains it up to 8.73. Job failure rate increases up to 1.05% and 0.83% for the two allocation policies, respectively. Without fault-aware allocation, the service unit loss can be up to 14.8%, which is a dramatic figure that cannot be ignored by system owners. The fault-aware allocation decreases the highest service unit loss rate to 11.79%.

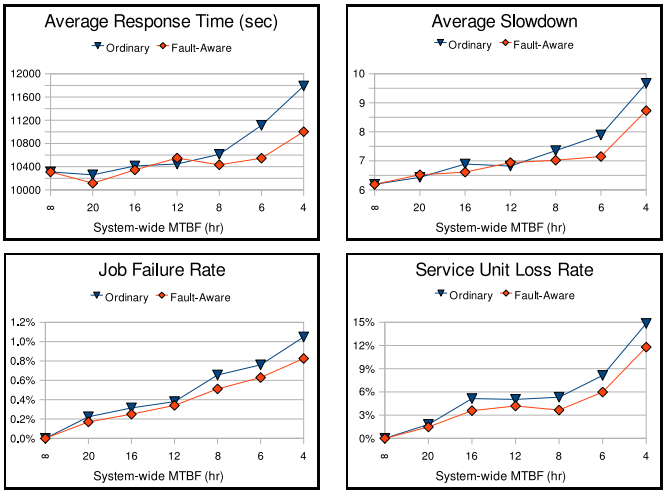

Fig. 7. Failure-present simulation using WFP3. The infinity sign ∞ indicates an ideal case – a failure-free environment. "Ordinary" means using ordinary allocation scheme without failure awareness. "Fault-Aware" means using fault-aware job allocation.

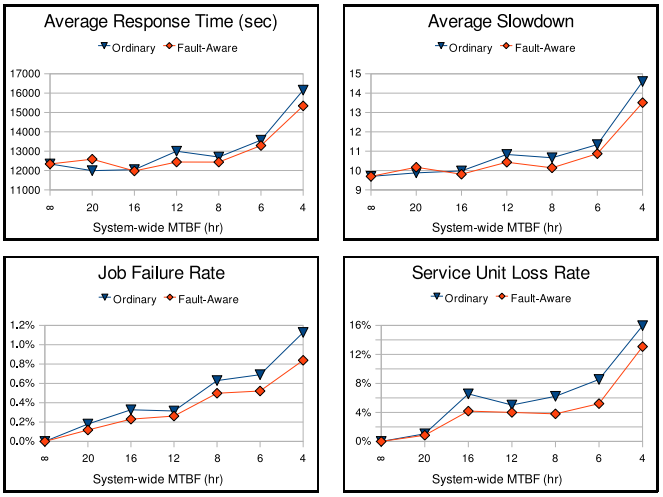

Fig. 8. Failure-present simulation using FCFS. The infinity sign ∞ indicates an ideal case – a failure-free environment. "Ordinary" means using ordinary allocation scheme without failure awareness. "Fault-Aware" means using fault-aware job allocation.

As shown in Figure 7, the trend of job failure rates is strictly increased by system failure rate, while other metrics show some volatility at the low failure rates. First, we observe that the performance degradations on response time and slowdown are not obvious when system failure rates are low, suggesting that with a few failures in the system, the average response time is not significantly impacted. On the contrary, it could be even better than the failure-free case (e.g., average response time at MTBF=20 is smaller than that at ∞) because of the uncertainty of the scheduling behavior; the failure interrupts of some long or large jobs could bring more chances for other long-waiting jobs. Second, in a few cases fault-aware allocation is not better than ordinary allocation for the performance metrics such as average response time and average slowdown, especially under low failure rate. One example is at the failure rate (MTBF=12) in the upper two charts. For reliability metrics, however, fault-aware allocation is always better than the ordinary one.

