2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Scheduling Batch and Heterogeneous Jobs with Runtime Elasticity in a Parallel Processing Environment

Dinesh Kumar, Zon-yin Shae, Hani Jamjoom
IBM T. J. Watson Research Center
Hawthorne, NY 10591, USA
Email: {kumardi, zshae, jamjoom}@us.ibm.com

***Abstract*—Today's schedulers for a parallel processing environment are generally optimized for *submit-time elasticity* of batch jobs only, where resource needs are specified only at submission time. They are not designed for *runtime elasticity* of *heterogeneous* workloads comprising both batch and interactive jobs. By runtime elasticity it is meant that resource requirements for a job can change during its execution. This paper examines today's workload models and schedulers from this novel perspective. We show the need for an extended workload model with runtime elasticity. We then propose Delayed-LOS and Hybrid-LOS, two novel scheduling algorithms that improve and build on an existing Dynamic Programming based scheduler (LOS) designed only for batch jobs. While Delayed-LOS improves significantly over LOS, Hybrid-LOS is specifically designed for heterogeneous parallel workloads. We further propose elastic versions of these algorithms that incorporate runtime elasticity as well. Extensive simulations with GridSim framework demonstrate that Delayed-LOS & Hybrid-LOS improve average utilization by up to 4.1% & 4.55%, thereby reducing mean job-waiting time and slowdown by up to 31.88% & 25.31% and 30.3% & 24.29%, respectively.**

***Keywords*-scheduling, high performance computing, runtime elasticity, cloud computing**

I. Introduction

The cloud computing model is emerging as the *de facto* mechanism for offering computing services. Not surprisingly, this new model is being embraced to improve the consumability of High Performance Computing (HPC) services as well. In this paper, we study the impact of demand elasticity—a key ingredient in the cloud service model—on job scheduling. In particular, we investigate the limitations of today's HPC schedulers in handling demand elasticity and advocate the need for new techniques that are better suited for this emerging workload model.

From a historical perspective, cloud computing is not an entirely new concept in the HPC or parallel processing domain. Grid Computing, for instance, has attracted significant research interest over the last decade or so, much of which focused on fundamental problems in federated resource management [1]. At a high level, HPC systems have generally used a queuing model to schedule incoming jobs [2], [3], [4], [5], [6], [7], [8]. Most optimizations revolve around how an HPC system is packed and how the queue is managed to maximize system utilization while minimizing job wait times. Much of the complexity then arises when balancing a job's expected runtime needs against scheduling of future jobs.

A. Runtime Elasticity

To some extent, elasticity—in the cloud model—also operates across similar dimensions. Basically, all users are expected to get what they want, when they want it, and pay for what they use. We differentiate between two types of elasticity: *submit-time* elasticity and *runtime* elasticity. The former allows varying job execution time requirements to be specified at submission time. In contrast, runtime elasticity gives its users the ability to change their execution time requirements on-the-fly. Today's cloud resource demand model allows for both types of elasticity, whereas general HPC schedulers implement submit-time elasticity only. The challenge, then, is how can HPC schedulers best manage the underlying resources under the complete demand model, similar to what is being offered by mainstream clouds.[1]

Part of the difficulty is due to the aggressive system utilization levels that HPC systems target. It is not uncommon for an HPC system to exceed 80% utilization. In contrast, mainstream data centers often run at 15% utilization. Especially with the use of virtualization, cloud data centers have significant spare capacity to provide runtime elasticity. We believe that even in the absence of both, abundant spare capacity and virtualization, HPC schedulers can provide a certain degree of runtime elasticity. A key dimension of runtime elasticity is the time dimension. Here, a job can explicitly change its execution time requirements after it starts (e.g., modify the kill-by time). A straightforward implementation of time elasticity can negatively impact scheduling of future jobs. Therefore, a careful balance is required between running jobs needing more time and future jobs waiting in queue when their execution time requirements can change on-the-fly.

B. Heterogeneous Workloads

Unpredictable wait times have been long recognized as a key issue in batch schedulers. For certain workloads, this unpredictability can be tolerated. For other workloads, like real-time workloads, better guarantees are a must. Especially

[1]Its important to note that while HPC systems make extensive use of job priority to regulate demand, it is, however, less emphasized in the cloud model. In its place, pricing plays a fundamental role in managing demand. This is depicted, for example, by Amazon EC2's three different pricing schemes: reserved, on-demand, and spot.

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.10

in the context of cloud, next-generation schedulers must support a wide array of wait-time needs. Inspired by mainstream cloud systems, we envision HPC schedulers to support *heterogeneous* workloads. Heterogeneous workloads are *defined* as those comprising *both* batch and dedicated/interactive (reserved-capacity) jobs. By dedicated/interactive jobs it is meant that instead of the jobs being scheduled by the scheduler at some optimal time (like for batch jobs), users can specify an exact start-time of the job and the job must be triggered at that start-time. Thus, unlike batch jobs, dedicated jobs are rigid in their start-times. Many scenarios in a parallel processing environment can be envisaged where some users need to run background simulation programs that are not time or deadline critical, where as some users may require rigid and fixed time slots such as for real-time traffic data processing during certain periods of the day/week, real-time geographical, satellite or sensor data processing during certain periods of the month/year. In this case, a single HPC scheduler must be capable of efficiently scheduling this mix of batch and dedicated jobs. Present day HPC schedulers are designed for handling only batch jobs and are incapable of efficiently handling such heterogeneous workloads through a systematic and optimal methodology.

C. Summary and Contributions

In this paper, we introduce two new schedulers called Delayed-LOS and Hybrid-LOS that improve and build on the *Lookahead Optimizing Scheduler* (LOS) [7]. LOS was designed to handle only batch jobs. We also introduce elastic versions of our algorithms, which optimize job placement for execution time elasticity.

The main contributions of this paper are *threefold*:

1) We present a unified approach for designing schedulers that are optimized for heterogeneous workloads and incorporate runtime elasticity at the same time. To the best of our knowledge, this unified approach has not been taken before.
2) Delayed-LOS – an improved version of LOS is proposed for greater efficiency in scheduling of batch jobs. Delayed-LOS outperforms LOS and another popular scheduler, Easy Backfill (EASY), in most scenarios.
3) Hybrid-LOS – an extension of Delayed-LOS is proposed to handle heterogeneous workloads comprising both batch and dedicated/interactive jobs. Hybrid-LOS also outperforms its LOS and EASY counterparts for heterogeneous workload, in most scenarios.

We conduct extensive simulations to compare the performance of our algorithms against EASY and LOS. Delayed-LOS & Hybrid-LOS improve average utilization by up to 4.1% & 4.55%, thereby reducing mean job-waiting time and slowdown by up to 31.88% & 25.31% and 30.3% & 24.29%, respectively. Note that in a parallel processing environment, improved utilization of the order of even 4% can lead to huge energy savings.

This paper is organized as follows. Section II provides description of related work on today's schedulers. Section III introduces our new algorithms. Section IV presents the simulation framework and a new *Cloud Workload Format* to support runtime elasticity for heterogeneous workloads in an HPC environment. A detailed performance evaluation study through simulations is presented in Section V. In Section VI we provide directions for future work and conclude this paper.

II. Problem Description and Related Work

Scheduling of jobs in a parallel processing environment is a well studied problem and an important aspect of the HPC domain [2], [3], [4], [5], [6], [7], [8]. The efficiency of a parallel processing computing system depends on how tightly batch jobs can be scheduled so as to maximize system utilization in order to achieve energy savings. In addition to batch jobs we also consider dedicated jobs whose requested start times are fixed and are not decided by the scheduler. Therefore, mixing of batch and dedicated jobs leads to additional complexity and scheduling of flexible batch jobs around rigid dedicated jobs becomes non-trivial. Adding the runtime elasticity feature where jobs can expand and contract in their execution time leads to further complexity as regards to capability and efficiency of a scheduling algorithm to accommodate this feature.

A. Motivation

Currently, all scheduling algorithms support *submit-time elasticity* for batch jobs only. Once batch jobs with user estimated execution times are submitted, they can not be explicitly altered at runtime. Today's algorithms account for both scheduled termination (kill-by time), and premature termination before the user estimated end time, but do not account for the inter-play of explicit, on-the-fly extensions or reductions in execution time, between batch and dedicated jobs. Our new algorithms are specifically designed to address these short-comings of todays' schedulers.

