Available online at www.sciencedirect.com

SCIENCE DIRECT®

J. Parallel Distrib. Comput. 65 (2005) 1090 – 1107

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Backfilling with lookahead to optimize the packing of parallel jobs[☆]

Edi Shmueli[a, b], Dror G. Feitelson[c, *]

[a]*Department of Computer Science, Haifa University, Haifa, Israel*
[b]*IBM Haifa Research Laboratory, Israel*
[c]*School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel*

Received 23 February 2004; received in revised form 14 March 2005; accepted 9 May 2005
Available online 7 July 2005

Abstract

The utilization of parallel computers depends on how jobs are packed together: if the jobs are not packed tightly, resources are lost due to fragmentation. The problem is that the goal of high utilization may conflict with goals of fairness or even progress for all jobs. The common solution is to use backfilling, which combines a reservation for the first job in the interest of progress with packing of later jobs to fill in holes and increase utilization. However, backfilling considers the queued jobs one at a time, and thus might miss better packing opportunities. We propose the use of dynamic programming to find the best packing possible given the current composition of the queue, thus maximizing the utilization on every scheduling step. Simulations of this algorithm, called lookahead optimizing scheduler (LOS), using trace files from several IBM SP parallel systems, show that LOS indeed improves utilization, and thereby reduces the mean response time and mean slowdown of all jobs. Moreover, it is actually possible to limit the lookahead depth to about 50 jobs and still achieve essentially the same results. Finally, we experimented with selecting among alternative sets of jobs that achieve the same utilization. Surprising results indicate that choosing the set at the head of the queue does not necessarily guarantee best performance. Instead, repeatedly selecting the set with the maximal overall expected slowdown boosts performance when compared to all other alternatives checked.

Keywords: Parallel job scheduling; Backfilling; Optimal packing

1. Introduction

A *parallel job* is composed of a number of concurrently executing processes, which collectively perform a certain computation. A *rigid* parallel job has a fixed number of processes (referred to as the job's *size*) which does not change during execution [7]. To execute such a parallel job, the job's processes are mapped to a set of processors using a one-to-one mapping. In a non-preemptive regime, these processors are then dedicated to running this job until it terminates. The set of processors dedicated to a certain job is called a *partition* of the machine. To increase utilization, parallel machines are typically partitioned into several non-overlapping partitions, allocated to different jobs running concurrently, a technique called *space slicing*.

To protect the machine resources and allow successful execution of jobs, users are not allowed to directly access the machine. Instead, they submit their jobs to the machine's scheduler—a software component that is responsible for monitoring and managing the machine resources. The scheduler typically maintains a queue of waiting jobs. The jobs in the queue are considered for allocation whenever the state of the machine changes. Two such changes are the submission of a new job (which changes the queue), and the termination of a running job (which frees an allocated partition). Upon such events, so-called *scheduling steps*, the scheduler examines the waiting queue and the machine resources and decides which jobs (if any) will be started at this time.

Allocating processors to jobs can be seen as packing jobs into the available space of free processors: each job takes

[☆] A preliminary version of this work has appeared in the Ninth Workshop on Job Scheduling Strategies for Parallel Processing, June 2003 [23].

[*] Corresponding author. Fax: +972 2 658 5261.

E-mail addresses: edi@il.ibm.com (E. Shmueli), feit@cs.huji.ac.il (Dror G. Feitelson).

a partition, and we try to leave as few idle processors as possible. The goal is therefore to maximize the machine utilization. The lack of knowledge regarding future jobs leads current on-line schedulers to use simple heuristics to perform the packing at each scheduling step, as described in Section 2. These heuristics do not guarantee to minimize the machine's *free capacity* which is the number of processors left unused.

We propose a new scheduling algorithm guaranteed to maximize utilization at each scheduling step. Unlike current schedulers that consider the queued jobs one at a time, our scheduler bases its scheduling decisions on the whole contents of the queue. Thus we named it LOS—an acronym for "lookahead optimizing scheduler". LOS starts by examining only the first waiting job. If it fits within the machine's free capacity it is immediately started. Otherwise, a reservation is made for this job so as to prevent the risk of starvation. The rest of the waiting queue is processed using an efficient scheduling algorithm based on dynamic-programming. The algorithm chooses a set of jobs which will maximize the machine utilization and will not violate the reservation for the first waiting job.

In some cases, it is possible to achieve the same utilization using several alternative sets of jobs. The initial algorithm respects the arrival order of the jobs, and uses the set of jobs that is closer to the head of the queue. However, we show that performance can further improve if we disregard the queue order and choose the set which contains the maximal number of jobs or the jobs with the maximal overall slowdown.

Section 3 provides a detailed description of the algorithm, and of the different alternatives when several job sets lead to the same utilization. It then presents a discussion on complexity. While the problem of packing jobs is in general NP-complete, we show that this particular instance is actually tractable using the dynamic programming pseudo-polynomial algorithm. Section 4 describes the simulation environment used in the evaluation and presents the experimental results from the simulations in which LOS was tested using trace files from real systems. It also presents and compares LOS's results when using alternative job sets.

2. Scheduling with backfilling

The first-come first-serve (FCFS) scheduling algorithm starts jobs in the same order in which they arrive in the queue. If the machine's free capacity cannot accommodate the first job, it will not attempt to start any subsequent job. It is a fair scheduling policy which guarantees freedom of starvation, since a job cannot be delayed by other jobs submitted at a later time. The problem with FCFS is the resulting poor utilization of the machine, since small jobs which could utilize idle processors are delayed until all jobs ahead of them are started.

To improve utilization and other performance metrics, the queue may be considered in some other order [12,22]. The *Shortest Job First* (SJF) algorithm sorts the waiting jobs by increasing estimated runtime and executes the jobs with the shortest runtime first. A job's runtime can be estimated through repeated executions of the job [6] or through compile-time analysis [20,3]. The opposite algorithm, *Longest Job First*, executes the jobs with the longest processing time first. The *Smallest Job First* [16] and the opposite *Largest Job First* algorithms sort the waiting jobs by increasing and decreasing size respectively. The latter is motivated by results in bin-packing that indicate that a simple first-fit algorithm achieves better packing if the packed items are sorted in decreasing size [5]. Finally, the *Smallest Cumulative Demand First* [16,21] algorithm sorts the jobs in an increasing order according to the product of their size and expected execution time, so small short jobs get the highest priority.

The problem with all the above algorithms is that jobs may suffer from starvation, and processing power is wasted if the first job cannot run. This problem is solved by *backfilling* algorithms, which allow small jobs from the back of the queue to execute before larger jobs that arrived earlier, thus utilizing the idle processors, while the latter are waiting for enough processors to be freed [15]. Backfilling is known to greatly increase user satisfaction since small jobs tend to get through faster, while bypassing large ones [11,2]. Note that backfilling algorithms require the jobs' runtimes to be known in advance. In real implementations, the users need to provide an estimate of their job's runtime, which in practice is often specified as a runtime upper-bound.

Backfilling was first implemented on a production system in the "EASY" scheduler developed by Lifka for the IBM SP1 parallel supercomputer [15], and later integrated with IBM's LoadLeveler product [24]. EASY implements an aggressive version of backfilling, in which any job can be backfilled provided it does not delay the first job in the queue. This means that starvation cannot occur since the queuing delay for the job at the head of the queue depends only on jobs that are already running, and these jobs will eventually either terminate or be terminated when they exceed their estimated runtime. The problem is that jobs other than the first may be repeatedly delayed by newly arriving jobs that skip them in the queue, which reduces predictability.

When predictability is required one can use "conservative" backfilling, which makes reservations for *all* queued jobs rather than only for the first one. In this version, backfilling is done subject to checking that it does not delay any previous job in the queue, and thus the risk of starvation is eliminated. The Maui scheduler [10] has a parameter that allows the system administrator to set the number of reservations. Mu'alem and Feitelson [17] compared EASY backfilling to conservative backfilling and show that for most cases the performance of the EASY backfilling algorithm was better than that of conservative backfilling. Further analysis showed this to be the result of complex interactions among the scheduler, the workload, and the metric used to evaluate the performance [8].

Another parameter of backfill algorithms is the order in which the queue is scanned. The Maui scheduler allows a general priority function to be defined [10]. Chiang et al. show that prioritizing jobs by estimated runtime (shortest first) or by expected slowdown (highest first) improves several performance metrics [4]. Our results corroborate these findings in the context of selecting among alternative job sets that achieve the same utilization.

Additional variants of backfilling allow the scheduler more flexibility. *Dynamic backfilling* allows the scheduler to overrule a previous reservation if introducing a slight delay will improve utilization considerably [11]. Talby and Feitelson presented *slack-based backfilling*, an enhanced backfill scheduler that supports priorities [26]. These priorities are used to assign each waiting job a slack, which determines how long it may have to wait before running: important jobs will have little slack in comparison with others. Backfilling is allowed only if the backfilled job does not delay any other job by more than that job's slack. Srinivasan et al. [25] have suggested a strategy called *selective backfilling* where reservations are provided selectively only to jobs whose expected slowdown exceeds some threshold. This is in fact equivalent to slack-based backfilling, where the slack is set to a value that will limit the slowdown to the desired threshold. Ward et al. have suggested the use of a *relaxed backfill* strategy, which is similar, except that the slack is a constant factor and does not depend on priority [27].

Lawson and Smirni presented a *multiple-queue backfilling* approach in which each job is assigned to a queue according to its expected execution time, and each queue is assigned to a disjoint partition of the parallel system on which only jobs from this queue can be executed [14]. Their simulation results indicate a performance gain compared to a single-queue backfilling, resulting from the fact that the multiple-queue policy reduces the likelihood that short jobs get delayed in the queue behind long jobs. Good results were also obtained by Chiang et al. when simulating a cluster of eight Origin 2000 machines, which effectively work like a multi-server queue [4].

One feature that all previous backfilling algorithms have in common is that they use heuristics that attempt to improve utilization and other performance metrics, but do not guarantee optimality. Our main contribution is to show that optimal utilization can in fact be achieved in this context, despite the NP-completeness of packing in general. This is due to the relatively limited repertoire of sizes provided by realistic machines, as shown in Section 3.4. However, this is still only optimal for each scheduling step; it is not optimal in the global sense as could be achieved by an off-line algorithm with knowledge of the future.

