Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling

Edi Shmueli
Department of Computer Science
Haifa University, Haifa, Israel
IBM Haifa Research Labs
edi@il.ibm.com

Dror G. Feitelson
School of Computer Science & Engineering
Hebrew University, Jerusalem, Israel
feit@cs.huji.ac.il

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. Simulation results show that this indeed improves utilization, and thereby reduces the average response time and average slowdown of all jobs.

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 [2]. 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 such time that it terminates [3]. 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* [1].

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 submittal of a new job (which changes the queue), and the termination of a running job (which frees an allocated partition) [8]. Upon such events, 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 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 maximize utilization at each scheduling step. The different heuristics used by various algorithms are described in Section 2. These heuristics do not guarantee to minimize the machine's idle capacity.

We propose a new scheduling heuristic seeking 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, newly developed dynamic-programming based scheduling algorithm that chooses the set of jobs which will maximize the machine utilization and will not violate the reservation for the first waiting job. The algorithm also respects the arrival order of the jobs, if possible. When two or more sets of jobs achieve the same maximal utilization, it chooses the set closer to the head of the queue.

Section 3 provides a detailed description of the algorithm, followed by a short discussion of its complexity, and suggests optional performance optimizations. 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. Section 5 concludes on the effectiveness and applicability of our proposed scheduling heuristic.

2 Related Work

We will focus on the narrow field of on-line scheduling algorithms of non-preemptive rigid jobs on distributed memory parallel machines, and especially on heuristics that attempt to improve utilization.

The base case often used for comparison is the First Come First Serve (FCFS) algorithm [5]. In this algorithm all jobs are started in the same order in which they arrive in the queue. If the machine's free capacity does not allow the first job to start, FCFS will not attempt to start any succeeding 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. It is also easily implemented. Its drawback is the resulting poor utilization of the machine. When the next job to be scheduled is larger than the machine free capacity, it holds back smaller succeeding jobs, which could utilize the machine.

In order to improve various performance metrics it is possible to consider the jobs in some other order. The Shortest Processing Time First (SPT) algorithm uses estimations of the jobs' runtimes to make scheduling decisions. It sorts the waiting jobs by increasing estimated runtime and executes the jobs with the shortest runtime first [5]. This algorithm is inspired by the "shortest job first" heuristic [11], which seeks to minimize the average response time. The rationale behind this heuristics is that if a short job is executed after a long one, both will have a long response time, but if the short job gets to be executed first, it will have a short response time, thus the average response time is reduced.

The opposite algorithm, Largest Processing Time First (LPT), executes the jobs with the longest processing time first [15, 16]. This policy aims at minimizing the makespan, but the average response time is increased because many small jobs are delayed significantly.

Other scheduling heuristics base their decisions on job size rather than on estimated runtime. The Smallest Job First (SJF) algorithm [17] sorts the waiting jobs by increasing size and executes the smallest jobs first. Inspired by SPT, this algorithm turned out to perform poorly because there is not much correlation between the job size and it's runtime. Small jobs do not necessarily terminate quickly [18, 19], which results in a fragmented machine and thus a reduction in performance.

The alternative Largest Job First (LJF) 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 [20, 21]. In terms of scheduling it means that scheduling larger jobs first may be expected to cause less fragmentation and therefore higher utilization than FCFS.

Finally, the Smallest Cumulative Demand First [17, 22, 23] algorithm uses both the expected execution time and job size to make scheduling decisions. It sorts the jobs in an increasing order according to the product of the jobs size and the expected execution time, so small short jobs get the highest priority. It turned out that this policy does not perform much better than the original smallest job first [17].

The problem with all the above schemes is that they may suffer from starvation, and may also waste processing power 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 [3]. Backfilling is known to greatly increase user satisfaction since small jobs tend to get through faster, while bypassing large ones.

Note that in order to implement backfilling, the jobs' runtimes must be known in advance. Two techniques, one to estimate the runtime through repeated executions of the job [12] and the second to get this information through compile-time analysis [13, 14] have been proposed. Real implementations, however, require the users to provide an estimate of their jobs runtime, which in practice is often specified as a runtime upper-bound. Surprisingly, it turns out that inaccurate estimates generally lead to better performance than accurate ones [10].

Backfilling was first implemented on a production system in the "EASY" scheduler developed by Lifka et al. [24, 25], and later integrated with IBM's LoadLeveler. This version is based on aggressive backfilling, in which any job can be backfilled provided it does not delay the first job in the queue. In fact, one of the important parameters of backfilling algorithms is the number of jobs that enjoy reservations. In EASY, only the first job gets a reservation. In conservative backfilling, all skipped jobs get reservations [10]. The Maui scheduler has a parameter that allows the system administrator to set the number of reservations [9]. Srinivasan et al. [26] have suggested a compromise strategy called *selective backfilling*, wherein jobs do not get a reservation until their expected slowdown exceeds some threshold. If the threshold is chosen judiciously, only the most needy jobs get a reservation.

Additional variants of backfilling allow the scheduler more flexibility. Talby and Feitelson presented *slack based backfilling*, an enhanced backfill scheduler that supports priorities [6]. 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. 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 jobs from the queue can be executed [7]. Their simulation results indicate a performance gain compared to a single-queue backfilling, resulting from the fact that the multiple-queue policy reduces the likehood that short jobs get delayed in the queue behind long jobs.

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. Section 3.2 presents the basic algorithm, and shows how to find a set of jobs that together maximize utilization. Section 3.3 then extends this by showing how to select jobs that also respect a reservation for the first queued job. Section 3.4 describes the factors that effect the algorithm time and space complexity, and Section 3.5 finalizes the algorithm description with two suggested optimizations aimed at improving its performance.