B. State of the Art

The *shortest-job-first* [3] algorithm sorts, in increasing order, the waiting jobs in the queue by their estimated job runtime. This algorithm must precisely estimate jobs' execution times, either through repeated executions of jobs [4] or through compile-time analysis [9]. Majumdart *et al.* [10] investigated *smallest-job-first* scheduling, where they found that performance is poor because jobs that require few resources do not necessarily terminate quickly and cause large fragmentation in resources. Li *et al.* [11] investigated *largest-job-first* scheduling. This approach is motivated by results in bin packing optimization where a simple first-fit algorithm achieves better packing if the packed items are sorted in decreasing size [12]. It may be expected to cause less fragmentation than smallest-job first scheduling. However, large jobs do not necessarily require long execution times. Studies [5], [13] indicate that both previously mentioned scheduling mechanisms do not necessarily perform better than a straightforward FCFS scheduling.

Backfill scheduling is a recent variation of FCFS. Jobs that overrun their estimated execution times are killed. Mu'alem *et al.* [6] proposed an *Easy Backfill* algorithm to improve the resource fragmentation by aggressively moving the small jobs ahead to fill in the holes in the schedule, provided that they do not delay the first job in the queue. However, the results indicated no visible improvement when compared to the conservative Backfill algorithm, where small jobs move ahead only if they do not delay any job in the queue. The actual performance depends on the characteristics of workload. It also demonstrated that Backfill scheduling works better when a job's execution time is over-estimated by two times. Skovira *et al.* [14] have shown that Backfill scheduling is an effective means of improving resource utilization.

The Backfill algorithm only considers a single job at a time and, thus, might miss better packing opportunities. Shmueli *et al.* [7] proposed *Lookahead Optimizing Scheduler* (LOS) that uses dynamic programming to find the best multi-job combination for filling the schedule. Trace based evaluation exhibited that LOS indeed improves utilization over Easy Backfill. However, as packing is in general NP-complete, this raises concerns regarding the runtime complexity of the algorithm. By limiting the lookahead to 50 jobs, the authors demonstrated that computation time can be reduced without significant reduction in packing efficiency.

Krevat *et al.* [8] investigate the effectiveness of migration combined with FCFS and Backfill in the IBM BlueGene/L system. BlueGene/L is a massively parallel cellular architecture system with a toroidal interconnect, which typically requires jobs to be both rectangular and contiguous. These restrictions introduce fragmentation and affect the resource utilization. Migration moves jobs around the toroidal machine, performing on-the-fly de-fragmentation to create larger contiguous free spaces for waiting jobs. The paper investigated the scheduling algorithm using simulations and showed that it can improve BlueGene's resource utilization. The benefits of combining migration and *Gang* scheduling have also been demonstrated [15]. However, these studies did not include the performance impact by migration due to application preemption overhead.

Finally, Netto *et al.* [16] introduced a service level agreement (SLA) concept into scheduling through advance resource reservation. The advance reservation is not strictly observed but can be flexible and adaptive and is a trade off between quality of service and resource utilization.

III. New Scheduling Algorithms

We looked at LOS [7] as a starting point for our new algorithms. The dynamic programming approach of LOS seemed adaptable for handling cloud workloads. LOS is shown to perform better than EASY for the real system traces (CTC, SDSC and KTH logs) used in [7]. We have implemented LOS and EASY in our simulation framework and performed experiments to validate that LOS indeed performs *better* than EASY for the real system traces. Implementation and simulator details are presented later in Section IV. Plotting the results from our experiments with the SDSC log, Figure 1 shows performance comparison between EASY and LOS algorithms. Clearly LOS outperforms EASY in terms of the

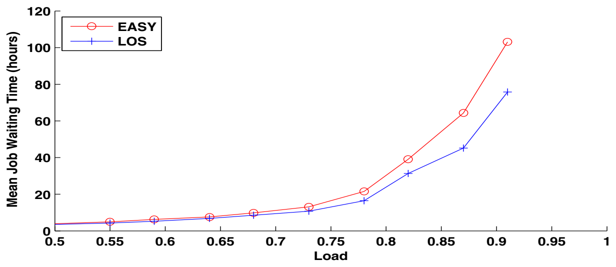

Fig. 1. Performance comparison between EASY and LOS for the SDSC log [7], using our simulation framework

mean job waiting time. On x-axis of Figure 1 the offered *Load* by a real system trace or log is calculated by multiplying the job's sizes by their runtimes, summing these values, and then dividing the result by the log's duration and the size of the machine in terms of number of processors.

Anomaly in LOS: Authors in [7] have performed experiments with variation in *load* only by multiplying the *arrival time* of each job by a constant factor (Section 4.1, [7]). We have taken the same approach for our results in Figure 1. They state that varying the load by changing the job sizes would affect their packing properties and do not consider that direction.

Our Claim: We claim here that changing job sizes for varying the load has a *significant* impact on the performance of LOS scheduler. Through experiments it is shown later in Section V that LOS performs *worse* than EASY when job sizes are varied. Our experiments make use of synthetic workloads that are generated using well derived and established analytical models of [17]. Job size related parameters in these analytical models are varied for generating the variable job size workloads.

Based on the above discussion, we propose a modified version of the LOS algorithm that performs better than EASY not just with real system logs, but with model generated synthetic workloads [17] as well. We name this improved version as Delayed-LOS. Like EASY and LOS, Delayed-LOS schedules homogeneous workload comprising batch jobs only. We then propose the Hybrid-LOS algorithm which schedules heterogeneous workload comprising both batch and dedicated (interactive) jobs.

A. Delayed-LOS: Improving LOS for Batch Jobs

The design of Delayed-LOS is motivated as follows. Let us first take a closer look at LOS which uses dynamic programming [7]. LOS examines all jobs in the waiting queue and tries to find a combination of jobs that together maximize utilization. It takes a greedy approach that achieves local optimum, but not necessarily global optimum. A globally optimal algorithm that uses off-line, linear or quadratic programming may run into scalability issues with large number of jobs or when anticipating future arrival of jobs. Moreover, it is hard to accurately predict future arrivals and an off-line algorithm can not be used for runtime elastic workload.

Authors in [7] develop LOS in two stages. They first propose a basic algorithm (Algorithm 1 in [7]) and then add reservations to the basic algorithm to avoid starvation of large sized jobs (Algorithm 3 in [7]). For our convenience, we name the corresponding dynamic programs as *Basic_DP* and *Reservation_DP*. In *Basic_DP*, the waiting queue of batch jobs is processed to find a set of jobs that will maximize current system utilization. This is better than Backfilling approach in which the queue is serially scanned to schedule *any* job whose size is less than or equal to the current free capacity. The reader is referred to Section 3.1 in [7] for a detailed discussion on this improvement using an example. Authors in [7] then observe that if there are large number of small jobs waiting behind a large job at the head of the queue, the small jobs can be repeatedly picked to maximize utilization, if the available capacity is less than the size of the large job. Thus, the large job at head of queue could be skipped repeatedly. So, instead of finding the right combination of jobs that maximize utilization at a given time, they propose to start the job at head of queue *right away* if enough capacity is available. This bounds the waiting time of the large job at head of queue. If enough capacity is not available then a reservation is made for this head job in future by considering the remaining or residual execution time of running jobs, and then the queue is scanned to find the right set of jobs to fill in holes before the reservation time. This modified version of *Basic_DP* is the *Reservation_DP*.

We *claim* that starting the job at head of queue right away is too *aggressive* an approach for bounding the head job's waiting time. Consider our example illustrated in Figure 2 which is analogous to the example in Figures 1 and 2 in [7]. There are a total of 10 processors in the machine. Suppose the waiting queue is empty and no job is running on the machine, i.e., total free capacity is 10 processors. Then jobs of size 7, 4 and 6 processors (with any duration) arrive in that order. Therefore, job at head of queue is of size 7 followed by job of size 4 and then 6. In this case starting the head job right away (Alternative-(a)) will clearly be a bad choice. It would lead to utilization of only 7 instead of 10 which could be achieved by selecting the rear two jobs (Alternative-(b)).