Table 1
Summary of notation

Symbol	Meaning
N	Machine size
n	Free capacity
rj_i	Running job number i
R	The set of all running jobs
wj_i	Waiting job number i
WQ	The set of all waiting jobs
S	The set of jobs selected for scheduling

3. The LOS scheduling algorithm

The LOS scheduling algorithm examines all the jobs in the queue in order to maximize the current system utilization. Instead of scanning the queue in some order, and starting any job that is small enough not to violate prior reservations, LOS tries to find a combination of jobs that together maximize utilization. This is done using dynamic programming. Note that this is still a greedy on-line algorithm, and therefore the result is a local optimum, but not necessarily a global optimum. A globally optimal schedule might choose to leave processors idle in anticipation of future arrivals.

To ease the exposition, Section 3.1 first presents the basic algorithm, showing how to find a set of jobs that together maximize utilization. Section 3.2 then extends this by showing how to also respect a reservation for the first queued job. Section 3.3 examines selection among alternative sets of jobs that achieve the same utilization value, in the interest of improving other performance metrics. Section 3.4 analyzes the complexity of the algorithm.

The notation we will use is summarized in Table 1. The machine size is *N*. At the time that the scheduler is called, denoted by *t*, the machine runs a set of jobs $R = \{rj_1, rj_2, \ldots, rj_r\}$, each with two attributes: its *size*, and its estimated remaining execution time, *rem*. For convenience, *R* is sorted by increasing *rem* values. The machine's free capacity is $n = N - \sum_{i=1}^{r} rj_i.size$. The queue contains a set of waiting jobs $WQ = \{wj_1, wj_2, \ldots, wj_q\}$, which also have two attributes: a *size* requirement and a user estimated runtime, *time*. The task of the scheduling algorithm is to select a subset $S \subseteq WQ$ of jobs, referred to as the *selected jobset*, which maximizes the machine utilization. These jobs are removed from the queue and started immediately. The selected jobset is *safe* if it does not impose a risk of starvation.

To provide an intuitive feel of the algorithms, the description includes an on-going scheduling example. Paragraphs describing the example are marked by ♣.

3.1. The basic algorithm

3.1.1. A two-dimensional matrix

Our goal is to find a set of jobs that will maximize utilization. To do so, the waiting queue, *WQ*, is processed using a dynamic-programming algorithm. Intermediate results are stored in a two dimensional matrix denoted *M* of size $(|WQ| + 1) \times (n + 1)$. Each cell $m_{i,j}$ contains an integer value *util*, and a boolean flag *selected*. *util* holds the maximal achievable utilization at this time, if the machine's free capacity is *j* and only waiting jobs $\{1 \ldots i\}$ are considered for scheduling. Note that *util* is not the machine's average utilization; rather, it is a momentary utilization value which represents the maximal number of processors that can be utilized by the considered waiting jobs. The *selected* flag, if set, indicates that wj_i was chosen for execution ($wj_i \in S$); when the algorithm finishes calculating *M*, it will be used to trace the jobs which construct *S*. For convenience, the $i = 0$ row and $j = 0$ column are initialized with zero values. Such padding eliminates the need of handling end cases.

3.1.2. Filling M

M is filled from left to right, top to bottom, as indicated in Algorithm 1. The values of each cell are calculated using values from previously calculated cells. The idea is that if adding another processor (bringing the total to *j*) allows the currently considered job wj_i to be started, we need to check whether including wj_i in the selected jobset increases the utilization. The utilization that would be achieved assuming this job is included is calculated in the variable $util'$. If this is higher than the utilization without this job, the *selected* flag is set to true for this job. If not, or if the size of job wj_i is larger than *j*, the utilization is simply what it was without this job, that is $m_{i-1,j}.util$. The computation stops when reaching cell $m_{|wq|,n}$ at which time *M* is filled with values. In particular, the last cell filled shows the maximal utilization that can be achieved at this stage, as it is based on considering all possible combinations of jobs.

A special case occurs when the utilization with wj_i turns out to be the same as without it. This may happen if two different sets of jobs, one which contains wj_i and one which does not, lead to the same utilization. We must then decide which set to select. The current algorithm ignores this dilemma; it selects the currently considered job only if it actually improves the utilization, and does not select it if it leads to the same utilization. Due to the order in which jobs are considered, this is equivalent to preferring jobs that appear closer to the head of the queue. However, other options are also possible, and we discuss them in Section 3.3.

The complexity of the basic algorithm is obviously the size of the matrix $|WQ| \times n$. This can be trimmed by first removing all jobs that are larger than *n* (the current free capacity) from consideration.

♣ Our example concerns a machine of size $N = 10$. At $t = 25$, when the scheduler is called (e.g. due to the termination of some previously running job), the machine runs a single job rj_1 with *size* = 5 and expected remaining execution time *rem* = 3. The machine's free capacity is thus $n = 5$. The set of waiting jobs and the resulting *M* is shown in Table 2. The *selected* flag is denoted by $\nwarrow$ if it is set, and by $\uparrow$ if cleared. The first job has size 7, so it does not fit in the 5 free processors. The utilization in its row is therefore 0, and the *selected* flag is false. The second job has size 3. When only 1 or 2 processors are considered, it too is too large to fit. But when 3 or more processors are considered, it is selected and the utilization is then 3. The third job has size 1. When only 1 or 2 processors are considered, it is selected and the utilization is 1. But when 3 processors are considered, it is better to select the second job and not the third one. With 4 or 5 processors, both can be selected, leading to a total utilization of 4. The fourth job is selected when two processors are considered (better than using the third job with utilization 1), or when 5 are considered (achieving a utilization of 5 together with job 2). Job 5 does not add anything and is never selected. Thus the maximal achievable utilization of the $j = 5$ free processors when considering all $i = 5$ jobs is $m_{5,5}.util = 5$. Note that a conventional backfilling algorithm, which considers jobs in the queue order, would select jobs 2 and 3 and only achieve a utilization of 4.

Algorithm 1. Constructing *M*

```
for $j = 0$ to $n$
    $m_{0,j}.util \leftarrow 0$                                  // init top row
for $i = 1$ to $|WQ|$                                 // outer loop on rows (jobs)
    $m_{i,0}.util \leftarrow 0$                                  // init first column
    for $j = 1$ to $n$                                // inner loop on columns (free processors)
        $m_{i,j}.util \leftarrow m_{i-1,j}.util$                 // default: don't use this job
        $m_{i,j}.selected \leftarrow False$
        if $wj_i.size \leqslant j$                          // job is a potential candidate
            $util' \leftarrow m_{i-1,j-wj_i.size}.util + wj_i.size$   // find achievable utilization with it
            if $util' > m_{i-1,j}.util$               // improves utilization
                $m_{i,j}.util \leftarrow util'$             // so use it
                $m_{i,j}.selected \leftarrow True$
```

Algorithm 2. Constructing S

```
$S \leftarrow \{\}$                                  // initially empty
$i \leftarrow |WQ|$                                  // start from end
$j \leftarrow n$
while $i > 0$ and $j > 0$                            // continue until reach edge
    if $m_{i,j}.selected = True$
        $S \leftarrow S \cup \{wj_i\}$               // add this job
        $j \leftarrow j - wj_i.size$                 // skip appropriate columns
    $i \leftarrow i - 1$
```

Table 2
Resulting M for the example

i (*size*)	j					
	0	1	2	3	4	5
0 (ϕ)	0	0	0	0	0	0
1 (7)	0	0 ↑	0 ↑	0 ↑	0 ↑	0 ↑
2 (3)	0	0 ↑	0 ↑	3↖	3↖	3↖
3 (1)	0	1↖	1↖	3 ↑	4↖	4↖
4 (2)	0	1 ↑	2↖	3 ↑	4 ↑	5↖
5 (2)	0	1 ↑	2 ↑	3 ↑	4 ↑	5 ↑

3.1.3. *Constructing S*

Starting at the last computed cell $m_{|wq|,n}$, S is constructed by following the boolean flags backwards as described in Algorithm 2. Each job is considered in turn. Jobs that are marked as selected are added to S. This induces a jump to a different column, that reflects the number of processors remaining after starting this job. Jobs that are not marked are simply skipped.

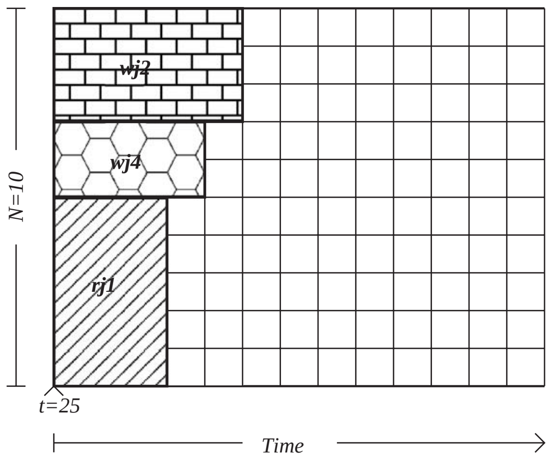
♣ The resulting S contains two jobs: wj_2 and wj_4, and its scheduling at time $t = 25$ is illustrated in Fig. 1. The list of jobs in the queue and their expected runtime is also shown.

3.2. *Adding reservations to the algorithm*

3.2.1. *Freedom from starvation*

Algorithm 1 has the drawback that it might starve large jobs. Consider the first queued job in our example. Its size is 7, so it cannot start running immediately, and other jobs are selected in its place, namely jobs wj_2 and wj_4. But after 3 time units job rj_1 will terminate, releasing its 5 processors. However, now jobs wj_2 and wj_4 are still running, so again 7 processors are not available, and again other jobs will be selected. This can continue indefinitely, if smaller jobs arrive and are backfilled.

The solution to this problem is to bound the waiting time of the first queued job. The algorithm begins by trying to start the first waiting job. If $wj_1.size \leqslant n$, it is removed from the waiting queue, added to the running jobs list and starts executing. Otherwise, the algorithm calculates the *shadow time* at which wj_1 can begin its execution [15]. It does so by traversing the list of running jobs while accumulating their sizes until reaching a job rj_s at which $wj_1.size \leqslant n + \sum_{i=1}^{s} rj_i.size$. The shadow time is defined to be $shadow = t + rj_s.rem$. A reservation is then made for job wj_1 at time *shadow* (recall that R is ordered by increasing *rem* times, so at this time



wj	*size*	*time*
1	7	4
2	3	5
3	1	6
4	2	4
5	2	2

Fig. 1. Scheduling wj_2 and wj_4 at $t = 25$.

the first *s* running jobs have terminated and freed their processors). To dismiss the concern of handling special cases, we set *shadow* to ∞ if wj_1 can be started at *t*. In this case *every* selected jobset is safe, as the first waiting job is assured to start without delay.