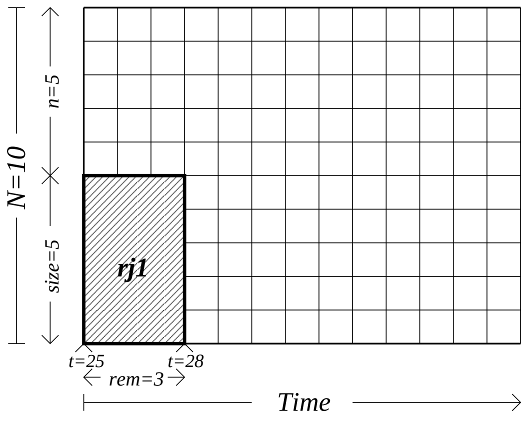
Before starting the description of the algorithm itself, Section 3.1 formalizes the state of the system and introduces the basic terms and notations used later. To provide an intuitive feel of the algorithms, each subsection is followed by an on-going scheduling example on an imaginary machine of size $N = 10$. Paragraphs describing the example are headed by ♣.

3.1 Formalizing the System State

At time t our machine of size N runs a set of jobs $R = \{rj_1, rj_2, ..., rj_r\}$, each with two attributes: their *size*, and 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 $WQ = \{wj_1, wj_2, .., wj_q\}$, 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 *produced schedule*, which maximizes the machine utilization. The produced schedule is $safe$ if it does not impose a risk of starvation.

♣ As illustrated in Figure 1, at $t = 25$, our machine runs a single job rj_1 with $size = 5$ and expected remaining execution time $rem = 3$. The machine's free capacity is $n = 5$. The table at the right describes the size and estimated runtime of the five waiting jobs in the waiting queue, WQ.



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

Figure 1: System state and queue at $t = 25$

3.2 The Basic Algorithm

3.2.1 Freedom of Starvation

The algorithm begins by trying to start the first waiting job.

If $wj_1.size \leq n$, 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 [24]. It does so by traversing the list of running jobs while accumulating their sizes until reaching a job rj_s at which $wj_1.size \leq n + \sum_{i=1}^{s} rj_i.size$. The shadow time is then defined to be $shadow = t + rj_s.rem$. By ensuring that *all* jobs in S terminate before that time, S is guaranteed to be a safe schedule, as it will not impose any delay on the first waiting job, thus ensuring a freedom from starvation.

To dismiss us of the concern of handling special cases, we set *shadow* to ∞ if wj_1 can be started at t. In this case *every* produced schedule 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 as illustrated in figure 2.

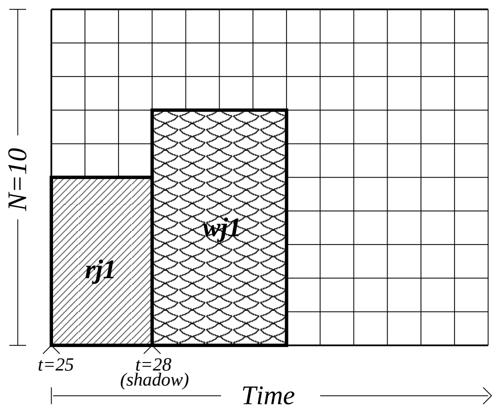


Figure 2: Computing the shadow time

3.2.2 A Two Dimensional Data Structure

After handling the first job, we need to find the set of subsequent 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 $(|WQ| + 1) \times (n + 1)$, and are later used for making successive decisions.

Each cell $m_{i,j}$ contains a single integer value *util*, and two boolean trace markers, *selected* and *bypassed*.

util holds the maximal achievable utilization at t, if the machine's free capacity is j and only waiting jobs $\{1..i\}$are available for scheduling.

The *selected* marker is set to indicate that wj_i was chosen for execution ($wj_i \in S$). The *bypassed* marker indicates the opposite. When the algorithm finishes calculating M, the trace markers are used to trace the jobs which construct S. It is possible that both markers will be set simultaneously in a given cell, which means that there is more than one way to construct S. It is important to note that either way, jobs in the produced schedule will always achieve the same overall maximal utilization.

For convenience, the $i = 0$ row and $j = 0$ column are initialized with zero values. Such padding eliminates the need of handling special cases.

♣ In the example, M is a 6×6 matrix. The *selected* and *bypassed* markers, if set, are noted by $\nwarrow$ and $\uparrow$ respectively. Table 1 describes M's initial values.

$\downarrow i$ (*size*) , $j \rightarrow$	0	1	2	3	4	5
0 (ϕ)	0	0	0	0	0	0
1 (7)	0	ϕ	ϕ	ϕ	ϕ	ϕ
2 (2)	0	ϕ	ϕ	ϕ	ϕ	ϕ
3 (1)	0	ϕ	ϕ	ϕ	ϕ	ϕ
4 (2)	0	ϕ	ϕ	ϕ	ϕ	ϕ
5 (3)	0	ϕ	ϕ	ϕ	ϕ	ϕ

Table 1: M's initial values

3.2.3 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 i to be started, we need to check whether including wj_i in the produced schedule increases the utilization. If not, or if the size of job i is larger than j, the utilization is simply what it was without this job, that is $m_{i-1,j}.util$.

As mentioned in Section 3.2.1, a safe schedule is guaranteed if all jobs in S terminate before the shadow time. The third line of Algorithm 1 ensures that every job wj_i that will not terminate by the shadow time is immediately *bypassed*, that is, excluded from S. This is done to simplify the presentation of the algorithm. In Section 3.3 we relax this restriction and present the full algorithm.

The computation stops when reaching cell $m_{|wq|,n}$ at which time M is filled with values.

♣ The resulting M is shown in Table 2. As can be seen, the *selected* flag is set only for wj_2, as it is the only job which can be started safely without imposing any delay on wj_1. Since all other jobs are *bypassed*, the maximal

Algorithm 1 Constructing M

- Note : To slightly ease the reading, $m_{i,j}.util$, $m_{i,j}.selected$, and $m_{i,j}.bypassed$ are represented by $util$, $selected$ and $bypassed$ respectively.

```
for $i = 1$ to $|WQ|$
    for $j = 1$ to $n$
        if $wj_i.size > j$ or $t + wj_i.time > shadow$
            $util \leftarrow m_{i-1,j}.util$
            $selected \leftarrow False$
            $bypassed \leftarrow True$
        else
            $util' \leftarrow m_{i-1,j-wj_i.size}.util + wj_i.size$
            if $util' \geq m_{i-1,j}.util$
                $util \leftarrow util'$
                $selected \leftarrow True$
                $bypassed \leftarrow False$
                if $util' = m_{i-1,j}.util$
                    $bypassed \leftarrow True$
            else
                $util \leftarrow m_{i-1,j}.util$
                $selected \leftarrow False$
                $bypassed \leftarrow True$
```

achievable utilization of the $j = 5$ free processors when considering all $i = 5$ jobs is $m_{5,5}.util = 2$.