We therefore propose Delayed-LOS, a modified version of LOS, as follows. Delayed-LOS is presented in Algorithm 1 where it first calls *Basic_DP* (line 7). We introduce a new attribute, *skip count*, denoted by $scount$, that represents the number of times the head job in waiting queue is skipped while selecting jobs using *Basic_DP*. $scount$ is initially set to zero for a new head job and is incremented by one at every scheduling cycle if the head job is not selected in that cycle using *Basic_DP* (lines $6-11$). When $scount$ surpasses a pre-determined *maximum skip count* threshold denoted by C_s, Delayed-LOS switches to the *Reservation_DP* (lines $3-5; 12-20$). This design will lead to a utilization of 10 in our Figure 2 example and still allow bounding the waiting time of head job when $scount$ exceeds C_s. We shall provide guidelines on choosing the value of C_s through experimentation in Section V. Formulating a systematic or analytical methodology to compute the optimal value of C_s using any characteristics of the workload is a non-trivial problem and lies outside the scope of this paper. It can be studied as a separate research problem in itself since it involves multiple workload characteristics such as job arrival time, job arrival rate, job size, job execution time, etc. The notation used in Section V is presented in the 'Notations' box which also details some definitions used later in this paper. The reader is strictly advised to read the 'Notations' box before proceeding further.

B. Hybrid-LOS for Heterogeneous Workload

Here, we consider scheduling of batch jobs in presence of dedicated or interactive jobs that are required to be scheduled at requested start time as described earlier. For this, we introduce an additional queue of waiting dedicated jobs. While batch jobs are selected to be scheduled with the objective of maximizing utilization, dedicated jobs must be scheduled at their requested start time. For this, we make explicit reservations for dedicated jobs in future and schedule the batch jobs around them similar to the approach in Delayed-LOS. Hybrid-LOS is thus an extension of Delayed-LOS for heterogeneous workload and is presented in Algorithm 2. If the dedicated queue is empty (line 3) then batch jobs are scheduled as per the Delayed-LOS algorithm (line 4). Otherwise, if the first dedicated job's requested start time has reached, it is moved to the head of batch queue to be scheduled in the next scheduling cycle (lines $6-7$). If the first dedicated job's requested start time has not yet reached, then 'freeze end time' and 'freeze end capacity' defined in 'Notations' box are computed (lines $8-15$) for scheduling batch jobs around the dedicated jobs with explicit reservations for the dedicated jobs. For a given requested start time of the first dedicated job in queue, in future, if there is enough capacity for all other dedicated jobs with identical start times (lines $16-17$), lines $18-22$ allow scheduling of batch jobs around these dedicated jobs. Since there is enough capacity for all dedicated jobs, they will be scheduled *on time* at their requested start times. If enough capacity is not available, lines $24-30$ allow scheduling of batch jobs around the dedicated jobs, but some dedicated jobs will be scheduled *with delay* as regards to their requested start times. This delay is unavoidable due to insufficient capacity available for dedicated jobs. Lines $35-37$ take care of the situation when the first batch job's $scount$ parameter surpasses the C_s threshold and lines $39-42$ again take care of the dedicated queue when batch queue is empty.

C. Scheduling with Runtime Elasticity

As such, Delayed-LOS and Hybrid-LOS process only non-elastic workload in which duration of jobs do not dynamically change at runtime. For elastic workload we introduce the concept of *Elastic Control Commands* (ECCs), detailed later in Section IV-C. ECCs are commands issued by the user for extending or reducing the user-estimated execution time that was originally specified at submission time. ECCs can be issued both for jobs in execution and those that are still queued. These commands are explicitly issued by the user and are different from the implicit kill-by time computed from the user-estimated execution time. In fact, ECCs result in changes to kill-by time and thus the actual job execution time. This will then change the residual or remaining execution times of jobs

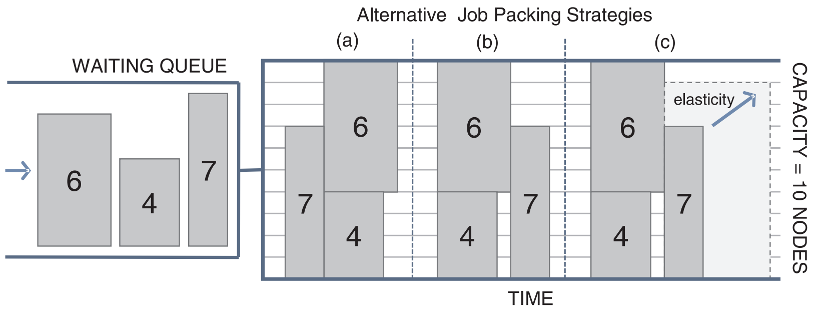

Fig. 2. Example for motivating Delayed-LOS

Notations:

- M : Total number of compute nodes (processors) that are available on an HPC system such as IBM's BlueGene/P.
- m : Total number of free or unreserved nodes available at time t; $M - m$ are reserved at time t.
- $\mathcal{W}^b$: Queue of all waiting batch jobs. $\mathcal{W}^b = \{w_1^b, w_2^b, \ldots, w_B^b\}$, $B = |\mathcal{W}^b|$. Each w_i^b is represented by the tuple $w_i^b = (num, dur, arr, scount)_i^b$, where num denotes the size or number of nodes required as part of this batch job, dur denotes duration or user-estimated execution time of the job, arr denotes arrival time of job and $scount$ denotes the skip count, i.e., number of times or scheduler cycles the job was skipped and did not get scheduled. C_s denotes an upper limit on $scount$.
- $\mathcal{W}^d$: List of all waiting dedicated (interactive) jobs. $\mathcal{W}^d = \{w_1^d, w_2^d, \ldots, w_D^d\}$, $D = |\mathcal{W}^d|$. Each w_i^d is represented by the tuple $w_i^d = (num, dur, start)_i^d$, where num denotes number of nodes required as part of this dedicated job, dur denotes duration of job and $start$ denotes the user requested start time of the dedicated job.
- $\mathcal{A}$: Sorted list of all active/running jobs including both batch and dedicated jobs. $\mathcal{A} = \{a_1, a_2, \ldots, a_A\}$, $A = |\mathcal{A}|$. Each a_i is represented by the tuple $a_i = (num, res)_i$, where num denotes number of nodes on which this active job is running and res denotes residual or remaining execution time of the active job.
- $\mathcal{S}$: Set of all jobs selected to be scheduled at time t computed after the Basic_DP is called.
- $\mathcal{S}_f$: Set of all jobs selected to be scheduled at time t computed after the Reservation_DP is called. Reservation_DP implements *shadow* times [7] or what we call *freeze* durations to avoid starvation of large jobs [7].
- $fret_b$ and $fret_d$ denote 'shadow time' or what we call 'freeze end time' for batch and dedicated jobs, respectively. $frec_b$ and $frec_d$ denote 'shadow free capacity' [7] or what we call 'freeze end capacity' for batch and dedicated jobs, respectively. $frenum$ denotes number of nodes required at freeze end time ('shadow size' in [7]) for the batch jobs present in $\mathcal{W}^b$.

Constraints: The invariant constraints are,

- $num \le M$, $start \ge t + 1$.
- $\mathcal{W}^b$ is maintained as FIFO queue, i.e., $w_1^b.arr \le w_2^b.arr \le \ldots \le w_B^b.arr$.
- $\mathcal{W}^d$ is maintained as sorted list in increasing instants of start time of dedicated jobs, i.e., $w_1^d.start \le w_2^d.start \le \ldots \le w_D^d.start$.
- $\mathcal{A}$ is maintained as sorted list in increasing order of residual duration $a_i.res$, i.e., $a_1.res \le a_2.res \le \ldots \le a_A.res$.

Input/Output: The input to Delayed-LOS and Hybrid-LOS algorithms are $\{M, \mathcal{W}^b\}$ and $\{M, \mathcal{W}^b, \mathcal{W}^d\}$, respectively. $\mathcal{W}^b$ and $\mathcal{W}^d$ are updated in real-time with new arriving jobs. The output of the algorithms are the sets $\mathcal{S}$ and $\mathcal{S}_f$ which translate into an update of $\mathcal{A}$.