♣ The 7 processors requirement of wj_1 prevents it from starting at $t = 25$. It will be able to start at $t = 28$ after rj_1 terminates, thus *shadow* is set to 28 in the example.

3.2.2. Maximizing utilization

One way to ensure the safeness of the selected jobset is to require all jobs in *S* to terminate before the shadow time, so as not to interfere with the first job's reservation. But this is overly restrictive. The idea is that some processors may be left over at the shadow time after wj_1 is started. These processors, referred to as the *shadow free capacity*, can be used by backfilled jobs without interfering with the reservation for wj_1. Using them can lead to a better jobset S', still safe but with a much improved utilization.

If the first waiting job, wj_1, can only start after rj_s has terminated, than the shadow free capacity, denoted by *extra*, is calculated as follows:

$$extra = n + \sum_{i=1}^{s} rj_i.size - wj_1.size.$$

To use the extra processors, the jobs which are expected to terminate before the shadow time are distinguished from those that are expected to still run at that time, and are therefore candidates for using the extra processors. Each waiting job $wj_i \in WQ$ will now be represented by two values: its original size and its *shadow size*—its size at the shadow time. Jobs expected to terminate before the shadow time have a shadow size of 0. The shadow size is denoted by *ssize*, and is calculated using the following rule:

$$wj_i.ssize = \begin{cases} 0 & t + wj_i.time \leqslant shadow, \\ wj_i.size & \text{otherwise.} \end{cases}$$

If wj_1 can start at *t*, the shadow time is set to ∞, as noted above. As a result, the shadow size *ssize* of *all* waiting jobs is set to 0, and any computations involving extra processors are unnecessary. In this case setting *extra* to 0 improves the algorithm runtime. All these calculation are done in a preprocessing phase, before running the dynamic programming algorithm.

♣ wj_1which can begin execution at $t = 28$ leaves 3 extra processors. *extra* is therefore set to 3. As for the queued jobs, wj_5 is the only job expected to terminate before the shadow time, thus its shadow size is 0.

3.2.3. A three-dimensional data structure

To manage the use of the *extra* processors, we use a three dimensional matrix denoted M' of size $(|WQ| + 1) \times (n + 1) \times (extra + 1)$. Each cell $m'_{i,j,k}$ now contains two integer values, *util* and *sutil*, as well as the boolean *selected* flag. *util* holds the maximal achievable utilization at *t*, if the machine's free capacity is *j*, the shadow free capacity is *k*, and only waiting jobs $\{1 \ldots i\}$ are considered for scheduling. *sutil* hold the minimal number of extra processors required to achieve the *util* value mentioned above. The *selected* flag is used in the same manner as described in Section 3.1.1. Likewise, the $i = 0$ rows and $j = 0$ columns are initialized with zero values, this time for all *k* planes.

3.2.4. Filling M'

The values in every $m'_{i,j,k}$ cell are calculated in an iterative manner using values from previously calculated cells as described in Algorithm 3. The calculation is similar to Algorithm 1, except for another encompassing loop, and the use of a slightly more complicated condition that checks that enough processors are available both now *and* at the shadow time. First, we initialize the cell as if the job is not selected. Then, if the job is small enough, we check whether it will improve the utilization. The job will be selected if it actually improves the utilization, or even if the utilization stays the same but less *extra* processes are used at the *shadow* time. The computation stops when reaching cell $m'_{|wq|,n,extra}$.

If the values of *util* and *sutil* when selecting a job remain the same as without that job, then we have found two sets of jobs that have identical resource usage. For now we ignore this special case, and simply skip the current job. Other options are considered in Section 3.3.

♣ We use the notation $size_{ssize}$ to represent the size and shadow size of the jobs. When the shadow free capacity is $k = 0$, only wj_5 who's $ssize = 0$ can be started. As a result, the maximal achievable utilization of the $j = 5$ free processors, when considering all $i = 5$ jobs is $m'_{5,5,0}.util$ =2, as can be seen in the top of Table 3.

When the shadow free capacity is $k = 1$, wj_3 who's $ssize = 1$ is also considered for scheduling. As can be seen in the second part in Table 3, starting at $m'_{3,1,1}$ the maximal achievable utilization is increased to 1, at the price of using a single extra processor. When considering job wj_5 it becomes preferable when only two processors are available, at $m'_{5,2,1}$. But from the next cell on, both jobs wj_3 and wj_5 are selected, and the utilization is 3.

As the shadow free capacity increases to $k = 2$, wj_4 who's shadow size is 2, joins wj_3 and wj_5 as a valid candidate. Its effect is illustrated in the third part of Table 3. At $m'_{4,2,2}$ it becomes preferred over job wj_3 and the maximal achievable utilization increases to 2. This remains the case till the end of the row, because it uses both available extra processors. Only when the fifth job is considered can we increase the utilization further, using the fact that it does not require any extra processors. The maximal utilization is then 4, the sum of wj_4 and wj_5 sizes, using a minimum of 2 extra processors, corresponding to wj_4's shadow size.

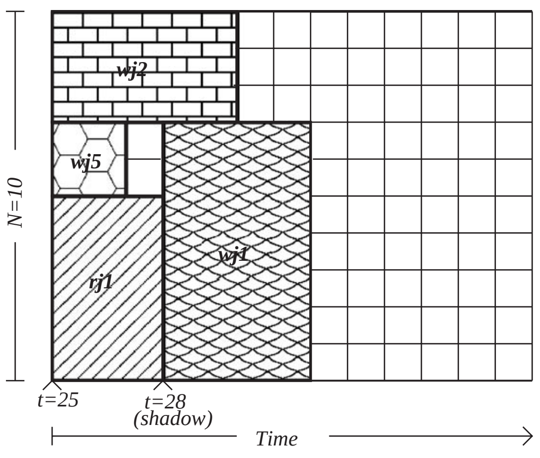
It is interesting to examine the $m'_{5,2,2}$ cell, as it introduces an interesting heuristic decision. When the machine's free capacity is $j = 2$ and jobs $\{1 \ldots 5\}$ are considered, the maximal achievable utilization can be accomplished by

Algorithm 3. Constructing M'

```
for $k = 0$ to $extra$                                              // outer loop on layers (extra processors)
    for $j = 0$ to $n$
        $m'_{0,j_k}.util \leftarrow 0$                               // init top row
        $m'_{0,j_k}.sutil \leftarrow 0$
    for $i = 1$ to $|WQ|$                                           // middle loop on rows (jobs)
        $m'_{i,0_k}.util \leftarrow 0$                               // init first column
        $m'_{i,0_k}.sutil \leftarrow 0$
        for $j = 1$ to $n$                                          // inner loop on columns (free processors)
            $m'_{i,j,k}.util \leftarrow m'_{i-1,j,k}.util$             // default: don't use this job
            $m'_{i,j,k}.sutil \leftarrow m'_{i-1,j,k}.sutil$
            $m'_{i,j,k}.selected \leftarrow False$
            if $wj_i.size \leqslant j$ and $wj_i.ssize \leqslant k$  // job is a potential candidate
                $util' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.util + wj_i.size$
                $sutil' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.sutil + wj_i.ssize$
                if $(util' > m'_{i-1,j,k}.util)$ or                // improves util or reduces shadow util
                  $(util' = m'_{i-1,j,k}.util$ and $sutil' < m'_{i-1,j,k}.sutil)$
                    $m'_{i,j,k}.util \leftarrow util'$                // so use it
                    $m'_{i,j,k}.sutil \leftarrow sutil'$
                    $m'_{i,j,k}.selected \leftarrow True$
```

Table 3
Filled M' in the example

i ($size_{ssize}$)	j					
	0	1	2	3	4	5
$k = 0$						
0 (ϕ_ϕ)	0_0	0_0	0_0	0_0	0_0	0_0
1 (7_7)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
2 (3_3)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
3 (1_1)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
4 (2_2)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
5 (2_0)	0_0	$0_0 \uparrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$
$k = 1$						
0 (ϕ_ϕ)	0_0	0_0	0_0	0_0	0_0	0_0
1 (7_7)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
2 (3_3)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
3 (1_1)	0_0	$1_1 \nwarrow$	$1_1 \nwarrow$	$1_1 \nwarrow$	$1_1 \nwarrow$	$1_1 \nwarrow$
4 (2_2)	0_0	$1_1 \uparrow$	$1_1 \uparrow$	$1_1 \uparrow$	$1_1 \uparrow$	$1_1 \uparrow$
5 (2_0)	0_0	$1_1 \uparrow$	$2_0 \nwarrow$	$3_1 \nwarrow$	$3_1 \nwarrow$	$3_1 \nwarrow$
$k = 2$						
0 (ϕ_ϕ)	0_0	0_0	0_0	0_0	0_0	0_0
1 (7_7)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
2 (3_3)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
3 (1_1)	0_0	$1_1 \nwarrow$	$1_1 \nwarrow$	$1_1 \nwarrow$	$1_1 \nwarrow$	$1_1 \nwarrow$
4 (2_2)	0_0	$1_1 \uparrow$	$2_2 \nwarrow$	$2_2 \nwarrow$	$2_2 \nwarrow$	$2_2 \nwarrow$
5 (2_0)	0_0	$1_1 \uparrow$	$2_0 \nwarrow$	$3_1 \nwarrow$	$4_2 \nwarrow$	$4_2 \nwarrow$
$k = 3$						
0 (ϕ_ϕ)	0_0	0_0	0_0	0_0	0_0	0_0
1 (7_7)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
2 (3_3)	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$3_3 \nwarrow$	$3_3 \nwarrow$	$3_3 \nwarrow$
3 (1_1)	0_0	$1_1 \nwarrow$	$1_1 \nwarrow$	$3_3 \uparrow$	$3_3 \uparrow$	$3_3 \uparrow$
4 (2_2)	0_0	$1_1 \uparrow$	$2_2 \nwarrow$	$3_3 \uparrow$	$3_3 \uparrow$	$3_3 \uparrow$
5 (2_0)	0_0	$1_1 \uparrow$	$2_0 \nwarrow$	$3_1 \nwarrow$	$4_2 \nwarrow$	$5_3 \nwarrow$



wj	$size_{ssize}$	$time$
1	7_7	4
2	3_3	5
3	1_1	6
4	2_3	4
5	2_0	2

Fig. 2. Scheduling wj_2 and wj_5 at $t = 25$.

either starting wj_4 or wj_5, both with a size of 2, yet wj_4 will use 2 extra processors while wj_5 will use none. The algorithm chooses to skip wj_4 and selects wj_5 as it leaves more extra processors to be used by other jobs.