$\downarrow i\ (size)\ ,\ j \rightarrow$	0	1	2	3	4	5
0 (ϕ)	0	0	0	0	0	0
1 (7)	0	0 $\uparrow$	0$\uparrow$	0$\uparrow$	0$\uparrow$	0$\uparrow$
2 (2)	0	0 $\uparrow$	2 $\nwarrow$	2 $\nwarrow$	2 $\nwarrow$	2 $\nwarrow$
3 (1)	0	0 $\uparrow$	2$\uparrow$	2$\uparrow$	2$\uparrow$	2$\uparrow$
4 (2)	0	0 $\uparrow$	2$\uparrow$	2$\uparrow$	2$\uparrow$	2$\uparrow$
5 (3)	0	0 $\uparrow$	2$\uparrow$	2$\uparrow$	2$\uparrow$	2$\uparrow$

Table 2: Resulting M

3.2.4 Constructing S

Starting at the last computed cell $m_{|wq|,n}$, S is constructed by following the trace markers as described in Algorithm 2.

It was already noted in Section 3.2.2 that it is possible that in an arbitrary cell $m_{x,y}$ both markers are set simultaneously, which means that there is more than one possible schedule. In such case, the algorithm will follow the *bypassed* marker.

In term of scheduling $wj_x \notin S$ simply means that wj_x is not started at t, but this decision has a deeper meaning in terms of queue policy. Since the queue is traversed by Algorithm 2 from tail to head, skipping wj_x means that other jobs, closer to the head of the queue will be started instead, and the same maximal utilization will still be achieved. By selecting jobs closer to the head of the queue our produced schedule is not only more committed to the queue FCFS policy, but also receives a better score from the evaluation metrics such as average response time, slowdown etc.

Algorithm 2 Constructing S

```
$S \leftarrow \{\}$
$i \leftarrow |WQ|$
$j \leftarrow n$
while $i > 0$ and $j > 0$
    if $m_{i,j}.bypassed = True$
        $i \leftarrow i - 1$
    else
        $S \leftarrow S \cup \{wj_i\}$
        $j \leftarrow j - wj_i.size$
        $i \leftarrow i - 1$
```

♣ The resulting S contains a single job wj_2, and its scheduling at t is illustrated in Figure 3. Note that wj_1 is not part of S. It is only drawn to illustrate that wj_2 does not effect its expected start time, indicating that our produced schedule is safe.

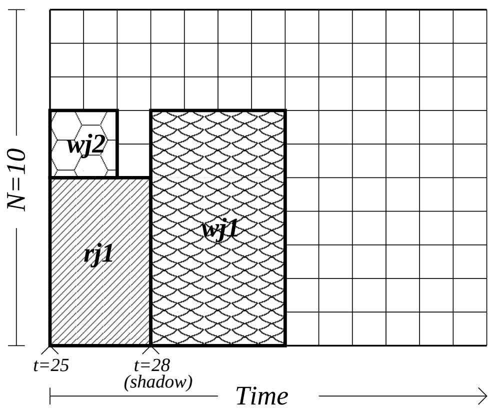


Figure 3: Scheduling wj_2 at $t = 25$

3.3 The Full Algorithm

3.3.1 Maximizing Utilization

One way to create a safe schedule is to require all jobs in S to terminate before the shadow time, so as not to interfere with that job's reservation. This restriction can be relaxed in order to achieve a better schedule S', still safe but with a much improved utilization. This is possible due to the *extra* processors left at the shadow time after wj_1 is started. Waiting jobs which are expected to terminate *after* the shadow time can use these extra processors, referred to as the *shadow free capacity,* and run side by side together with wj_1, without effecting its start time. As long as the total size of jobs in S' that are still running at the shadow time does not exceed the shadow free capacity, wj_1will not be delayed, and S' will be a safe schedule.

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 *ssize,* and is calculated using the following rule:

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

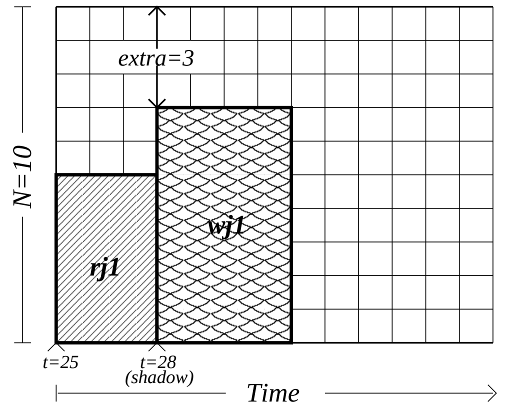
If wj_1 can start at t, the shadow time is set to ∞. As a result, the shadow size *ssize*, of *all* waiting jobs is set to 0, which means that any computation which involves extra processors is unnecessary. In this case setting *extra* to 0 improves the algorithm performance.

All these calculation are done in a pre-processing phase, before running the dynamic programming algorithm.

♣ wj_1which can begin execution at $t = 28$ leaves 3 extra processors. *shadow* and *extra* are set to 28 and 3 respectively, as illustrated in Figure 4. In the queue shown on the right, we use the notation $size_{ssize}$ to represent the two size values. wj_2 is the only job expected to terminate before the shadow time, thus its shadow size is 0.

3.3.2 A Three Dimensional Data Structure

To manage the use of the *extra* processors, we need a three dimensional matrix denoted M' of size $(|WQ| + 1) \times (n + 1) \times (extra + 1)$.



wj	$size_{ssize}$	$time$
1	7_7	4
2	2_0	2
3	1_1	6
4	2_2	4
5	3_3	5

Figure 4: Computing *shadow* and *extra*, and the processed job queue

Each cell $m'_{i,j,k}$ now contains two integer values, *util* and *sutil*, and the two trace markers.

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 $\{1..i\}$ are available for scheduling.

sutil hold the minimal number of extra processors required to achieve the *util* value mentioned above.