Algorithm 1 $Delayed - LOS$: Delayed_LOS_Batch_Scheduler(m)

```
1:  $m \leftarrow M - \sum_{i=1}^{A} a_i.num$;
2:  if ($m > 0$) and ($\mathcal{W}^b \neq \phi$) then
3:      if ($w_1^b.num \leq m$) and ($w_1^b.scount \geq C_s$) then
4:          Remove $w_1^b$ from $\mathcal{W}^b$ and add to $\mathcal{A}$;
5:          Activate $w_1^b$;
6:      else if ($w_1^b.num \leq m$) and ($w_1^b.scount < C_s$) then
7:          Call Basic_DP;
8:          Compute $\mathcal{S}$;
9:          if ($w_1^b \notin \mathcal{S}$) then $w_1^b.scount++$;
10:         Remove $\mathcal{S}$ from $\mathcal{W}^b$ and add to $\mathcal{A}$;
11:         Activate $\mathcal{S}$;
12:     else if ($w_1^b.num > m$) then
13:         Compute $s$ such that: $m + \sum_{i=1}^{s-1} a_i.num \ < \ w_1^b.num \ \leq \ m + \sum_{i=1}^{s} a_i.num$;
14:         $fret_b \leftarrow t + a_s.res$;
15:         $frec_b \leftarrow m + \sum_{i=1}^{s} a_i.num - w_1^b.num$;
16:         $\forall w_i^b \in \mathcal{W}^b$ such that $w_i^b.num \leq m$: $w_i^b.frenum \leftarrow (t + w_i^b.dur < fret_b)\ ?\ 0\ :\ w_i^b.num$;
17:         Call Reservation_DP($frec_b$);
18:         Compute $\mathcal{S}_f$;
19:         Remove $\mathcal{S}_f$ from $\mathcal{W}^b$ and add to $\mathcal{A}$;
20:         Activate $\mathcal{S}_f$;
21:     end if
22: end if
```

as well. ECCs can be issued for both batch and dedicated jobs that were previously submitted. A maximum count on number of ECCs can be imposed for a given job.

For processing a workload that is injected with ECCs, we introduce an additional 'elastic control queue'. ECCs from this queue are processed on a first-come first-served (FCFS) basis by an *ECC processor* (Figure 3). Since EASY, LOS, Delayed-LOS and Hybrid-LOS, all four consider the residual execution times in their design, processing of ECCs would impact their performance. We shall explore this performance impact later in Section V. For processing the ECC workload, we append the Delayed-LOS and Hybrid-LOS algorithms with the ECC processor (Figure 3) and name the new algorithms as Delayed-LOS-E and Hybrid-LOS-E.

Since both Delayed-LOS and Hybrid-LOS use the *Basic_DP* and *Reservation_DP* at their core, their time and space complexities are identical to that of LOS (see Section 3.4.1 in [7]).

IV. Simulation Framework

The algorithms presented in Section III were implemented within a simulation framework written in Java programming language. The entire simulation framework comprises three different and complementary simulation packages: GridSim, ALEA 2 and CWF workload generator. These are described here and Figure 3 shows an architecture of the entire simulation framework. The entities shaded in grey belong to one of the three packages and the non-shaded entities were developed by us.

A. GridSim

GridSim is a grid simulation toolkit for resource modeling and application scheduling for parallel and distributed computing [18]. It allows modeling and simulation of an HPC environment with parallel processors. Entities such as system-users, applications, resources (processors, memory, disks, etc.), and resource brokers (schedulers) can be modeled and simulated for design and evaluation of scheduling algorithms. It provides a comprehensive framework for creating different classes of resources on which compute and data intensive applications (jobs) can be scheduled for studying the performance of new scheduling algorithms. We used GridSim to simulate IBM's BlueGene/P system on which nodes are clustered in groups of 32 processors each. In other words, only integer multiples of 32 processors can be assigned to jobs [19]. The system was simulated to have a total of 320 processors. GridSim is depicted in Figure 3 by the block on the right. GridSim does not include an event-based simulation framework and a workload generator; both are required separately.

B. ALEA 2

ALEA (version 2) [20] is a job scheduling simulator that builds on the latest GridSim 5.0 simulation toolkit. It provides an event-based simulation framework with queues and an experiment controller. The framework allows for implementation of new scheduling algorithms and their performance evaluation. The two blocks in center of Figure 3 depict ALEA. ALEA includes a *JobLoader* class that can read jobs or *gridlets* from a text file specifying the job characteristics, but does not include a workload generator.

Algorithm 2 $Hybrid - LOS$: Hybrid_LOS_Scheduler

```
1: $m \leftarrow M - \sum_{i=1}^{A} a_i.num$;
2: if $(m > 0)$ and $(\mathcal{W}^b \neq \phi)$ then
3:     if $(\mathcal{W}^d = \phi)$ then
4:         Call Delayed_LOS_Batch_Scheduler($m$);
5:     else if $(\mathcal{W}^d \neq \phi)$ and $(w_1^b.scount < C_s)$ then
6:         if $(w_1^d.start \leq t)$ then
7:             Call Move_Dedicated_Head_To_Batch_Head;
8:         else if $(t < w_1^d.start)$ then
9:             $fret_d \leftarrow w_1^d.start$;
10:            if $(w_1^d.start \leq t + a_A.res)$ then
11:                Compute $s$ such that: $t + a_{s-1}.res \ < \ w_1^d.start \ \leq \ t + a_s.res$;
12:                $frec_d \leftarrow M - \sum_{i=s}^{A} a_i.num$;
13:            else
14:                $frec_d \leftarrow M$;
15:            end if
16:            $tot\_start\_num \leftarrow \sum_{i|w_i^d.start=w_1^d.start} w_i^d.num$;
17:            if $(tot\_start\_num \leq frec_d)$ then
18:                $frec_d \leftarrow frec_d - tot\_start\_num$;
19:                $\forall w_i^b \in \mathcal{W}^b$ such that $w_i^b.num \leq m$: $w_i^b.frenum \leftarrow (t + w_i^b.dur < fret_d)\ ?\ 0\ :\ w_i^b.num$;
20:                Call Reservation_DP($frec_d$);
21:                Compute $\mathcal{S}_f$;
22:                if $(w_1^b \notin \mathcal{S}_f)$ then $w_1^b.scount + +$;
23:            else
24:                Compute $s$ such that: $m + \sum_{i=1}^{s-1} a_i.num \ < \ tot\_start\_num \ \leq \ m + \sum_{i=1}^{s} a_i.num$;
25:                $fret_d \leftarrow t + a_s.res$;
26:                $frec_d \leftarrow m + \sum_{i=1}^{s} a_i.num - tot\_start\_num$;
27:                $\forall w_i^b \in \mathcal{W}^b$ such that $w_i^b.num \leq m$: $w_i^b.frenum \leftarrow (t + w_i^b.dur < fret_d)\ ?\ 0\ :\ w_i^b.num$;
28:                Call Reservation_DP($frec_d$);
29:                Compute $\mathcal{S}_f$;
30:                if $(w_1^b \notin \mathcal{S}_f)$ then $w_1^b.scount + +$;
31:            end if
32:            Remove $\mathcal{S}_f$ from $\mathcal{W}^b$ and add to $\mathcal{A}$;
33:            Activate $\mathcal{S}_f$;
34:        end if
35:    else if $(\mathcal{W}^d \neq \phi)$ and $(w_1^b.scount \geq C_s)$ then
36:        Remove $w_1^b$ from $\mathcal{W}^b$ and add to $\mathcal{A}$;
37:        Activate $w_1^b$;
38:    end if
39: else if $(\mathcal{W}^d \neq \phi)$ then
40:    if $(w_1^d.start \leq t)$ then
41:        Call Move_Dedicated_Head_To_Batch_Head;
42:    end if
43: end if
44: Call Hybrid_LOS_Scheduler at next event;
```

Algorithm 3 Move_Dedicated_Head_To_Batch_Head

```
if $(w_1^d \neq null)$ then
    $w^b \leftarrow new()$;
    $w^b.num \leftarrow w_1^d.num$;
    $w^b.dur \leftarrow w_1^d.dur$;
    $w^b.arr \leftarrow w_1^d.arr$;
    $w^b.scount \leftarrow C_s$;
    Remove $w_1^d$ from $\mathcal{W}^d$ and add $w^b$ to head of $\mathcal{W}^b$;
end if
```