Finally, the full $k = 3$ shadow free capacity is considered. wj_2, who's shadow size is 3 can now join the other jobs as a valid candidate. As can be seen in the bottom part of Table 3, the maximal achievable utilization at $t = 25$, when the machine's free capacity is $n = j = 5$, the shadow free capacity is $extra = k = 3$ and all five waiting jobs are considered by the algorithm is $m'_{5,5,3}.util = 5$. The minimal number of extra processors required to achieve this utilization value is $m'_{5,5,3}.sutil = 3$.

3.2.5. Constructing S'

S' is constructed in essentially the same way as S (Algorithm 2), with the necessary extension to handle the third index k. The construction starts at the last computed cell $m'_{|wq|,n,extra}$, follows the *selected* flags, and stops when reaching the 0 boundary of any plane.

♣ In our example the selected flag in $m'_{5,5,3}$ is set, so wj_5 is added to S'. Moving to cell $m'_{4,3,3}$ we find that the flag is not set, and the same applies in cell $m'_{3,3,3}$. Therefore the two jobs wj_4 and wj_3 are skipped. Landing in cell $m'_{2,3,3}$ we find a set flag, and add job wj_2 to S', thus completing the algorithm and achieving a full utilization of 5 processors. This selected jobset is safe, and ensures that wj_1 will start without a delay at time $t = 28$. The resulting jobset is illustrated in Fig. 2. Note the difference from the jobset shown in Fig. 1, which achieves the same utilization, but is not safe, as it does not respect the reservation for job wj_1.

3.3. Improving performance by job selection

3.3.1. Adding merit values

In Section 3.2.4, we noted that there are cases where several different sets of jobs lead to exactly the same *util* and *sutil* values. In these cases, we need to choose one of these sets. Rather than make an arbitrary decision, we can define additional metrics that will guide the decision. To do so, we enhanced our three dimensional data structure described in Section 3.2.3 by including an additional *merit* value in every $m'_{i,j,k}$ cell, in addition to the existing *util*, *sutil*, and *selected* fields. We also modified LOS's core algorithm for constructing M' to update and consider the merit value. The idea is that whenever the same utilization value can be achieved either by selecting or skipping job i, the modified algorithm considers the merit value in order to decide whether to set the *selected* flag or not. By doing so, the selected jobset S' is optimized in view of the merit.

Algorithm 4 describes the use of the merit for the construction of M'. As in the original algorithm, it fills the three-dimensional matrix one cell at a time. The default is not to select the current job i, and to use the *util*, *sutil*, and *merit* values from the corresponding cell in row $i - 1$. However, the current job will be selected for inclusion in S' if one of three conditions holds. The first is that this will cause the utilization to increase. The second is that the utilization will stay the same, but this will be achieved using less extra processors. The third and new condition is that both the utilization and the extra processors are the same, but the merit value improves.

It is important to note that the use of the merit does *not* change any of the utilization values in any of M's cells when compared to the values computed by the original algorithm, and that the selected set, S', will still maximize the machine utilization. Likewise, it is also important to understand that M' is still filled in a similar iterative manner as described in Section 3.2.4, thus the use of the merit does not change the complexity of the algorithm.

The calculation of the merit values is represented in Algorithm 4 by the function *calc_merit*(). We experimented with various merit functions, which are summarized in Table 4 and described in the following sub-sections.

Algorithm 4. Constructing M' With Merit

```
for $k = 0$ to *extra*                                  // outer loop on layers (extra processors)
    for $j = 0$ to $n$
        $m'_{0,j_k}.util \leftarrow 0$                  // init top row
        $m'_{0,j_k}.sutil \leftarrow 0$
        $m'_{0,j_k}.merit \leftarrow 0$
    for $i = 1$ to $|WQ|$                               // middle loop on rows (jobs)
        $m'_{i,0_k}.util \leftarrow 0$                  // init first column
        $m'_{i,0_k}.sutil \leftarrow 0$
        $m'_{i,0_k}.merit \leftarrow 0$
        for $j = 1$ to $n$                              // inner loop on columns (free processors)
            $m'_{i,j,k}.util \leftarrow m'_{i-1,j,k}.util$        // default: don't use this job
            $m'_{i,j,k}.sutil \leftarrow m'_{i-1,j,k}.sutil$
            $m'_{i,j,k}.merit \leftarrow m'_{i-1,j,k}.merit$
            $m'_{i,j,k}.selected \leftarrow False$
            if $wj_i.size \leqslant j$ and $wj_i.ssize \leqslant k$      // job is a potential candidate
                $util' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.util + wj_i.size$
                $sutil' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.sutil + wj_i.ssize$
                $merit' \leftarrow calc\_merit(m'_{i-1,j-wj_i.size,k-wj_i.ssize}.merit, wj_i)$
                if ($util' > m'_{i-1,j,k}.util$) or              // job leads to improvement
                  ($util' = m'_{i-1,j,k}.util$ and $sutil' < m'_{i-1,j,k}.sutil$) or
                  ($util' = m'_{i-1,j,k}.util$ and $sutil' = m'_{i-1,j,k}.sutil$ and $merit' > m'_{i-1,j,k}.merit$)
                    $m'_{i,j,k}.util \leftarrow util'$           // so use it
                    $m'_{i,j,k}.sutil \leftarrow sutil'$
                    $m'_{i,j,k}.merit \leftarrow merit'$
                    $m'_{i,j,k}.selected \leftarrow True$
```

3.3.2. *Always skip or always select*

The original algorithm never selects a job unless it actually improves the utilization or reduces the use of the extra processors. This can be called the "always skip" approach. It is motivated by the desire to select jobs that are closer to the head of the waiting queue, and therefore have waited longer. In the framework of the merit values, we use $-i$ (minus the job's serial number) as the merit value. In this way, jobs that have a low rank in the queue get a higher merit value.

The opposite approach is "always select", in which we prefer to include the current job and thereby tend to select jobs that are closer to the tail of the waiting queue. Surprisingly, simulation results indicated that this improves the system's performance (as measured by the mean response time and bounded slowdown metrics). This is explained by the fact that the population of the waiting queue is not homogeneous. The backfilling algorithm repeatedly picks jobs out of the queue and starts them. The jobs that are thus removed are those that use idle processors, and will not interfere with the reservation for the first job; hence these are jobs that tend to be small and short. The head of the waiting queue has been subjected to such selections for a longer time, so the characteristics of the remaining jobs are different from those that are present near the end of the queue.

Table 4
Options for defining merit

Scheme	$calc\_merit(m, wj_i)$
Always skip	$-i$
Always select	i
Max jobs	$m + 1$
Short first	$m - wj_i.time$
Max slowdown	$m + \frac{(t - wj_i.arrival) + wj_i.time}{wj_i.time}$

♣ In our example, using *always select* instead of *always skip* leads to selecting jobs wj_3 and wj_4 instead of job wj_2. This leads to the same maximal current utilization, and uses the same number of extra processors, but the jobs are closer to the tail of the queue.

3.3.3. *Maximizing the number of jobs*

An alternative approach attempts to improve the performance metrics directly. Both the response time and bounded slowdown metrics are computed as an average over all jobs executed by the system. In this average, all jobs have the same weight. Therefore, improving the performance of a larger set of jobs will boost the performance metrics more than doing so for a smaller set of jobs.

In terms of the merit value, this idea is expressed by simply counting the jobs in S'. We select the current job and add it to S' if this will lead to a larger set of selected jobs; otherwise we skip it. And indeed, simulation results indicated that this did in fact improve the performance metrics over the always select and always skip schemes.

3.3.4. *Running shortest jobs first*

It is well-known that the optimal off-line scheduling scheme with respect to the average response time metric is the shortest jobs first algorithm. An approximation of this algorithm can actually be implemented in the context of backfilling, because we are given user estimates of the running time of each job. The merit value is then calculated as the sum of these runtime estimates, with a minus sign. As a result, the set of jobs with smaller runtimes will have a higher merit value.

3.3.5. *Maximizing the total slowdown*

Another optional merit value is maximizing the overall slowdown of the selected jobs. Slowdown is the ratio of the time it takes to run the job on a loaded system divided by the time it takes on a dedicated system, i.e. $slowdown = \frac{response\ time}{running\ time}$. Since $response\ time = wait\ time + running\ time$ and the jobs' actual *running time* is unknown at the time of scheduling, we use the user-estimated runtime for that job instead. Thus for each considered job wj_i, its slowdown is computed as follows:

$$wj_i.slowdown = \frac{wait\ time + estimated\ runtime}{estimated\ runtime} = \frac{(t - wj_i.arrival) + wj_i.time}{wj_i.time}.$$

The proposed merit value is the total jobs slowdown with the purpose of choosing the set S' which *maximizes* this factor. This goes against intuition which states that performance metrics will increase less if we add smaller values, not larger values. But this short-term view is wrong. The reason is that if we choose the jobs with the minimal slowdown, we actually focus on those that have the least waiting time; this is similar to the always select scheme described above. But by selecting those jobs that have the maximal slowdown, we focus on those that are the *most sensitive* to the slowdown metric (i.e. the shortest jobs) and have also suffered the most so far. These are the jobs that are also expected to cause the most significant further degradation to the performance metrics if we leave them in the queue. It is therefore better to start them immediately, and prevent worse degradation of the metrics in the future.

In other words, maximizing slowdown is a generalization of scheduling short jobs first. It prefers short jobs, but also takes the time that they have already waited in the queue into account. Simulation results shown in Section 4.3 indicate that this is the best job selection scheme among those checked.