The *selected* and *bypassed* markers are used in the same manner as described in section 3.2.2.

As mentioned in section 3.2.2, the $i = 0$ rows and $j = 0$ columns are initialized with zero values, this time for all k planes.

♣ M' is a $6 \times 6 \times 4$ matrix. *util* and *sutil* are noted $util_{sutil}$. The notation of the *selected* and *bypassed* markers is not changed and remains $\nwarrow$ and $\uparrow$ respectively.

Table 3 describes the initial $k = 0$ plane. Planes 1..3 are initially similar.

$\downarrow i\ (size_{ssize})\ , j \rightarrow$	0	1	2	3	4	5
0 (ϕ_ϕ)	0_0	0_0	0_0	0_0	0_0	0_0
1 (7_7)	0_0	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ
2 (2_0)	0_0	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ
3 (1_1)	0_0	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ
4 (2_2)	0_0	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ
5 (3_3)	0_0	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ	ϕ_ϕ

Table 3: Initial $k = 0$ plane

3.3.3 Filling M'

The values in every $m'_{i,j,k}$ cell are calculated in an iterative matter using values from previously calculated cells as described in Algorithm 3. The calculation is exactly the same as in Algorithm 1, except for an addition of a slightly more complicated condition that checks that enough processors are available both now *and* at the shadow time.

The computation stops when reaching cell $m'_{|wq|,n,extra}$.

Algorithm 3 Constructing M'

- Note : To slightly ease the reading, $m'_{i,j,k}.util$, $m'_{i,j,k}.sutil$, $m'_{i,j,k}.selected$, and $m'_{i,j,k}.bypassed$ are represented by $util$, $sutil$, $selected$, and $bypassed$ respectively.

```
for $k = 0$ to $extra$
    for $i = 1$ to $|WQ|$
        for $j = 1$ to $n$
            if $wj_i.size > j$ or $wj_i.ssize > k$
                $util \leftarrow m'_{i-1,j,k}.util$
                $sutil \leftarrow m'_{i-1,j,k}.sutil$
                $selected \leftarrow False$
                $bypassed \leftarrow True$
            else
                $util' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.util + wj_i.size$
                $sutil' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.sutil + wj_i.ssize$
                if $util' > m'_{i-1,j,k}.util$ or
                  ($util' = m'_{i-1,j,k}.util$ and $sutil' \leq m'_{i-1,j,k}.sutil$)
                    $util \leftarrow util'$
                    $sutil \leftarrow sutil'$
                    $selected \leftarrow True$
                    $bypassed \leftarrow False$
                    if $util' = m_{i-1,j,k}.util$ and $sutil' = m_{i-1,j,k}.sutil$
                        $m'_{i,j,k}.bypassed \leftarrow True$
                else
                    $util \leftarrow m'_{i-1,j,k}.util$
                    $sutil \leftarrow m'_{i-1,j,k}.sutil$
                    $selected \leftarrow False$
                    $bypassed \leftarrow True$
```

♣ When the shadow free capacity is $k = 0$, only wj_2 who's $ssize = 0$ can be scheduled. 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 Table 4. This is of course the same utilization value (and the same schedule) achieved in Section 3.2.3, as the $k = 0$ case is identical to considering only jobs that terminate before the shadow time.

When the shadow free capacity is $k = 1$, wj_3 who's $ssize = 1$ is also available

$\downarrow i\ (size_{ssize}), j \rightarrow$	0	1	2	3	4	5
$0\ (\phi_\phi)$	0_0	0_0	0_0	0_0	0_0	0_0
$1\ (7_7)$	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
$2\ (2_0)$	0_0	$0_0 \uparrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$
$3\ (1_1)$	0_0	$0_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$
$4\ (2_2)$	0_0	$0_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$
$5\ (3_3)$	0_0	$0_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$	$2_0 \uparrow$

Table 4: $k = 0$ plane

$\downarrow i\ (size_{ssize}), j \rightarrow$	0	1	2	3	4	5
$0\ (\phi_\phi)$	0_0	0_0	0_0	0_0	0_0	0_0
$1\ (7_7)$	0_0	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$	$0_0 \uparrow$
$2\ (2_0)$	0_0	$0_0 \uparrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$
$3\ (1_1)$	0_0	$1_1 \nwarrow$	$2_0 \uparrow$	$3_1 \nwarrow$	$3_1 \nwarrow$	$3_1 \nwarrow$
$4\ (2_2)$	0_0	$1_1 \uparrow$	$2_0 \uparrow$	$3_1 \uparrow$	$3_1 \uparrow$	$3_1 \uparrow$
$5\ (3_3)$	0_0	$1_1 \uparrow$	$2_0 \uparrow$	$3_1 \uparrow$	$3_1 \uparrow$	$3_1 \uparrow$

Table 5: $k = 1$ plane

for scheduling. As can be seen in Table 5, starting at $m'_{3,3,1}$ the maximal achievable utilization is increased to 3, at the price of using a single extra processor. The two *selected* jobs are wj_2 and wj_3.

As the shadow free capacity increases to $k = 2$, wj_4 who's shadow size is 2, joins wj_2 and wj_3 as a valid scheduling option. Its effect is illustrated in Table 6 starting at $m'_{4,4,2}$, as the maximal achievable utilization has increased to 4 — the sum of wj_2 and wj_4 sizes. This comes at a price of using a minimum of 2 extra processors, corresponding to wj_4's shadow size.

It is interesting to examine the $m'_{4,2,2}$ cell, as it introduces an interesting heuristic decision. When the machine's free capacity is $j = 2$ and only jobs $\{1..4\}$ are considered for scheduling, the maximal achievable utilization can be accomplished by either scheduling wj_2 or wj_4, both with a size of 2, yet wj_4 will use 2 extra processors while wj_2 will use none. The algorithm chooses to bypass wj_4 and selects wj_2 as it leaves more extra processors to be used by other jobs.

Finally the full $k = 3$ shadow free capacity is considered. wj_5, who's shadow size is 3 can now join $wj_1..wj_4$ as a valid scheduling option.