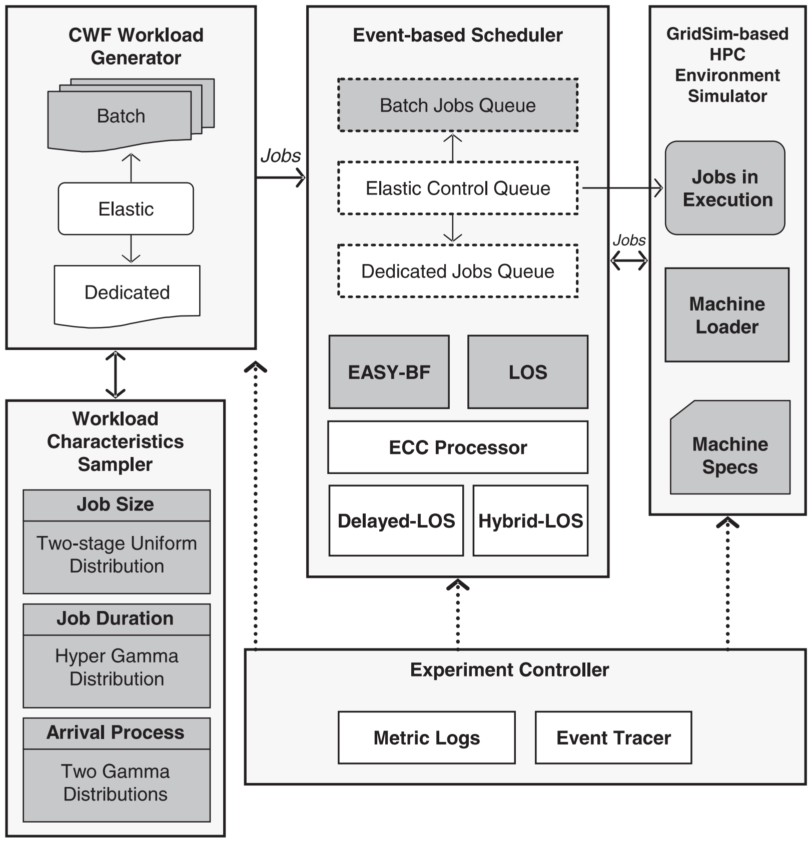

Fig. 3. Simulation framework

C. Cloud Workload Format (CWF)

To create an effective *cloud* workload generator, we had to modify the Standard Workload Format (SWF) [21] to support runtime elasticity. The SWF is a simple way of representing all properties of a workload in a single text file using numerical values. Each job is represented by a single line in the file and the entire file can comprehensively represent any given workload for a parallel machine. SWF provides a generic framework to ease the use of real-system workload logs and mathematical model generated workload traces, in evaluating performance of job schedulers in HPC systems. In its current form, it does not support runtime elasticity nor it easily supports heterogeneous requests (i.e., mixing of batch and dedicated jobs) due to absence of a 'requested start time' field for dedicated jobs. We thus propose a new format, *Cloud Workload Format* (CWF), as a natural extension to SWF. Figure 4 presents the relevant fields in SWF and our proposed extensions (Fields 19-21). These new fields constitute the *Elastic Control Commands* (ECCs) introduced earlier in Section III-C. ECCs allow a previously submitted job to be explicitly extended or reduced in its execution time requirements. Whether a batch or dedicated job is running or queued, an ECC is equivalent to the job requesting changes to its own execution time requirements. This is different from the job pre-maturely terminating before its kill-by time.

D. CWF Workload Generator

The CWF workload generator produces a synthetic sequence of jobs to be executed in a parallel processing environment. The sequence of jobs are represented in the Cloud Workload Format. The CWF workload generator extends an SWF workload generator which is available at the Parallel Workloads Archive [22]. The SWF workload generator is based on the analytical workload models for parallel computer systems developed by Lublin *et al.* [17]. Intuitively, the size, duration and inter-arrival times of jobs could have been modeled with Pareto, exponential and exponential distributions, respectively. However, Lublin *et al.* have analyzed real system workload logs at parallel computing clusters around the world and derived thorough analytical models that better fit the actual workload logs. Their models are based on a detailed statistical study of distribution fitting and involve inherent characteristics of real workloads such as degree of parallelism, job runtime model, correlation between parallelism and runtime, arrival process and workload consistency. The Kolmogorov-Smirnov test [17] is employed to assess the goodness of fit of a particular distribution. Since the approach of modeling real system workloads in [17] is very rigorous, we consider the analytical workload models of [17] as a fairly good representation of real workloads. Obviously, the purpose here is to be able to modify the model parameters and generate synthetic workloads that well represent the reality. This is

#	Name	Description
1	Job ID	Job ID counter
2	Submit Time	Arrival time of job into the system
8	Requested Number of Processors	Number of processors required to run this job.
9	Requested Time	User estimate of job runtime. In SWF, it is fixed. In our HEWF, it can change as described below.
19	Requested Start Time	Requested start time for dedicated or interactive jobs. It is set to -1 for batch jobs.
20	Request Type	Denotes whether its a usual job submission or an extension/reduction request for a previously submitted job (with the same ID). Possible values of this field are: **S**: for usual job submission **ET**: for time extensions **EP**: for processors extension **RT**: for time reduction **RP**: for processors reduction
21	Extension/ Reduction Amount	Denotes the extension/reduction amount for Field #20: **S**: -1, since the field is irrelevant **ET**: number of seconds to be added to Requested Time **EP**: number of processors to be added to Requested Number of Processors. **RT**: number of seconds to reduce Requested Time by **RP**: number by which Requested Number of Processors must be reduced.

Fig. 4. Cloud Workload Format: the shaded rows belong to SWF

much less time-consuming an exercise than measuring real system workloads in a parallel data center over a long period of time. We discuss here the various attributes of jobs in their workload model and provide values of various parameters that were chosen for our experiments:

- **Size of a job/gridlet** is sampled from a two-stage uniform distribution. The two uniform distributions are for sampling the size of *small* and *large* sized jobs. Size of small jobs is obtained by uniformly sampling from the interval $[1, 3]$ and multiplying the rounded sampled value by 32. Thus, all small sized jobs are of size either 32, 64 or 96 processors. Similarly, size of large jobs is obtained by uniformly sampling from the interval $[4, 10]$. Thus, the size of large jobs is either $128, 160, \dots,$ or 320 processors. This way the packing properties of our synthetically generated workload are different from those of the SDSC log used in [7]. The first distribution is chosen with probability P_S and the second with probability $1 - P_S$, respectively. In other words, size of a job in the synthetically generated workload is small with probability P_S and large with probability $1 - P_S$. The value of P_S is varied across most of the experiments in Section V to vary the packing properties of the synthetic workload and study its impact on the performance of various algorithms.
- **Runtime or duration of a job** is sampled from a bimodal hyper-Gamma distribution with parameters α_1 and β_1 for the first Gamma distribution, α_2 and β_2 for the second and p which determines the probability of sampling either of the Gamma distributions [17]. The parameter p is correlated with the job size s through a linear relationship, $p = p_a \cdot s + p_b$ [17]. Thus, *runtimes of jobs are correlated with their size*. Consequently, we do not vary job runtime parameters throughout our experiments and Table I shows the fixed values that we use.
- **Arrival process of jobs** is modeled with two separate Gamma distributions and an additional parameter $ARAR$ (Arrive Rush-to-All Ratio) [17]. The first Gamma with parameters α_{arr} and β_{arr} represents the inter-arrival time for jobs arriving with in a 1-hour interval. They are independent of the hour the jobs arrived in. The second Gamma with parameters α_{num} and β_{num} represents the number of jobs that arrive in each interval. Table II lists values of all these parameters that we choose for our experiments. The value of β_{arr} is varied across experiments to obtain variation in the *average load* on a machine which is defined as,

Parameter	Value
α_1	4.2
β_1	0.94
α_2	312
β_2	0.03
p_a	-0.0054
p_b	0.78

TABLE I
PARAMETER VALUES FOR JOB RUNTIMES

Parameter	Value
α_{arr}	13.2303
β_{arr}	Varies in $[0.4101, 0.6101]$
α_{num}	15.1737
β_{num}	0.9631
$ARAR$	1.0225

TABLE II
PARAMETER VALUES FOR ARRIVAL PROCESS

$$Load \quad = \quad \frac{\lambda}{M} \sum_{i=1}^{N_J} \frac{w_i.num}{\mu_i}.$$

Here, λ is the inverse of total duration of an experiment, M is the total number of processors on an HPC system, N_J is the total number of jobs scheduled during an experiment, $w_i.num$ is the number of requested processors for job i, and μ_i is the inverse of duration (runtime) of job i. By varying β_{arr}, we effectively vary λ. We additionally introduce,

$$\bar{\mu} = \frac{1}{N_J} \sum_{i=1}^{N_J} \mu_i \ ,$$

and

$$\bar{n} = \frac{1}{N_J} \sum_{i=1}^{N_J} w_i.num \ ,$$

where, $\bar{\mu}$ denotes average job runtime and $\bar{n}$ denotes average job size over a given experiment.