3.4. *Complexity of the algorithm*

The complexity of the EASY algorithm is linear in the size of the queue: $O(|WQ|)$. The complexity of LOS is much higher. However, it is still quite low in absolute terms, and it leads to an optimal packing.

3.4.1. *Complexity analysis*

The most time and space demanding task is the construction of M', which depends on three input parameter: $|WQ|$—the length of the waiting queue, *n*—the machine's free capacity at *t*, and *extra*—the shadow free capacity. $|WQ|$ depends on the system load. Since each $m'_{i,j,k}$ cell is computed in a constant time, the total running time is simply $|WQ| \times n \times extra$. Both *n* and *extra* are bounded by *N* — the size of the machine, but there is a dependence between them: the first queued job must be bigger than *n*, implying that $extra < N - n$. Their product therefore satisfies

$$n \times extra \leqslant n \times (N - n) = Nn - n^2. \tag{1}$$

This is maximized when $n = N/2$, leading to a value of $N^2/4$. Therefore, the time complexity of the algorithm for constructing M' and thus for producing the optimal schedule is

$$T = O(|WQ| \times N^2). \tag{2}$$

While this is a polynomial expression, it is important to understand that it is *not* a polynomial in *the size of the input*. The input is the list of jobs sizes. To compute the size of the input we first need to encode each of the waiting jobs' sizes in a binary format. The length of encoding an integer *x* is $\log x$. As the sizes of jobs may be as big as *N*, each requires $\log N$ input bits. Hence the size of encoding the entire input is

$$I = O(|WQ| \times \log N). \tag{3}$$

As *T* is not bounded by a polynomial in *I*, this is not a polynomial time algorithm.

What we have here is an algorithm whose running time *T* is bounded by a polynomial in *two* variables: the size of the input, and the largest input *value*. Such algorithms, often based on dynamic programming, are known as *pseudo-polynomial* algorithms (defined in [9] and reviewed in textbooks such as [18, Chapter 16]). They are designed to solve NP-complete problems using the fact that in practice it is sufficient to solve the problem for a restricted set of inputs, in contrast to the unbounded values which are considered in general theoretical analysis. Not all NP problems have such solutions: the ones that do not are called NP-complete “in the strong sense”. As we have found a pseudo-polynomial solution, our problem is not strongly NP-complete (unless $P = NP$).

In our case, *N* is a predefined constant. Moreover, for realistic systems it is even quite small, on the order of hundreds or maybe thousands of processors. This restriction allows the optimal schedule to be produced in a reasonable time, feasible for practical implementation. This is demonstrated in our experimental results in Section 4.5. We show there that during all our simulations, in which we scheduled a total of more than 100,000 jobs, the scheduler never took more than about 0.6 seconds to run, and the average was less than 2 milliseconds.

3.4.2. *Limited lookahead*

The length of the waiting queue, $|WQ|$, depends on the system load. On heavy loaded systems the mean waiting queue length can reach tens of jobs with peaks sometimes reaching hundreds—a fact that significantly increases the runtime of the algorithm.

A possible enhancement is to limit the number of jobs examined by the algorithm by including only the first C waiting jobs in WQ where C is a predefined constant. We call this approach *limited lookahead* since we limit the number of jobs the algorithm is allowed to examine. It is often possible to produce a schedule which maximizes the machine's utilization by looking only at the first C jobs, thus achieving the same result at a lower cost. But obviously this is not always the case, and such a restriction might result in a jobset which is not optimal. The effect of limiting the lookahead on LOS's results is examined in Section 4.4.

A more sophisticated possibility is not to use a constant lookahead, but rather to set this dynamically according to need and overhead. This is beyond the scope of the present paper and is left for future work.

4. Experimental results

4.1. *The simulation environment*

We implemented all aspects of the algorithm as described above and integrated them into the framework of an event-driven job scheduling simulator. In the simulations, we used workload logs from the Cornell Theory Center (CTC) IBM SP2, the San-Diego Supercomputer Center (SDSC) IBM SP2, and the Swedish Royal Institute of Technology (KTH) IBM SP2 parallel supercomputers [19]. Each log contains a list of jobs, and for each job, a record of its size, arrival time, actual and estimated runtimes, and other descriptive fields. Time related information is specified in seconds. Details about the logs are given in Table 5. Each simulation used the number of nodes available on the machine from which the log was taken. Unfortunately we cannot evaluate the algorithms for other arbitrary machine sizes, as that would dramatically change the packing properties of the jobs, and lead to unreliable results.

For each of these logs we calculated its *duration*, which is the difference between the arrival time of the last and the first jobs. We then calculated the log's *offered load* by multiplying the jobs sizes by their runtimes, summing these values, and then dividing the result by the log's duration and the size of the machine it represents. Given the offered load, we generated logs of varying loads ranging from 0.5 to 0.95 by multiplying the *arrival time* of each job by a constant factor. For example, if the offered load in the log is 0.60, then by multiplying each job's arrival time by $\frac{0.60}{0.90}$ the duration of the log is reduced, leading to a load of 0.9. This is better than changing the jobs' sizes, as that would affect their packing properties, because in the original logs most job sizes are powers of two. The logs were used as an input for the simulator, which generates *arrival* and *termination* events according to the specifications in the log.

On each arrival or termination event, the simulator invokes LOS which examines the waiting queue, and based on the current system state decides which jobs to start. For each started job, the simulator updates the system free capacity and enqueues a termination event corresponding to the job termination time. For each terminated job, the simulator records its response time, bounded slowdown (applying a threshold of $\tau = 10\,\mathrm{s}$), and wait time. Note that in this type of simulation the overall utilization is not a meaningful metric, as the system utilization is dictated by the workload and is not affected by the scheduler (this is essentially the $\rho = \lambda/\mu$ of open-systems queueing analysis). However, one must verify that the scheduler is not overwhelmed and that the system is not saturated. When this happens, the measured utilization becomes lower than the offered load.

4.2. *Improvement over EASY*

We used the framework mentioned above to run simulations of the EASY scheduler [15,24], and compared its results to those of LOS which was limited to a maximal lookahead of 50 jobs. The reason for comparing with EASY is that it is the most popular backfilling algorithm today, and is used in many systems.

By comparing the average system utilization vs. the offered load of each simulation, we saw that for the CTC and SDSC logs a discrepancy occurs at loads higher than 0.9 (Fig. 3(a,b)), whereas for the KTH log it occurs only at loads higher than 0.95 (Fig. 3(c)). Such discrepancies indicate that the simulated system is actually saturated, a state which is characterized by a continuously growing length of the waiting queue. As a result, all measured values are meaningless and depend on the length of the simulation. For this reason, when reporting our results, we limit the x-axis to the range where the simulation is still stable, and ignore the results beyond the point at which the discrepancy begins.

As the results of schedulers processing the same jobs may be similar, we need to compute confidence intervals to assess the significance of observed differences. Rather than doing so directly, we first apply the "common random numbers" variance reduction technique [13]. For each job in the workload file, we tabulate the *difference* between its response time (or slowdown) under EASY and under LOS. We then compute 90% confidence intervals on these differences

Table 5
The workload logs used to evaluate LOS

Log	Location	Nodes	Jobs	Load	Duration
CTC	Cornell Theory Ctr.	512	79,302	0.55	6/96–7/97
SDSC	San-Diego Supercomputer Ctr.	128	67,667	0.69	4/98–4/00
KTH	Royal Inst. Technology, Sweden	100	28,490	0.83	9/96–8/97

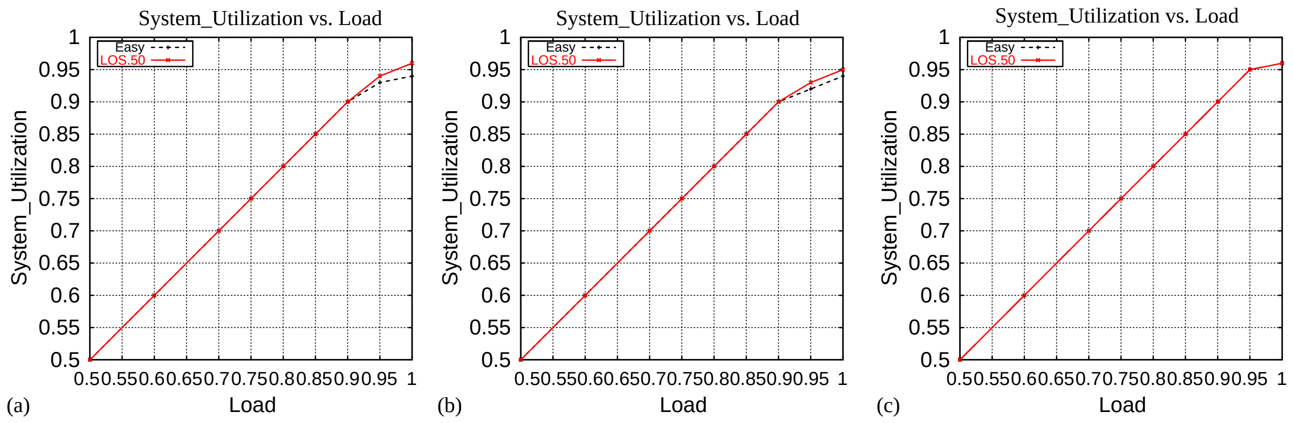


Fig. 3. Average system utilization vs. load: (a) CTC log, (b) SDSC log and (c) KTH log.

using the batch means approach. By comparing the difference between the schedulers on a job-by-job basis, the variance of the results is greatly reduced, and so are the confidence intervals.

The results for the response time are shown in Fig. 4, on the left of each sub-figure. The results for bounded slowdown are on the right. All graphs show the actual mean job response time (or bounded slowdown) of the two schedulers as well as the differential results. As can be seen, the mean job differential results are positive across the entire load range for both metrics and all three logs, indicating that LOS outperforms EASY in all cases. This result is statistically significant, as witnessed by the fact that all lower boundaries of the confidence intervals remain above zero. Comparing the actual results with the differential ones illustrates the significance of these differences and their dependency on the load. For example, by looking at sub-Fig. 4(a), we see that for a load of 0.75, the mean job differential response time is about 900 s. Comparing this to the actual mean job response time for the same load, we see that this is an improvement of 7% of the absolute value. On the other hand, at 0.90 load, the difference in response is 4200 s, which in absolute value means an improvement of more than 20%.