As can be seen in Table 7, 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 available for scheduling is $m'_{5,5,3}.util = 5$. The minimal number of extra processors required to achieve this utilization value is

$\downarrow i\ (size_{ssize})\ ,\ j \rightarrow$	0	1	2	3	4	5
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 (2_0)	0_0	$0_0 \uparrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$
3 (1_1)	0_0	$1_1 \nwarrow$	$2_0\uparrow$	$3_1\nwarrow$	$3_1\nwarrow$	$3_1\nwarrow$
4 (2_2)	0_0	$1_1 \uparrow$	$2_0\uparrow\star$	$3_1\uparrow$	$4_2\nwarrow$	$4_2\nwarrow$
5 (3_3)	0_0	$1_1 \uparrow$	$2_0\uparrow$	$3_1\uparrow$	$4_2\uparrow$	$4_2\uparrow$

Table 6: $k = 2$ plane

$\downarrow i\ (size_{ssize})\ ,\ j \rightarrow$	0	1	2	3	4	5
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 (2_0)	0_0	$0_0 \uparrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$	$2_0 \nwarrow$
3 (1_1)	0_0	$1_1 \nwarrow$	$2_0\uparrow$	$3_1\nwarrow$	$3_1\nwarrow$	$3_1\nwarrow$
4 (2_2)	0_0	$1_1 \uparrow$	$2_0\uparrow$	$3_1\uparrow$	$4_2\nwarrow$	$5_3\nwarrow$
5 (3_3)	0_0	$1_1 \uparrow$	$2_0\uparrow$	$3_1\uparrow$	$4_2\uparrow$	$5_3\nwarrow\uparrow$

Table 7: $k = 3$ plane

$m'_{5,5,3}.sutil = 3$.

3.3.4 Constructing S'

Algorithm 4 describes the construction of S'. It starts at the last computed cell $m'_{|wq|,n,extra}$, follows the trace markers, and stops when reaching the 0 boundaries of any plane.

As explained in section 3.2.4, when both trace markers are set simultaneously, the algorithm follows the *bypassed* marker, a decision which receives a better score from the evaluation metrics.

♣ Both trace markers in $m'_{5,5,3}$, are set, which means there is more than one way to construct S'. In our example there are two possible schedules, both utilize all 5 free processors, resulting in a fully utilized machine. Choosing $S' = \{wj_2, wj_3, wj_4\}$ is illustrated in Figure 5. Choosing $S' = \{wj_2, wj_5\}$ is illustrated in Figure 6.

Both schedules fully utilize the machine and ensure that wj_1 will start without a delay, thus both are safe schedules, yet the first schedule (illustrated in Figure 5) contains jobs closer to the head of the queue, thus it is more committed to the queue FCFS policy. Based on the explanation in section 3.2.4, choosing $S' = \{wj_2, wj_3, wj_4\}$ is expected to gain better results when evaluation metrics are considered.

Algorithm 4 Constructing S'

```
$S' \leftarrow \{\}$
$i \leftarrow |WQ|$
$j \leftarrow n$
$k \leftarrow extra$
while $i > 0$ and $j > 0$
    if $m'_{i,j,k}.bypassed = True$
        $i \leftarrow i - 1$
    else
        $S' \leftarrow S' \cup \{wj_i\}$
        $j \leftarrow j - wj_i.size$
        $k \leftarrow k - wj_i.ssize$
        $i \leftarrow i - 1$
```

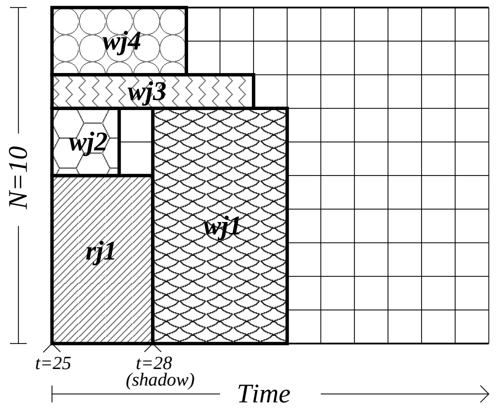


Figure 5: Scheduling wj_2, wj_3 and wj_4 at $t = 25$

3.4 A Note On Complexity

The most time and space demanding task is the construction of M'. It depends on $|WQ|$ — 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. On heavy loaded systems the average waiting queue length can reach tens of jobs with peaks reaching sometimes hundreds.

Both n and $extra$ fall in the range of 0 to N. Their values depend on the size and time distribution of the waiting and running jobs. A termination of a small job causes nothing but a small increase to the system's free capacity, thus n is increased by a small amount. On the other hand, when a large job terminates, it leaves much free space and n will consequently be large. $extra$ is a function of

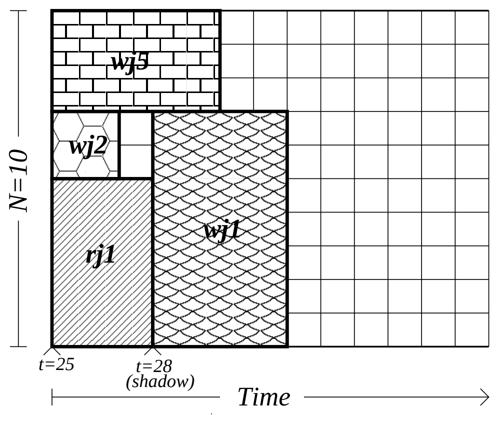


Figure 6: Scheduling wj_2 and wj_5 at $t = 25$.

the size of the first waiting job, and the size and time distribution of the running jobs. If wj_1 is small but it can start only after a large job terminates, *extra* will consequently be large. On the other hand, if the size of the terminating job is small and wj_1's size is relatively large, fewer *extra* processors will be available.

3.5 Optimizations

It was mentioned in Section 3.4 that on heavily loaded systems the average waiting queue length can reach tens of jobs, a fact that has a negative effect on the performance of the scheduler, since the construction of M' directly depends on $|WQ|$. Two enhancements can be applied in the pre-processing phase. Both result in a shorter waiting queue $|WQ'| < |WQ|$ and thus improve the scheduler performance.

The first enhancement is to exclude jobs larger than the machine's current free capacity. If $wj_i.size > n$ it is clear that it will not be started in the current scheduling step, so it can be safely excluded from the waiting queue without any effect on the algorithm results.