The above discussion on various parameters applies to both batch and dedicated jobs in a heterogeneous workload. Further, a job is chosen to be a dedicated one with probability P_D and a batch one with probability $1 - P_D$ in a synthetically generated heterogeneous workload. The value of P_D is varied across some experiments in Section V to vary the proportion of batch and dedicated jobs and study its impact on the performance of various algorithms.

As for the presence of ECCs with in a synthetic workload, the ET commands are injected with a job runtime extension probability, P_E, and RT commands are injected with a job runtime reduction probability, P_R. For brevity we keep these values fixed at $P_E = 0.2$ and $P_R = 0.1$ for all our experiments. The 'Requested Start Time' for dedicated jobs and 'Extension/Reduction Amount' for job runtime are sampled from a Poisson (exponential) distribution.

V. Performance Comparison of Algorithms

Performance of EASY, LOS and Delayed-LOS can be compared directly as they all process only batch jobs. However, performance of Hybrid-LOS can not be compared directly with EASY and LOS algorithms since the former is for heterogeneous workload. For this, we append the EASY and LOS algorithms with the dedicated job queue and name these new algorithms as EASY-D and LOS-D. EASY-D and LOS-D schedule batch jobs around the rigid dedicated jobs whose start times are fixed or modifiable through ECCs. Performance of Hybrid-LOS can then be directly compared with EASY-D and LOS-D.

For performance comparison of Delayed-LOS-E with EASY and LOS we append the latter two with the ECC processor and name them as EASY-E and LOS-E. Performance of Hybrid-LOS-E can be compared with their counterparts by appending EASY-D and LOS-D with the ECC processor as well. This leads to two new algorithms that are termed as EASY-DE and LOS-DE whose performance can be directly compared with Hybrid-LOS-E.

Table III summarizes the scope of all algorithms defined above. The various performance metrics that we consider in our simulation study are, *mean values* of HPC system utilization, job waiting time and slowdown. Slowdown is defined as the fraction,

$$\frac{avg.waitingtime + avg.runtime}{avg.runtime}.$$

While the plots depict only system utilization and job waiting time metrics, slowdown metric data is only listed in Tables IV-VII for the sake of brevity. Each point on the plotted lines in all figures presented hereafter corresponds to a single simulation run with a total of $N_J = 500$ jobs over a simulated BlueGene/P parallel processing environment with $M = 320$ processors. We also ran simulations for a couple of scenarios with $10,000$ jobs and found no significant difference in performance metrics from the 500 job runs.

Algorithm	**Workload Scheduling**	**ECC Processor**
EASY	Batch	No
EASY-D	Heterogeneous	No
EASY-E	Batch	Yes
EASY-DE	Heterogeneous	Yes
LOS	Batch	No
LOS-D	Heterogeneous	No
LOS-E	Batch	Yes
LOS-DE	Heterogeneous	Yes
Delayed-LOS	Batch	No
Hybrid-LOS	Heterogeneous	No
Delayed-LOS-E	Batch	Yes
Hybrid-LOS-E	Heterogeneous	Yes

TABLE III
List of all algorithms

A. Performance Improvement with Delayed-LOS

We discuss here the performance improvement obtained by Delayed-LOS over LOS and EASY and validate our claims made in Section III.

Validating our Claims: Recall the maximum skip count threshold C_s from Section III-A. Figure 5 shows variation of mean utilization and mean job waiting time metrics with increasing value of C_s from 1 to 20 for all three batch scheduling algorithms. Clearly, Delayed-LOS *outperforms* LOS and EASY in terms of these metrics. Moreover, for Delayed-LOS it can be seen that until about $C_s = 7$ or 8, mean values of job waiting time decrease and then become stable after a slight increase. Similarly, utilization first increases and then decreases slightly to become stable. This clearly indicates an optimal value of C_s beyond which no performance improvement can be obtained. In other words, queue head jobs could be skipped on an average 7 or 8 times to obtain better performance instead of scheduling them right away when enough capacity is available. This is a very important and useful observation and forms the key basis of Delayed-LOS. Figure 5 is for the case when $P_S = 0.5$ and $\bar{n} = 139.35$ processors. Figure 6 shows similar plots for utilization and waiting time with $P_S = 0.8$ and $\bar{n} = 89.72$ processors. Here, it is seen that when proportion of small sized jobs increases and there are fewer large sized jobs, performance improvement with Delayed-LOS is insensitive to values of C_s greater than 3. This indicates dependence of C_s on the packing properties (proportion of small and large sized jobs) of workload. This observation is intuitive and strengthens our claims made earlier. Recall again our claim made in third paragraph of Section III that changing job sizes can have a significant impact on the performance of LOS scheduler. The above discussion validates this claim. Moreover, starting a job at the head of queue right away as is done in LOS is not the most efficient strategy.

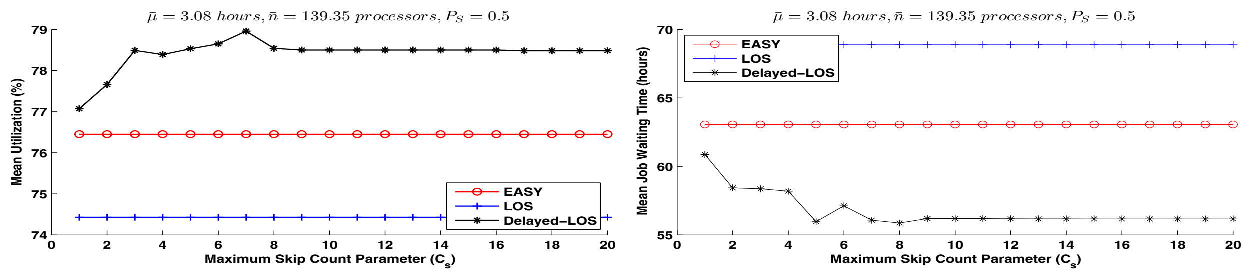

Fig. 5. Batch Workload: Variation of metrics with Maximum Skip Count Parameter (C_s) for Load=0.9, $P_S = 0.5$

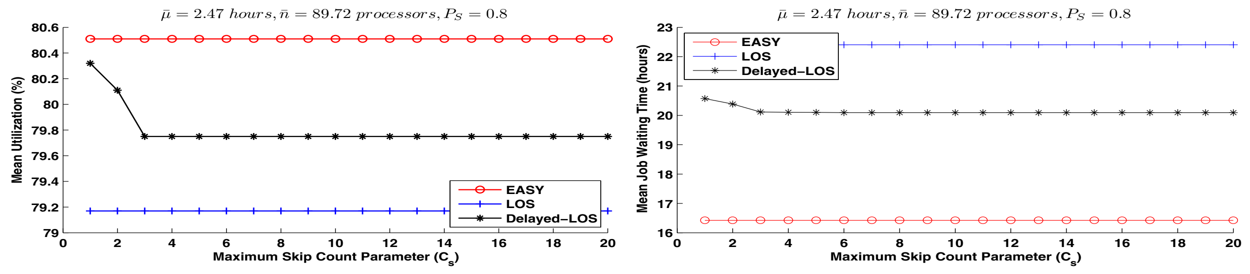

Fig. 6. Batch Workload: Variation of metrics with Maximum Skip Count Parameter (C_s) for Load=0.9, $P_S = 0.8$