We also conducted similar experiments using FCFS, since it is a widely used job scheduling policy. Figure 8 shows the simulation results. The performance trends are similar to those using WFP3, except that the absolute values of average response time and slowdown are higher than that of WFP3. Hence, the benefit brought by smart utility functions as against FCFS applies to the failure-present condition. The job failure rate and service unit loss rate, however, do not show any gap with that using WFP3. The reason may be that the two reliability metrics are not sensitive to scheduling policies but depend mainly on system failure rate, failure location, and job allocation. Overall, we can observe that, using FCFS, the system performance degrades as the system failure rate gets higher and that the fault-aware job allocation mitigates the impact of failures.

Figure 9 presents the relative gains brought by fault-aware job allocation compared to the ordinary method, for WFP3 and FCFS respectively. As shown in the figure, the gains on average response time and slowdown are comparatively modest. For WFP3, the response time gain is up to 6.68%, and the slowdown gain is up to 9.72%. For FCFS, the values are up to 5.03% and 7.47%, respectively. The gains on JFR and SULR are more significant. For WFP3, the gains are between 17.24% and 23.77% on JFR and between 17.96% and 31.21% on SULR; FCFS also achieves relative gains of 16.66%–34.45% and 18.11%–38.94% on these two metrics.

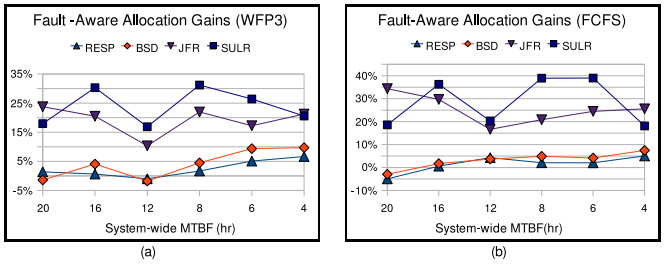

Fig. 9. Relative gains brought by fault-aware job allocation as against ordinary job allocation, using scheduling policy WFP3 and FCFS, respectively.

In summary, the ranges of relative gains suggest that fault-aware allocation can significantly improve the reliability metrics as against ordinary allocation scheme, while the performance metrics, such as average response time and slowdown, depend more heavily on utility functions. Collectively, to achieve overall good system-wide performance, we need well-designed utility functions to achieve better job scheduling performance and fault-aware allocation to achieve high reliability of jobs and to mitigate unnecessary loss of computing resource caused by system faults.

VII. Summary

In this paper, we have presented a flexible utility-based scheduling framework. More specifically, the utility-based job selection mechanism enables system owners to customize scheduling policies under different circumstances without changing the scheduler code. The fault-aware job allocation mechanism allows the scheduler to intelligently map user jobs onto suitable partitions by considering the failure possibility of system components. Furthermore, we have developed a new event-driven scheduling simulator, Qsim, which can be directly used with the Cobalt resource manager. We have demonstrated the effectiveness of the integrated facilities by comprehensive experiments with real job logs collected from the 40-rack Blue Gene/P system at Argonne National Laboratory.

While this study can directly benefit the system management of the Blue Gene/P systems, it has some limitations that remain as our future work. For example, we are designing more utility functions and measuring them on more job traces. Further study and extensive experiments will be conducted to explore the complicated relationship between utility functions, workload characteristic, and the performance metrics. Ultimately, we plan to integrate Cobalt with our previous work on failure prediction.

Acknowledgment

This work is supported in part by US National Science Foundation grants CNS-0834514, CNS-0720549, and CCF-0702737. The work at Argonne National Laboratory is supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research uses resources of the Argonne Leadership Computing Facility.

References

[1] Cobalt project web site, *http://trac.mcs.anl.gov/projects/cobalt.*

[2] Blue Gene/P red books, *http://www.redbooks.ibm.com/abstracts/sg247417.html.*

[3] B. Allen, "Monitoring Hard Disk with SMART," *Linux Journal*, 2004.