The second enhancement is to limit the number of jobs examined by the algorithm by including only the the first C waiting jobs in WQ' where C is a predefined constant. We call this approach *limited lookahed* 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 by limiting the lookahead, the same result are achieved, but with much less computation effort. Obviously this is not always the case, and such a restriction might produce a schedule which is not optimal. The effect of limiting the lookahead on the performance of LOS is examined in Section 4.3.

♣ Looking at our initial waiting queue described in the table in Figure 4, it is clear that wj_1 cannot start at t since its size exceeds the machine's 5 free processors. Therefore it can be safely excluded from the processed waiting queue without effecting the produced schedule. The resulting waiting queue WQ' holds only four jobs as shown in Table 8.

wj	$size_{ssize}$
2	2_0
3	1_1
4	2_2
5	3_3

Table 8: Optimized Waiting Queue WQ'

We could also limit the lookahead to $C = 3$ jobs, excluding wj_5 from WQ'. In this case the produced schedule will contain jobs wj_2, wj_3 and wj_4, and not only that it maximizes the utilization of the machine, but it is also identical to the schedule shown in Figure 5. By limiting the lookahead we improved the performance of the algorithm and achieved the same results.

4 Experimental Results

4.1 The Simulation Environment

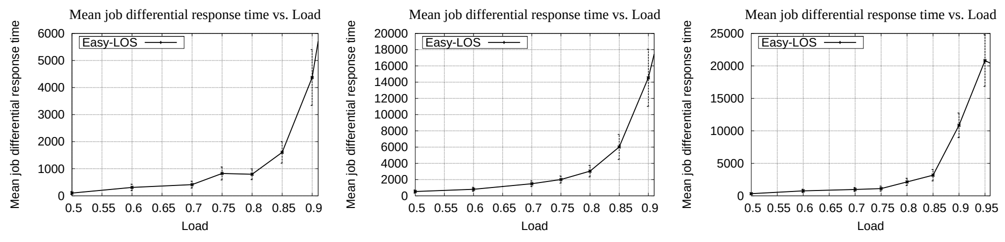
We implemented all aspects of the algorithm including the mentioned optimizations in a job scheduler we named LOS, and integrated LOS into the framework of an event-driven job scheduling simulator. We used logs of the Cornell Theory Center (CTC) SP2, the San Diego Supercomputer Center (SDSC) SP2, and the Swedish Royal Institute of Technology (KTH) SP2 supercomputing centers as a basis [28], and generated logs of varying loads ranging from 0.5 to 0.95, by multiplying the *arrival time* of each job by constant factors. For example, if the offered load in the CTC log is 0.60, then by multiplying each job's arrival time by 0.60 a new log is generated with a load of 1.0. To generate a load of 0.9, each job's arrival time is multiplied by a constant of $\frac{0.60}{0.90}$. We claim that in contrast to other log modification methods which modify the jobs' sizes or runtimes, our generated logs and the original ones maintain resembling characteristics. The logs were used as an input for the simulator, which generates *arrival* and *termination* events according to the jobs characteristics of a specific log.

On each arrival or termination event, the simulator invokes LOS which examines the waiting queue, and based on the current system state it decides which jobs to start. For each started job, the simulator updates the system free capacity and enqueues a *temination* event corresponding to the job termination time. For each terminated job, the simulator records its response time, bounded slowdown (applying a threshold of $\tau = 10$ seconds), and wait time.

4.2 Improvement over EASY

We used the framework mentioned above to run simulations of the EASY scheduler [24, 25], and compared its results to those of LOS which was limited to a maximal lookahead of 50 jobs. By comparing the achieved utilization vs. the offered load of each simulation, we saw that for the CTC and SDSC workloads a discrepancy occurs at loads higher than 0.9, whereas for the KTH workload it occurs only at loads higher than 0.95. As such discrepancies indicate that the simulated system is actually saturated, we limit the x axis to the indicated ranges when reporting our results.

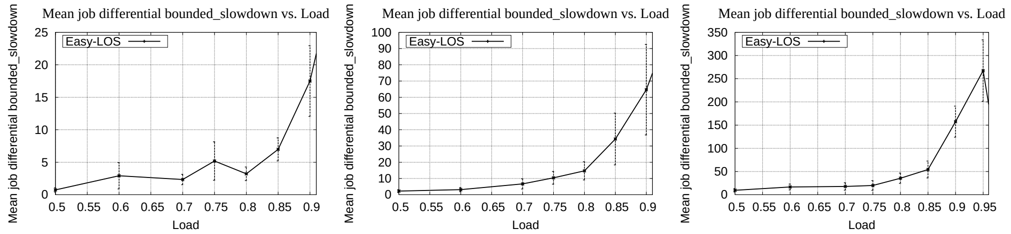
As the results of schedulers processing the same jobs may be similar, we need to compute confidence intervals to assess the significance of observed differences. Rather than doing so directly, we first apply the "common random numbers" variance reduction technique [29]. For each job in the workload file, we tabulate the *difference* between its response time under EASY and under LOS. We then compute confidence intervals on these differences 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.



(a) CTC Log (b) SDSC Log (c) KTH Log

Figure 7: Mean job differential response time *vs* Load

The results for response time are shown in Figure 7, and for bounded slowdown in Figure 8. The results for wait time are the same as those for response time, because we are looking at differences. In all the plots, the mean job differential response time (or bounded slowdown) is positive across the entire load range for all three logs, indicating that LOS outperforms Easy with respect to these metrics. This observation is reinforced by that fact that all lower boundaries of the 90% confidence interval measured at key load values, remain above the load axis, indicating the accuracy of our results.