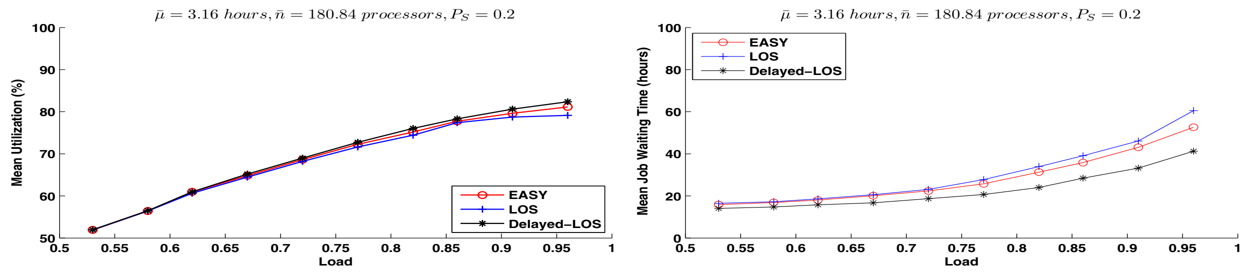

Fig. 7. Batch Workload: Variation of metrics with Load for $P_S = 0.2$

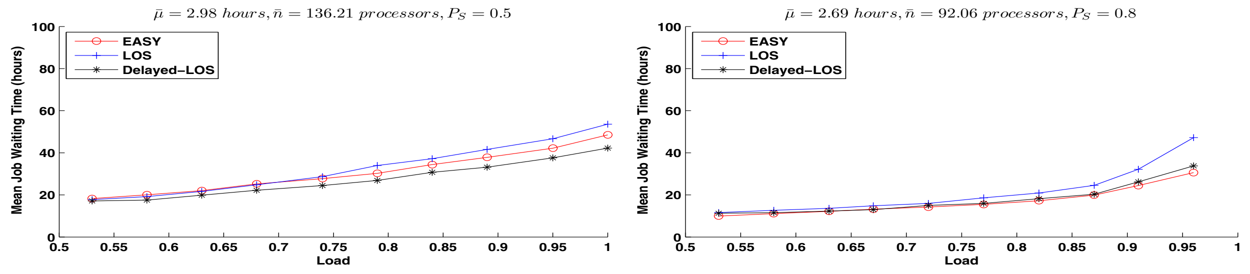

Fig. 8. Batch Workload: Variation of waiting time with Load for two cases: $P_S = 0.5$ and $P_S = 0.8$

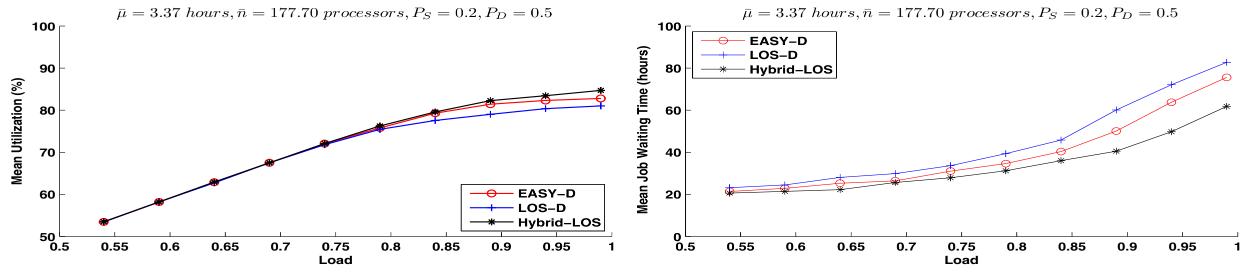

Fig. 9. Heterogeneous Workload: Variation of metrics with Load for $P_D = 0.5$ and $P_S = 0.2$

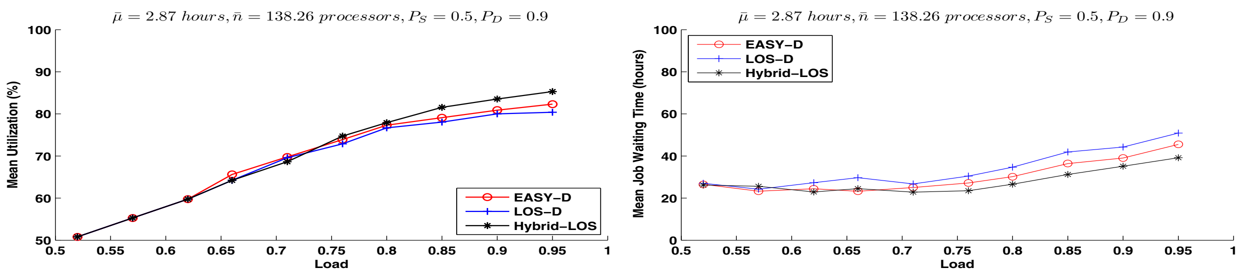

Fig. 10. Heterogeneous Workload: Variation of metrics with Load for $P_D = 0.9$ and $P_S = 0.5$

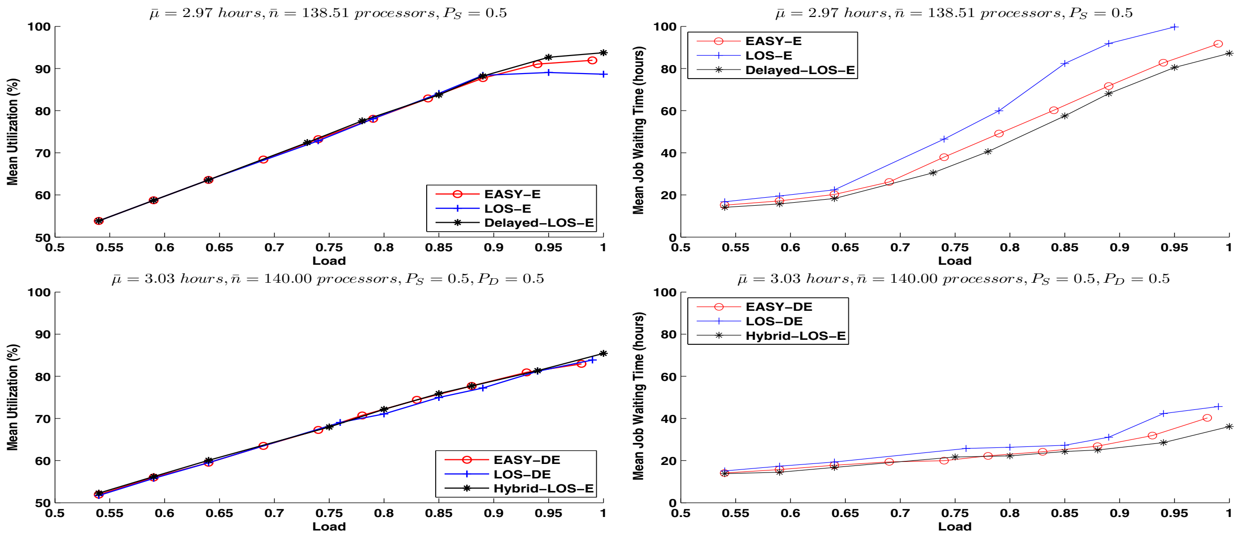

Fig. 11. Elastic Workload: Elastic Control Commands with Batch ($P_S = 0.5$) and Heterogeneous ($P_S = 0.5$ & $P_D = 0.5$)

In all the following experiments with variation in Load, we first empirically obtain the optimal value of C_s for a given value of P_S which is then used for Delayed-LOS and hence Hybrid-LOS in those experiments.

Figure 7 shows variation trends in performance metrics with increasing Load in the interval $[0.5, 1]$ for $P_S = 0.2$ and $\bar{n} = 180.84$ processors. It can be observed that for a low value of $P_S = 0.2$, i.e., lesser small sized jobs and more large sized jobs, Delayed-LOS outperforms LOS and EASY. Moreover, LOS performs *worse* than EASY and this further validates our claim made in third paragraph of Section III. Clearly when there are a lot of large sized jobs and very few small jobs, the large sized jobs will not be tightly packed and very few small jobs will be available to fill in the holes between the large sized jobs. This will result in reduced utilization of all the processors and higher waiting time for the jobs in queue.

The *maximum percentage improvement* of Delayed-LOS over LOS and EASY in terms of different metrics is summarized in Table IV. It is remarkable to note here the improvement in utilization of 4.1% and waiting time of 31.88%, which is very significant for an HPC environment in terms of the potential energy savings that can be achieved. Note that the variation in improvements of our new policies is not *stationary* in nature for varying values of load in all the plots discussed here. In simpler words, the improvements are not uniform over the entire variation in load. Therefore, though the figures plot mean values of various performance metrics for a *given* load (a single point in the plots), we have listed *maximum* percentage improvements in the various tables. Listing *mean* percentage improvements across *varying* loads will not make sense.

Figure 8 shows variation in waiting time with Load for $P_S = 0.5$ and $P_S = 0.8$. With increasing value of P_S from 0.2 in Figure 7 to 0.5 and 0.8 in this figure, we see that performance of Delayed-LOS comes closer to EASY, but it still outperforms LOS. Thus, for large number of small sized jobs and few large sized jobs, performance of Delayed-LOS is close to EASY and they both outperform LOS. This observation can lead to design of a dynamic, *algorithm selection policy* that selects the best performing algorithm among Delayed-LOS and EASY, for different proportions of small and large sized jobs in a parallel processing system.