[4] F. Chen, "A Utility-based Approach to Scheduling Multimedia Streams in Peer-to-Peer Systems," *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[5] DOE INCITE project web site, *http://www.sc.doe.gov/ascr/incite*.

[6] N. Desai, E. Lusk, D. Buettner, A. Cherry, and Theron Voran, "Simulating Failures on Large-Scale Systems," *International Conference on Parallel Processing - Workshops (ICPPW'08)*, 2008.

[7] Hardware Monitoring by LM Sensors, http://secure.netroedge.com/-lm78/info.html.

[8] Health API, *http://www.renci.org*.

[9] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B. Park, "Dynamic Meta-Learning for Failure Prediction in Large-Scale Systems: A Case Study," *Proc. of International Conference on Parallel Processing (ICPP'08)*, 2008.

[10] P. Gujrati, Y. Li, Z. Lan, R. Thakur, and J. White, "A Meta-Learning Failure Predictor for Blue Gene/L Systems," *Proc. of International Conference on Parallel Processing (ICPP'07)*, 2007.

[11] D. Jackson, Q. Snell, and M. Clement, "Core Algorithms of the Maui Scheduler," *International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'01)*, 2001.

[12] C. Lee and A. Snavely, "Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers," *Proc. of IEEE Symposium on High-Performance Distributed Computing (HPDC'07)*, 2007.

[13] Y. Li, Z. Lan, P. Gujrati, and X. Sun, "Fault-Aware Runtime Strategies for High Performance Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 4, pp. 460-473, 2009.

[14] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo,"Blue Gene /L Failure Analysis and Models," *Proc. of DSN'06*, 2006.

[15] K. Limaye, C. Leangsuksun, and A. Tikotekar, "Fault Tolerance Enabled HPC Scheduling with HA-OSCAR Framework," *Proc. of the High Availability and Performance Workshop*, 2005.

[16] Maui project web site, *http://mauischeduler.sourceforge.net*.

[17] A. Oliner, R. Sahoo, J. Moreira, and M. Gupta, "Fault-aware Job Scheduling for BlueGene/L Systems," *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[18] A. Oliner and J. Stearly, "What Supercomputers Say: A Study of Five System Logs," *Proc. of DSN'07*, 2007.

[19] TOP500 Super Computing Web Site, *http://www.top500.org*.

[20] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical Event Prediction for Proactive Management in Large-scale Computer Clusters," *Proc. of International Conference on Knowledge Discovery and Data Mining*, 2003.

[21] B. Schroeder and G. A. Gibson, "A Large-scale Study of Failures in High Performance Computing Systems," *Proc. of DSN'06*, 2006.

[22] A. Tikotekar, G. Vallee, T. Naughton, S. Scott, C. Leangsuksum, "Evaluation of Fault-Tolerant Policies Using Simulation," *Proc. of IEEE Cluster'07*, 2007

[23] K. Trivedi and K. Vaidyanathan, "A Measurement-based Model for Estimation of Resource Exhaustion in Operational Software Systems," *Proc. of ISSRE'99*, 1999.

[24] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos, "Adaptive Data-Aware Utility-Based Scheduling in Resource-Constrained Systems," *Research Disclosure*, No. 513, pp. 38-39, 2007.

[25] R. Vilalta and S. Ma, "Predicting Rare Events in Temporal Domains," *Proc. of ICDM'02*, 2002.

[26] G. Weiss and H.Hirsh, "Learning to Predict Rare Events in Eevent Sequences," *Proc. of SIGKDD*, 1998.

[27] Y. Zhang, M. Squillante, A. Sivasubramaniam and R. Sahoo, "Performance Implications of Failures in Large-Scale Cluster Scheduling," *Proc. of Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'01)*, 2004.

[28] Z. Zheng, Y. Li, and Z. Lan, "Anomaly Localization in Large-Scale Clusters," *Proc. of IEEE Cluster'07*, 2007.