4.3. *Job selection effect on performance*

In Section 3.3, we introduced several alternative merit values for guiding the selection when several sets of jobs lead to the same utilization. The first was the *always-select* scheme. Unlike the original *always-skip* scheme which selects jobs closer to the head of the queue, the *always-select* scheme favors jobs near the tail. The results for the CTC log are shown in Fig. 5(a). We decided to focus our analysis on the mean job bounded slowdown metric, since it uses relative runtime values, and thus more accurately reflects the difference between the two algorithms. Results for KTH are generally similar to CTC, and SDSC are generally somewhat lower, but still show a positive difference; this is true for all job selection schemes.

We see that the mean job bounded slowdown difference is positive across the entire load range—a clear indication that the *always-select* algorithm outperforms the original *always-skip* with respect to this metric. On the other hand, if we compare the resulting plots to the corresponding plots in Fig. 4, where LOS was compared to EASY, we see that the curves here are significantly lower and in fact some of the lower boundaries of the confidence interval bars fall below zero. For example, the mean job differential bounded slowdown at 0.90 load for the CTC log in Fig. 5(a) is 2, while in Fig. 4(a) it is about 18. The reason for the low values is the fact that unlike Section 4.2, where we compared LOS to the suboptimal EASY algorithm, we now compare two versions of the same scheduler, both of which maximize the utilization, and only differ in their jobset selection. Therefore we can expect the performance gaps to be smaller.

Another approach we suggested was to maximize the number of jobs in the selected jobset. We stated that by considering the number of jobs which will start on each scheduling step, and selecting the jobset which holds the maximal number of jobs (in addition to maximizing utilization), LOS's performance is expected to improve since less jobs will remain waiting.

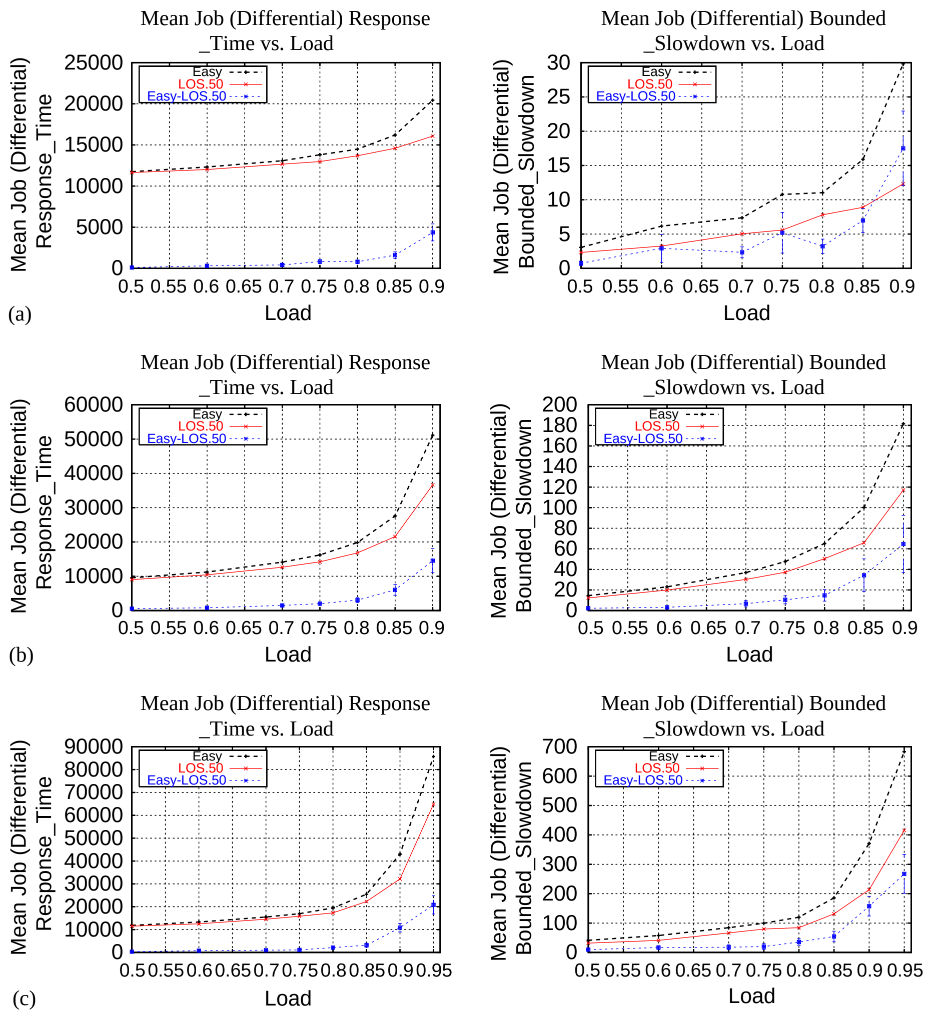


Fig. 4. Raw results and difference comparing LOS with EASY: (a) CTC log, (b) SDSC log and (c) KTH log.

The results of simulations using this approach are shown in Fig. 5(b). The fact that the mean job differential bounded slowdown remains positive for the entire load range indicates that the *max-jobs* approach also outperforms the original algorithm for constructing M'. But it is not significantly better than the *always-select* approach.

The third proposed approach was to select the jobset with the smaller sum of runtimes, so as to approximate the shortest job first scheduling scheme. As shown in Fig. 5(c), this indeed improves performance considerably more than the previous two schemes.

We also introduced the *Max-Slowdown* approach in which the set S' is chosen in a way that its overall total slowdown is maximized. The results using this approach are presented in Fig. 5(d). These results far exceed those of the *always-select* algorithm in Fig. 5(a) and the *max-jobs* approach in Fig. 5(b), and are even slightly better than the *shortest-jobs-first* approach in Fig. 5(c). This is also true for the other workloads; for example, the maximal differential bounded slowdown for the KTH log using *always-select* compared to *always-skip* is 60, using *max-jobs* it is 55, and for the *Max-Slowdown* approach it is 90.

To complete the performance evaluation, we also compared LOS when using the *Max-Slowdown* approach directly with the EASY scheduler. The results are shown in Fig. 6, for all three logs, again with the response time metric on the left and the bounded slowdown metric on the right. These should be compared with Fig. 4, where the *always-skip* algorithm for constructing S' was used. As can be seen, for all three logs and for the entire load range, the mean job differential bounded slowdown curves in Fig. 6 are higher than the corresponding curves in Fig. 4. The fact that the new curves are higher indicates that the *difference* between the jobs bounded slowdown under EASY and under LOS has increased. Since EASY was not modified, it is another indication that the *Max-Slowdown* approach further reduces the jobs' bounded slowdown, and thus outperforms the original algorithm. We can also assess the significance of the improvement in terms of absolute values. For example, in sub-Fig. 6(a), we see that at 0.90 load, the mean job bounded

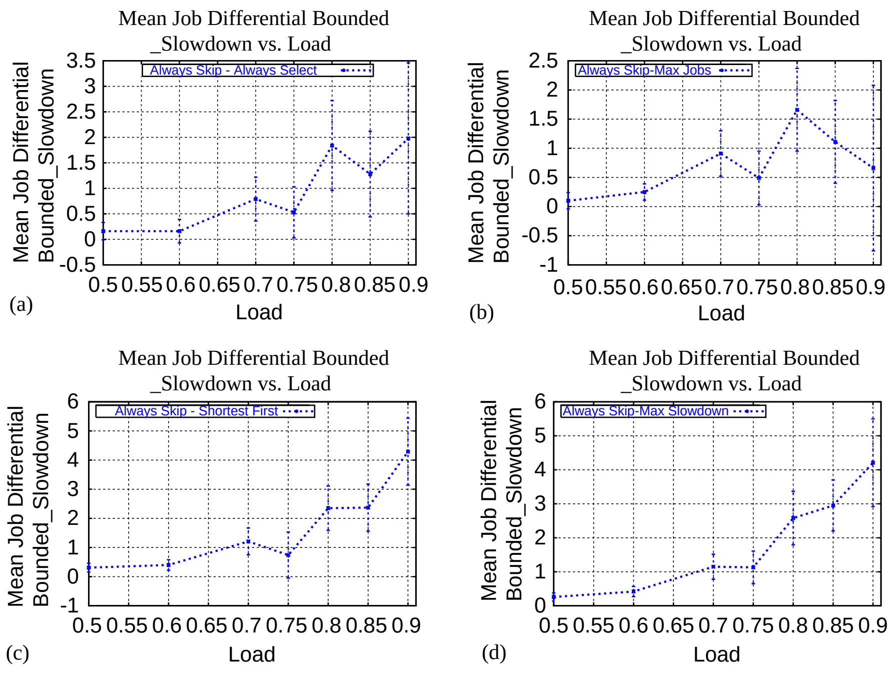


Fig. 5. Improved performance using different job selection schemes, CTC log: Improvement with: (a) *always select*, (b) *max jobs*, (c) *shortest first* and (d) *max slowdown*.

slowdown has dropped from 30 to 8, an improvement of more that 73%, while in the corresponding sub-Fig. 4(a) it improves by no more than 60%.

Considering the response time results, we see that for all three logs, the curves for the mean job differential response time of the *Max-Slowdown* approach are higher than those of the original algorithm, which means that the *Max-Slowdown* approach outperforms the original algorithm with respect to the response time metric as well. On the other hand, if we compare the absolute results, we see that at 0.90 load in SDSC and 0.95 in KTH, there is a slight advantage for the unmodified algorithm. This does not mean that the *Max-Slowdown* has failed to perform and in fact a positive mean response difference of 13,900 in SDSC (19,000 in KTH) is a major improvement over EASY, which is about 25% improvement in terms of absolute values. What this means is that for extremely high loads when the machine almost saturates, a change in the heuristic may be considered if the scheduler target is to minimize the response time of the jobs.

4.4. Limiting the lookahead

Section 3.4.2 proposed an enhancement called *limited lookahead* aimed at reducing the runtime of LOS. We explored the effect of limiting the lookahead on LOS's results by performing six LOS simulations with a limited lookahead of 10, 25, 35, 50, 100 and 250 jobs, respectively. Fig. 7 presents the effect of limiting the lookahead on the mean job response time and the mean job bounded slowdown. The notation *LOS.X* is used to represent LOS's results where *X* is the maximal number of waiting jobs that LOS was allowed to examine on each scheduling step (i.e. its lookahead limitation). We also plotted EASY's result curve to allow a comparison.