B. *Performance of Hybrid-LOS*

For a heterogeneous workload, Figure 9 shows performance variation of different metrics with increasing Load for $P_D = 0.5$, $P_S = 0.2$ and $\bar{n} = 177.7$ processors. Again, for a low value of $P_S = 0.2$, i.e., lesser small sized jobs and more large sized jobs, Hybrid-LOS outperforms LOS-D and EASY-D.

Performance Metric	LOS (%)	EASY (%)
Utilization	4.1	1.52
Job waiting time	31.88	21.65
Slowdown	30.3	20.41

TABLE IV
MAXIMUM % IMPROVEMENT OF DELAYED-LOS OVER LOS AND EASY FOR FIGURE 7

Performance Metric	LOS-D (%)	EASY-D (%)
Utilization	4.55	2.33
Job waiting time	25.31	18.24
Slowdown	24.29	17.43

TABLE V
MAXIMUM % IMPROVEMENT OF HYBRID-LOS OVER LOS-D AND EASY-D FOR FIGURE 9

Performance Metric	LOS-E (%)	EASY-E (%)
Utilization	4.93	1.78
Job waiting time	18.94	12.19
Slowdown	18.39	11.79

TABLE VI
MAXIMUM % IMPROVEMENT OF DELAYED-LOS-E OVER LOS-E AND EASY-E FOR FIGURE 11

Performance Metric	LOS-DE (%)	EASY-DE (%)
Utilization	1.88	3.02
Job waiting time	20.76	10.18
Slowdown	19.81	14.6

TABLE VII
MAXIMUM % IMPROVEMENT OF HYBRID-LOS-E OVER LOS-DE AND EASY-DE FOR FIGURE 11

Maximum percentage improvement of Hybrid-LOS over LOS-D and EASY-D in terms of the various metrics is summarized in Table V. Improvements by 4.55% and 25.31% in utilization and job waiting time, respectively, are highly significant for an HPC environment in terms of potential energy savings. These improvements are again attributed to the same reasons as for Delayed-LOS.

Figure 10 shows mean utilization and mean job waiting time variations with Load for $P_D = 0.9$ and $P_S = 0.5$. Again, we see that performance of Hybrid-LOS outperforms LOS-D and EASY-D. Thus, for large number of dedicated jobs and few batch jobs, performance of Hybrid-LOS still outperforms both LOS-D and EASY-D. Since Hybrid-LOS is an extension of Delayed-LOS for heterogeneous workloads, it is expected that Hybrid-LOS outperforms the counterparts of LOS and EASY for heterogeneous workloads.

C. Performance Improvement with Runtime Elasticity

Finally, in presence of Elastic Control Commands (ECCs), Figure 11 shows performance improvements of Delayed-LOS-E over LOS-E and EASY-E, and Hybrid-LOS-E over LOS-DE and EASY-DE for batch and heterogeneous workloads, respectively. Tables VI and VII show these performance improvements in terms of various metrics. As compared to Tables IV and V, these figures are lower. This is because presence of elasticity at runtime can impact the packing properties of jobs which render the schedulers to be less efficient. However, Delayed-LOS-E and Hybrid-LOS-E still outperform their LOS and EASY counterparts by significant amounts for an HPC environment.

VI. CONCLUSION AND FUTURE WORK

This paper examines job scheduling in a parallel processing environment and proposes efficient algorithms to handle batch jobs and heterogeneous workloads. We argue that runtime elasticity must support two key primitives: (1) mixing and blending of batch and dedicated (reservation-based) jobs, and (2) extensions and reductions in the time dimension. We introduced Delayed-LOS and Hybrid-LOS, two novel scheduling algorithms that focus on these primitives. While Delayed-LOS improves significantly on LOS for batch workloads, Hybrid-LOS is optimized for heterogeneous HPC workloads. We have further proposed elastic versions of these algorithms that incorporate runtime elasticity.

In this paper we have focused on runtime elasticity only in the execution time of jobs. In future work we want to extend the same notion of runtime elasticity in the resource dimension i.e., size of jobs or number of processors required. Supporting resource extensions while maintaining high system utilization will require an innovative approach to the scheduling problem. To allow a running job to shrink or expand in size while maintaining space continuity—a common requirement in supercomputers like BlueGene/P—we need to incorporate the probability with which each given job can shrink or expand in size at any point in time. We are actively looking into new techniques for supporting runtime elasticity in the resource dimension.

REFERENCES

[1] Buyya, R.: High Performance Cluster Computing: Architectures and Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA (1999)

[2] Feitelson, D.: Job scheduling in multiprogrammed parallel systems. In: RC 19790 IBM. (1997)

[3] Krakowiak, S.: Priciples of Operating Systems. MIT Press (1988)

[4] Devarakonda, M.V., Iyer, R.K.: Predictability of process resource usage: a measurement based study on unix. In: IEEE Trans. Software Eng. (DEC 1989) 1579–1586

[5] Krueger, P., Lai, T.H., Dixit-Radiya, V.A.: Job acheduling is more important than processor allocation for hypercube computers. In: IEEE Trans. Parallel and Distributed Syst. (MAY 1994) 488–497

[6] Mu'alem, A., Feitelson, D.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. In: IEEE J. Parallel and Distributed Comput. (NOV 2001) 529–543

[7] Shmueli, E., Feitelson, D.: Backfilling with lookahead to optimize the packing of parallel jobs. In: IEEE J. Parallel and Distributed Comput. (SEP 2005) 1090–1107

[8] Krevat, E., Castanos, J.G., Moreira, J.E.: Job scheduling for the bluegene/l system. In: Euro-Par. (2002) 207–211

[9] Sarkar, V.: Determining avearage program execution times and their variance. In: Proc. SIGPLAN Conf. Prog. lang. Design and implementation. (JUN 1989) 298–312

[10] Majumdar, S., Eager, D., Bunt, R.: Scheduling in multiprogrammed parallel systems. In: SIGMETRICS Conf Measurement and Modeling of Comput. Syst. (May 1988) 104–113

[11] Li, K., Cheng, K.H.: Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. In: IEEE Trans. Parallel and Distributed System. (OCT 1991) 413–422

[12] Coffman, E.G., Garey, J.M.R., Johson, D.S.: Performance bounds for level-oriented two-dimensioanl packing algoritthms. In: SIAM J. Comput. (NOV 1980) 808–826

[13] Krueger, P., Lai, T.H., Radiya, V.: processor allocation vs. job scheduling on hypercube computers. In: iith Intl. Conf. Distributed Comput. Syst. (MAY 1991) 394–401

[14] Skovira, J., Chen, W., Zhou, H.: The easy-loadleveler api project. In: IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing, Volume 1162 of Lecture Notes in Computer Science, Springer-Verlag. (APR 1996) 41–47

[15] Zhang, Y., Franke, H., Moreira, J.: The impact of migration on parallel job scheduling for distributed systems. In: 6th Euro-Par. (AUG 2000) 242–251

[16] Netto, M., Bubendorfer, K., Buyya, R.: Sla-based advance reservations with flexible and adaptive time qos parameters. In: Proceedings of the International Conference on Service oriented Computing (ICSOC'07). (2007) 119–131

[17] Lublin, U., Feitelson, D.: The workload on parallel supercomputers: modeling the characteristics of grid jobs. In: IEEE J. Parallel and Distributed Comput. (NOV 2003) 1105–1122

[18] Buyya, R., Garg, S.K.: Gridsim: A grid simulation toolkit for resource modeling and application scheduling for parallel and distributed computing. In: http://www.cloudbus.org/gridsim/

[19] Carlos Sosa, B.K.: Ibm system bluegene solution: Bluegene/p application development. In: http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf

[20] Dalibor Klusáček, H.R.: Alea 2 – job scheduling simulator. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010), ICST (2010)

[21] Chapin, S.J., Cirne, W., Feitelson, D.G., Jones, J.P., Leutenegger, S.T., Schwiegelshohn, U., Smith, W., Talby, D.: Benchmarks and standards for the evaluation of parallel job schedulers. In: Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), Lect. Notes Comput. Sci. vol. 1659, Springer-Verlag. (1999) 66–89

[22] Feitelson, D.: Parallel workloads archive. In: http://www.cs.huji.ac.il/labs/parallel/workload/