(a) CTC Log (b) SDSC Log (c) KTH Log

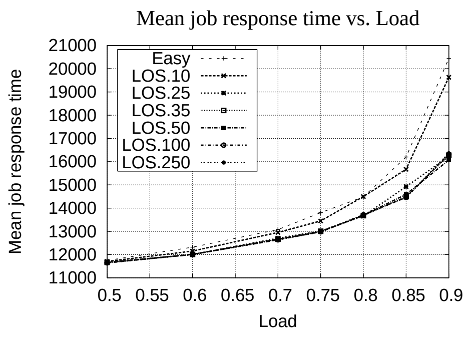
Figure 8: Mean job differential bounded slowdown ($\tau = 10$) *vs* Load

4.3 Limiting the Lookahead

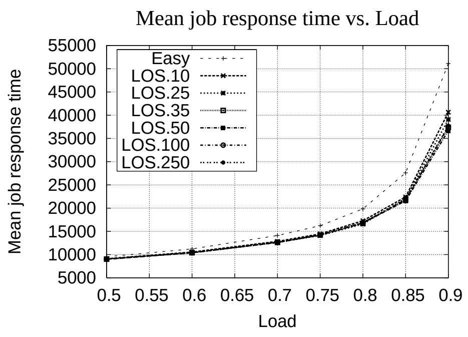
Section 3.5 proposed an enhancement called *limited lookahead* aimed at improving the performance of the algorithm. We explored the effect of limiting the lookahead on the scheduler performance by performing six LOS simulations with a limited lookahead of 10, 25, 35, 50, 100 and 250 jobs respectively. Figure 9 present the effect of the limited lookahead on the mean job response time. Figure 10 presents its effect on the mean job bounded slowdown. Again, the effect on wait time is the same as that on response time.

The notation *LOS.X* is used to represent LOS's result curve, 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. We observe that for the CTC log in Figure 9(a) and the KTH log in Figure 9(c), when LOS is limited to examine only 10 jobs at each scheduling step, its resulting mean job response time is relatively poor, especially at high loads, compared to the result achieved when the lookahead restriction is relaxed. The same observation also applies to the mean job bounded slowdown for these two logs, as shown in figure 10(a,c). As most clearly illustrated in figures 9(a) and 10(a), the result curves of LOS and Easy intersect several times along the load axis, indicating that the two schedulers achieve the same results with neither one consistently outperforming the other as the load increases . The reason for the poor performance is the low probability that a schedule which maximizes the machine utilization actually exists within the first 10 waiting jobs, thus although LOS produces the best schedule it can, it is rarely the case that this schedule indeed maximizes the machine utilization. However, for the SDSC log in Figures 9(b) and 10(b), LOS manages to provide good performance even with a limited lookahead of 10 jobs.

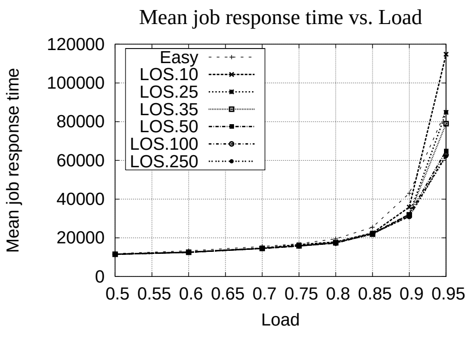
As the lookahead limitation is relaxed, LOS performance improves but the improvement is not linear with the lookahead factor, and in fact the resulting curves for both metrics are relatively similar for lookaheads in the range of 25–250 jobs. Thus we can safely use a bound of 50 on the lookahead, thus bounding



(a) CTC Log



(b) SDSC Log

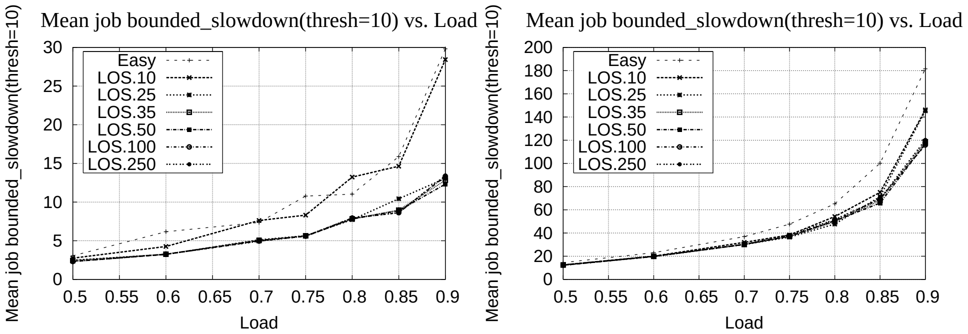


(c) KTH Log

Figure 9: Limited lookahead affect on mean job response time

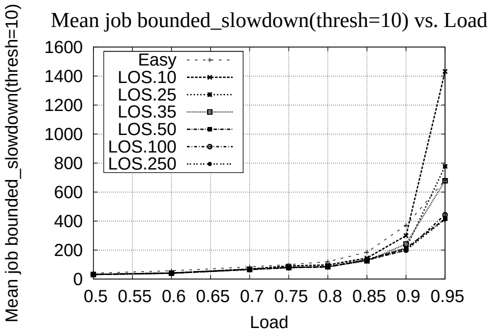
the complexity of the algorithm.

The explanation is that at most of the scheduling steps, especially under low loads , the length of the waiting queue is kept small, so lookahead of hundreds of jobs has no effect in practice. As the load increases and the machine advances toward its saturation point, the average number of waiting jobs increases, as shown in Figure 11, and the effect of changing the lookahead is more clearly seen. Interestingly, with LOS the average queue length is actually shorter, because it is more efficient in packing jobs, thus allowing them to terminate faster.



(a) CTC Log

(b) SDSC Log



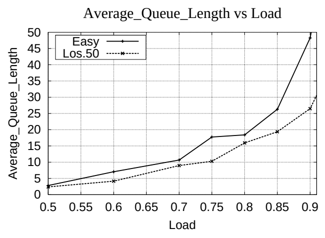
(c) KTH Log

Figure 10: Limited lookahead affect on mean job bounded slowdown ($\tau = 10$)

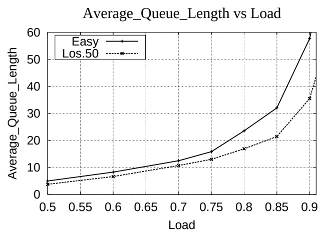
5 Conclusions

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 average response time and average bounded slowdown metrics.