The lookahead limitation that can be tolerated without degrading the performance depends on the log. For the CTC log (Fig. 7(a)) we find that when LOS is limited to examine only 10 jobs at each scheduling step, its resulting mean job response time and bounded slowdown is relatively poor. In fact, the result curve of LOS.10 and EASY even intersect several times along the load axis, indicating that the two schedulers achieve similar results with neither one consistently outperforming the other as the load increases. The reason for the poor performance is the low probability that a jobset which maximizes the machine utilization actually exists within the first 10 waiting jobs, thus although LOS selects the best jobset it can, it is rarely the case that this jobset indeed maximizes the machine utilization. However, with a lookahead of 25 or more LOS achieved essentially optimal results.

For the SDSC log in Fig. 7(b), LOS manages to provide good performance even with a limited lookahead of only 10 jobs. But for the KTH log in Fig. 7(c), a lookahead of 50 is required for optimal performance, at least under the highest possible load. For lower loads, a lookahead of 10 suffices. To summarize, it seems that we can safely place a bound of 50 on the lookahead, and thus also bound the runtime of the algorithm.

The explanation for the good performance under limited lookahead is that for most of the scheduling steps, especially

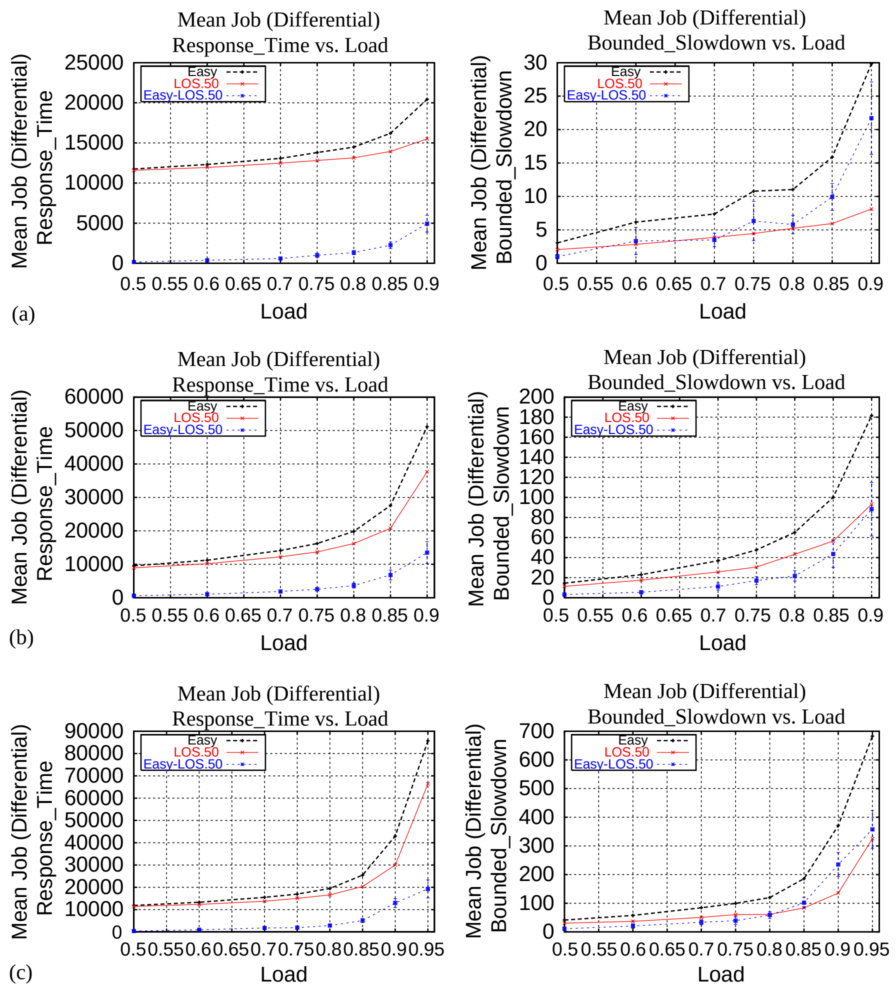


Fig. 6. Raw results and differences comparing LOS + *Max-Slowdown* with EASY: (a) CTC log, (b) SDSC log and (c) KTH log.

under low loads, the length of the waiting queue is actually small, so a lookahead of hundreds of jobs has no effect in practice. As the load increases and the machine advances toward its saturation point, the number of waiting jobs increases, and the effect of limiting the lookahead is more clearly seen. The left-hand side of Fig. 8 compares the mean queue length under EASY and LOS which was limited to a lookahead of 50 jobs. We can make two interesting observation based on these measurements. First, with LOS, the mean queue length is actually smaller compared to EASY, due to its efficiency in packing jobs, which allows more jobs to terminate faster. The second observation is that only for the KTH log in sub-Fig. 8(c), the mean number of waiting jobs exceeds the lookahead limitation of 50 jobs, and this happens only at a load of 0.95.

The problem with the plots on the left of Fig. 8 is that the mean queue length provides only a summary to what happened over the entire simulation. We therefore performed a detailed time-dependent analysis of the queue behavior, where the queue length is examined at every scheduling step across the entire simulation. The results show that although peaks of hundreds of jobs actually exist, they are relatively rare, and that LOS often manages to keep the queue length below 50 jobs at times when it reaches a length of more than 100 under EASY.

4.5. *Running time*

The main feature of the LOS algorithm is that it computes the optimal packing of queued jobs. As packing is in general NP-complete, this may raise concerns regarding the running time of the algorithm. In Section 3.4, we claimed that our specific instance of the packing problem is actually tractable, due to the relatively small number of processors (currently less than 10,000 even in the biggest machines in the world). To support this claim, we present runtime data from the simulations in Table 6. Note that the simulations contain a full implementation of LOS, and execute exactly the same algorithm as would be executed in a real system using LOS. Therefore the simulation time gives an upper bound on the running time of the algorithm in a real system processing the same workload.

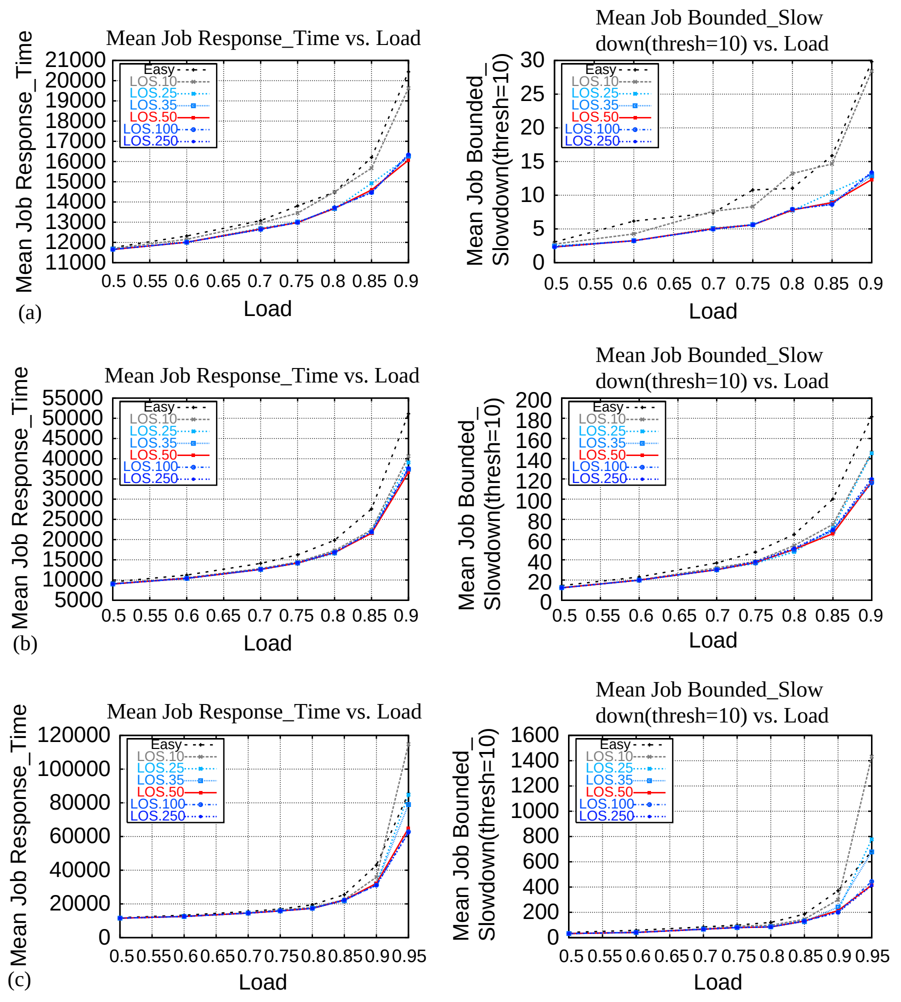


Fig. 7. Limited lookahead affect on mean job response time and bounded slowdown: (a) CTC log, (b) SDSC log and (c) KTH log.

The results are that the running time of the full simulation, at high load values, and processing up to nearly 80,000 jobs, is often less than a minute and never much more than two minutes. When this is divided by the number of jobs, we find that the average scheduling overhead per job is measured in milliseconds. The maximum measured in any single scheduling step is also much less than a second. While these numbers are significantly higher than for EASY (where simulations complete in a few seconds), they are still very low in absolute terms, especially relative to other overheads on parallel machines, where loading a parallel job for execution may take several minutes [1].

Thus we conclude that optimal packings can indeed be found realistically.

5. Conclusions

Backfilling algorithms have several parameters. In the past, two parameters have been studied: the number of jobs that receive reservations, and the order in which the queue is traversed when looking for jobs to backfill. We introduce a third parameter: the amount of lookahead into the queue. We show that by using a lookahead window of about 50 jobs it is possible to derive much better packing of jobs under high loads, and that this improves both the mean job response time and mean job bounded slowdown metrics.

In addition, improving packing positively effects secondary metrics such as the queue length behavior. We show that on heavily loaded systems under the control of traditional backfilling schedulers, the waiting queue length can reach tens of jobs with peaks sometimes reaching hundreds. On the other hand, when lookahead is used and packing is optimized, the waiting queue is shorter across a large fraction of the scheduling steps.

There is often more than a single way to pack jobs and achieve the same utilization value. We explored various alternatives by including a merit calculation in the lookahead process and choosing the set of jobs which maximizes the merit value. We show that performance is indeed sensitive to such choices, despite the fact that all lead to the same utilization. Surprisingly, performance is boosted

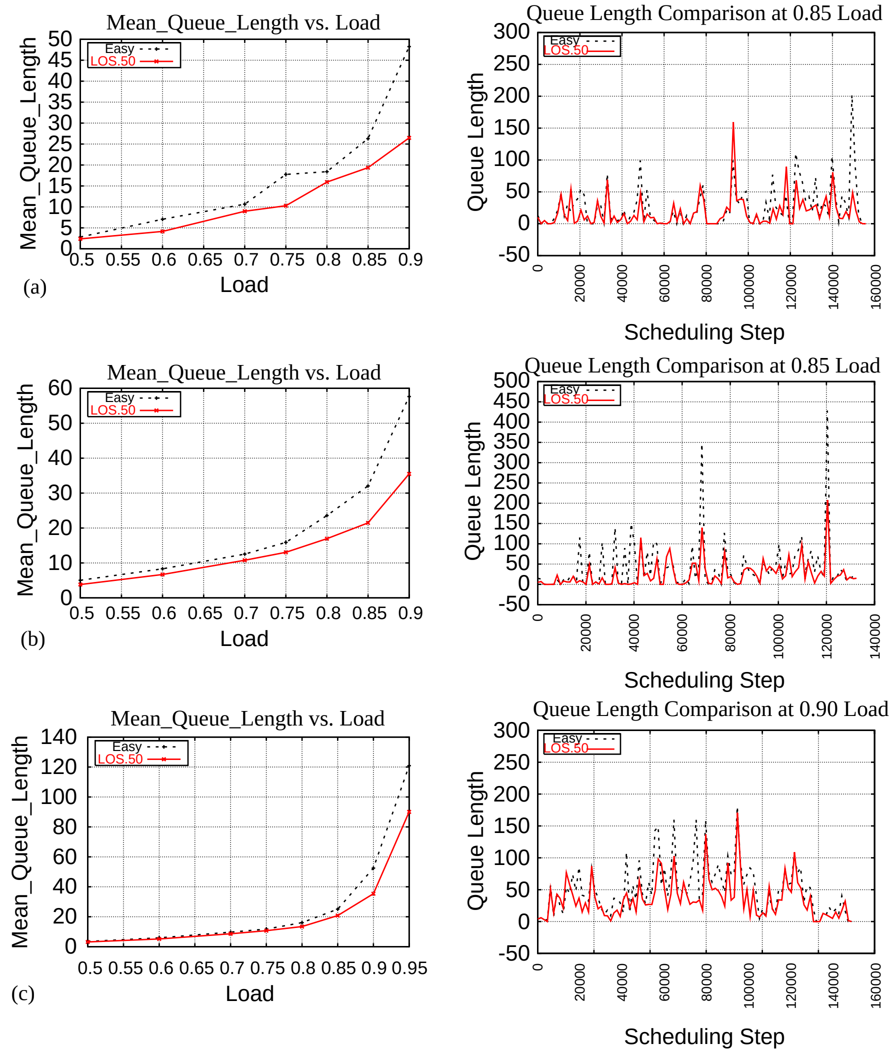


Fig. 8. Queue behavior in simulations: (a) CTC log, (b) SDSC log and (c) KTH log.

Table 6
Running time of simulations

Log	#Jobs	Load	Tot sim (s)	Avg time/job (ms)	Max time/step (ms)
CTC	79,302	0.90	129	1.63	613.3
SDSC	67,667	0.90	36	0.53	204.9
KTH	28,490	0.90	8	0.28	139.8

when choosing the set of jobs with the *maximal* total slowdown. A possible reason is the nature of the slowdown metric which is mostly effected by the shorter jobs; therefore a set with a large total slowdown is likely to contain the shortest jobs, and specifically, those short jobs that have waited the most. By starting these jobs ahead of other ones, a further degradation of the performance metrics is avoided.

Future work can further explore various ways to reduce the algorithm runtime. For example, it is possible to calculate the utilization in an on-going fashion and stop the construction of M' when the utilization reaches a certain threshold. Extending our algorithm to perform reservations for more than a single job and exploring the effect of such a heuristic on performance also presents an interesting challenge. On the other hand, it is possible to simplify the algorithm significantly by removing reservations altogether, and relying on the selection of jobs with maximal slowdowns to prevent starvation. The question is how well this performs in practice.

References

[1] G.A. Abandah, E.S. Davidson, Modeling the communication performance of the IBM SP2, in: D.G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), 10th International Parallel Processing Symposium, April 1996, pp. 249–257.

[2] O. Arndt, B. Freisleben, T. Kielmann, F. Thilo, A comparative study of on-line scheduling algorithms for networks of workstation, Cluster Comput. 3 (2) (2000) 95–112.

[3] V. Balasundaram, G. Fox, K. Kennedy, U. Kremer, A static performance estimator to guide data partitioning decisions, in: Third Symposium Principles and Practice of Parallel Programming, April 1991, pp. 213–223.

[4] S.-H. Chiang, A. Arpaci-Dusseau, M.K. Vernon, The impact of more accurate requested runtimes on production job scheduling performance, in: Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2537, Springer, Berlin, 2002, pp. 103–127.

[5] E.G. Coffman, M.R. Garey Jr., D.S. Johnson, Approximation algorithms for bin-packing—an updated survey, in: G. Ausiello, M. Lucertini, P. Serafini (Eds.), Algorithm Design for Computer Systems Design, Springer, Berlin, 1984, pp. 49–106.

[6] M.V. Devarakonda, R.K. Iyer, Predictability of process resource usage: a measurement based study on UNIX, IEEE Tans. Software Eng. 15 (12) (December 1989) 1579–1586.

[7] D.G. Feitelson, A Survey of Scheduling in Multiprogrammed Parallel Systems, Research Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994 (revised version, August 1997).

[8] D.G. Feitelson, Experimental analysis of the root causes of performance evaluation results: a backfilling case study, IEEE Trans. Parallel Distributed Systems 16 (2) (February 2005) 175–182.

[9] M.R. Garey, D.S. Johnson, "Strong" NP-completeness results: motivation, examples, and implications, J. Assoc. Comput. Mach. 25 (3) (July 1978) 499–508.

[10] D. Jackson, Q. Snell, M. Clement, Core algorithms of the Maui scheduler, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2221, Springer, Berlin, 2001, pp 87–102.

[11] J.P. Jones, B. Nitzberg, Scheduling for parallel supercomputing: a historical perspective of achievable utilization, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1659, Springer, Berlin, 1999, pp. 1–16.

[12] D. Karger, C. Stein, J. Wein, Scheduling algorithms, in: M.J. Atallah (Ed.), Handbook of Algorithms and Theory of Computation, CRC Press, Boca Raton, FL, 1997.

[13] A.M. Law, W.D. Kelton, Simulation Modeling and Analysis, third ed., McGraw Hill, New York, 2000.

[14] B.G. Lawson, E. Smirni, Multiple-queue backfilling scheduling with priorities and reservations for parallel systems, in: D.G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2537, Springer, Berlin, 2002, pp. 72–87.

[15] D. Lifka, The ANL/IBM SP scheduling systems, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 949. Springer, Berlin, 1995, pp. 295–303.

[16] S. Majumdar, D.L. Eager, R.B. Bunt, Scheduling in multiprogrammed parallel systems, in: SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1988, pp. 104–113.

[17] A.W. Mu'alem, D.G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Trans. Parallel Distributed Systems 12 (6) (June 2001) 529–543.

[18] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[19] Parallel Workloads Archive, URL http://www.cs.huji.ac.il/labs/parallel/workload.

[20] V. Sarkar, Determining average program execution times and their variance, in: Proceedings of SIGPLAN Conference Programming Language Design and Implementation, June 1989, pp. 298–312.

[21] K.C. Sevcik, Application scheduling and processor allocation in multiprogrammed parallel processing systems, Performance Evaluation 19 (2–3) (March 1994) 107–140.

[22] J. Sgall, On-line scheduling—a survey, in: A. Fiat, G.J. Woeginger (Eds.), Online Algorithms: The State of the Art, Lecture Notes in Computer Science, vol. 1442, Springer, Berlin, 1998, pp. 196–231.

[23] E. Shmueli, D.G. Feitelson, Backfilling with lookahead to optimize the performance of parallel job scheduling, in: D.G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2862, Springer, Berlin, 2003, pp. 228–251.

[24] J. Skovira, W. Chan, H. Zhou, D. Lifka, The EASY—loadleveler API project, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, Springer, Berlin, 1996, pp. 41–47.

[25] S. Srinivasan, R. Kettimuthu, V. Subramani, P. Sadayappan, Selective reservation strategies for backfill job scheduling, in: D.G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2537, Springer, Berlin, 2002, pp. 55–71.

[26] D. Talby, D.G. Feitelson, Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling, in: 13th International Parallel Processing Symposium, April 1999, pp. 513–517.

[27] W.A. Ward, C.L. Mahood Jr., J.E. West, Scheduling jobs on parallel systems using a relaxed backfill strategy, in: D.G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2537, Springer, Berlin, 2002, pp. 88–102.

Edi Shmueli is an IBM research staff member. He currently conducts research on advanced scheduling techniques for the IBM BlueGene/L supercomputer project, at the Haifa research laboratory in Israel. He received his B.Sc. (2002) and M.Sc. (2004) in Computer Science, from the Haifa University in Israel. His main research interests are architectures of high-performance parallel computers, their management and job scheduling systems, parallel programming models, operating systems and their interaction with underlying hardware.

Dror Feitelson is on the faculty of the School of Computer Science and Engineering of the Hebrew University of Jerusalem, where he received his Ph.D. in 1991. He is the founding co-organier of JSSPP, a series of annual workshops on Job Scheduling Strategies for Parallel Processing, that was started in 1995. His recent research deals with workload modeling, with the goal of placing performance evaluations of computer systems on a more solid scientific basis.