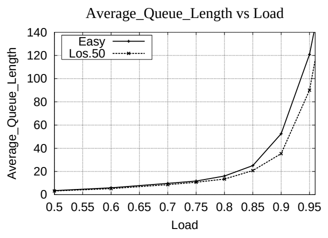
A future study should explore how the packing effects secondary metrics such as the queue length behavior. In Section 3.4 we stated that on a heavily loaded system the waiting queue length can reach tens of jobs, so a scheduler capable of maintaining a smaller queue across large portion of the scheduling



(a) CTC log



(b) SDSC log



(c) KTH log

Figure 11: Average queue length *vs* Load

steps, increases the users' satisfaction with the system. Alternative algorithms for constructing S' when several optional schedules are possible might also be examined. In Section 3.2.4 we stated that by following the *bypassed* marker we expect a better score from the evaluation metrics, but other heuristics such as choosing the schedule with the minimal overall expected termination time are also worthy of evaluation. Finally, extending our algorithm to perform reservations for more than a single job and exploring the effect of such a heuristic on performance presents an interesting challenge.

References

[1] D. G. Feitelson, *"A Survey of Scheduling in Multiprogrammed Parallel Systems"*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994 - The revised version, Aug 1997.

[2] D. G. Feitelson and L. Rudolph, *"Toward Convergence in Job Schedulers for Parallel Supercomputers"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 1162, pp. 1-26, 1996.

[3] D. G. Feitelson, L. Rudolph, U. Schweigelshohn, K. C. Sevcik and P. Wong, *"Theory and Practice in Parallel Job Scheduling"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 1291, pp 1-34, 1997.

[4] D. G. Feitelson and L. Rudolph, *"Metrics and Benchmarking for Parallel Job scheduling"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 1459, pp. 1-24, 1998.

[5] O. Arndt, B. Freisleben, T. Kielmann and F. Thilo, *"A Comparative Study of On-Line Scheduling Algorithms for Networks of Workstation"*. Cluster Computing 3(2), pp. 95-112, 2000.

[6] D. Talby and D. G. Feitelson, *"Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling"*. In 13th Intl. Parallel Processing Symp. (IPPS), pp 513-517, Apr 1999.

[7] B. G. Lawson and E. Smirni, *"Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2537, pp. 72-87, 2002.

[8] E. Krevat, J. G. Castanos and J. E .Moreira , *"Job Scheduling for the Blue-Gene/L System"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2537, pp. 38-54, 2002.

[9] D. Jackson, Q. Snell, and M. Clement, "*Core Algorithms of the Maui Scheduler*". In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2221, pp 87–102, 2001.

[10] A. W. Mu'alem and D. G. Feitelson, *"Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling"*, In IEEE Trans. on Parallel and Distributed Syst. 12(6), pp. 529-543, Jun 2001.

[11] S. Krakowiak, *"Principles of Operating Systems"*. The MIT Press, Cambridge Mass., 1998.

[12] M. V. Devarakonda and R. K. Iyer, *"Predictability of Process Resource Usage : A Measurement Based Study on UNIX"*. IEEE Tans. Sotfw. Eng. 15(12), pp. 1579-1586, Dec 1989.

[13] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, *"A Static Performance Estimator to Guide Data Partitioning Decisions"*. In 3rd Symp. Principles and Practice of Parallel Programming, pp. 213-223, Apr 1991.

[14] V. Sarkar, *"Determining Average Program Execution Times and Their Variance"*. In Proc. SIGPLAN Conf. Prog. Lang. Design and Implementation, pp. 298-312, Jun 1989.

[15] D. Karger, C. Stein and J. Wein, *"Scheduling Algorithms"*. In Handbook of algorithms and Theory of computation, M. J. Atallah, editor. CRC Press, 1997.

[16] J. Sgall, *"On-Line Scheduling — A Survey"*. In Online Algorithms: The State of the Art, A. Fiat and G. J. Woeginger, editors, Springer-Verlag, 1998. Lect. Notes Comput. Sci. Vol. 1442, pp. 196-231.

[17] S. Majumdar, D. L. Eager, and R. B. Bunt, *"Scheduling in Multiprogrammed Parallel Systems"*. In SIGMETRICS Conf. Measurement and Modeling of Comput. Syst., pp. 104-113, May 1988.

[18] S. T. Leutenegger and M.K. Vernon, *"The Performance of Multiprogrammed Multiprocessor Scheduling Policies"*. In SIGMETRICS Conf. Measurement and Modeling of Comput. Syst., pp. 226-236, May 1990.

[19] P. Krueger, T-H. Lai, and V. A. Radiya, *"Processor Allocation vs. Job Scheduling on Hypercube Computers"*. In 11th Intl. Conf. Distributed Comput. Syst., pp. 394-401, May 1991.

[20] E. G. Coffman, Jr., M. R. Garey , D. S. Johnson, and R. E. Tarjan, *"Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms"*. SIAM J. Comput. 9(4), pp. 808-826, Nov 1980.

[21] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, *"Approximation Algorithms for Bin-Packing - An Updated Survey"*. In Algorithm Design for Computer Systems Design, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49-106, Springer-Verlag, 1984.

[22] S. T. Leutenegger and M. K. Vernon, *"Multiprogrammed Multiprocessor Scheduling Issues"*. Research Report RC 17642 (#77699), IBM T. J. Watson Research Center, Nov 1992.

[23] K. C. Sevick, *"Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems"*. Performance Evaluation 19(2-3), pp. 107-140, Mar 1994.

[24] D. Lifka, *"The ANL/IBM SP Scheduling System"*, In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), pp. 295-303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. Vol. 949.

[25] J. Skovira, W. Chan, H. Zhou, and D. Lifka, *"The EASY - LoadLeveler API Project"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), pp. 41-47, Springer-Verlag, 1996. Lect. Notes Comput. Sci. Vol. 1162.

[26] S. Srinivasan, R. Kettimuthu, V. Subramani and P. Sadayappan, *"Characterization of Backfilling Strategies for Parallel Job Scheduling"*. In Proceedings of 2002 Intl. Workshops on Parallel Processing, Aug, 2002.

[27] W. A. Ward, Jr., C. L. Mahood and J. E. West *"Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2537, pp. 88-102, 2002.

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

[29] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis.* 3rd ed., McGraw Hill, 2000.