Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling

Edi Shmueli

Thesis Submitted In Partial Fulfillment Of The Requirements For The Master Degree

University of Haifa

Faculty of Social Sciences

Department of Computer Science

10, 2003

Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling

By : Edi Shmueli
Supervised by : Dr. Dror G. Feitelson
Supervised by : Prof. Alek Vainshtein

Thesis Submitted In Partial Fulfillment Of The Requirements For The M.A Degree

University of Haifa
Faculty of Social Sciences
Department of Computer Science

10, 2003

Approved by : _____ Date : _____
(Supervisor)

Approved by : _____ Date : _____
(Supervisor & Chairman of M.A Committee)

Special Thanks

I'd like to thank

Contents

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. We expect that by maximizing the utilization on every scheduling step, the overall performance of the system will improve.

We developed a dynamic programming based scheduling algorithm that looks at the entire content of the waiting queue and chooses the set of jobs which together maximize the machine utilization, while ensuring that long-waiting jobs will not be starved. We implemented the algorithm in a job scheduler we named LOS — an acronym for "Lookahead Optimizing Scheduler", and integrated LOS into the framework of an event-driven job scheduling simulator. We then ran simulations of LOS on trace files of real parallel systems and compared its results to those of traditional backfilling algorithms.

The results show that LOS indeed improves utilization, and thereby reduces the mean response time and mean slowdown of all jobs which are key metrics used for on-line systems. We also found that is not necessary to examine the whole waiting queue to reach high performance, and that we can limit the lookahead depth and still achieve the same results but with much less computation effort.

Finally, we experimented with selections among alternative groups of jobs that achieve the same utilization in the interest of improving other performance metrics. Surprising

simulation results indicate that choosing the group at the head of the queue does not necessarily guarantee best performance. Instead, repeatedly selecting the group with the maximal overall expected slowdown boost performance when compared to all other alternatives.

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

A distributed memory *parallel machine* consists of a set of processors, each associated with a private memory, which are connected using a fast network. A *parallel job* is composed of a number of concurrently executing processes communicating using message passing, 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, 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.

1.1 The Goals of the Job Scheduler

While the primary goal of all schedulers is to enable a successful execution of jobs, different scheduling algorithms try to optimize certain secondary global or local goals aimed at satisfying groups or individual needs respectively [2], and thus choose to start different jobs at different scheduling steps.

To better understand these goals, often referred to as *metrics*, we will look at an example in which at $t = 0$ four jobs $j_1..j_4$, each attributed with a *size* and an estimated runtime *time*, had been submitted and placed in the waiting queue of a parallel machine of size $N = 5$.

The queue state and a possible schedule of the four jobs are illustrated in Figures 1.1 and 1.2 respectively.

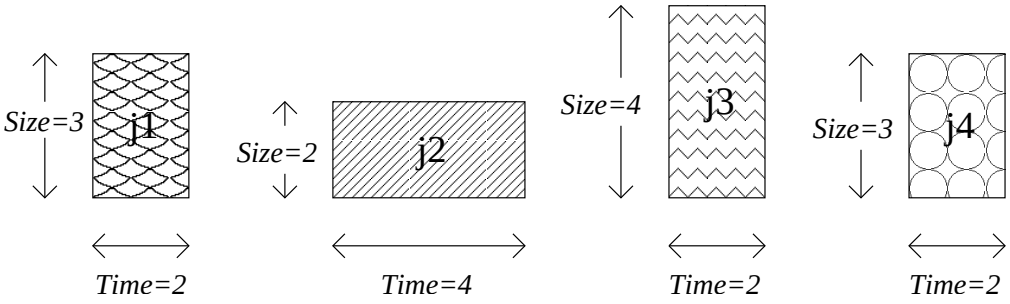


Figure 1.1: The waiting queue holds four jobs headed by j_1.

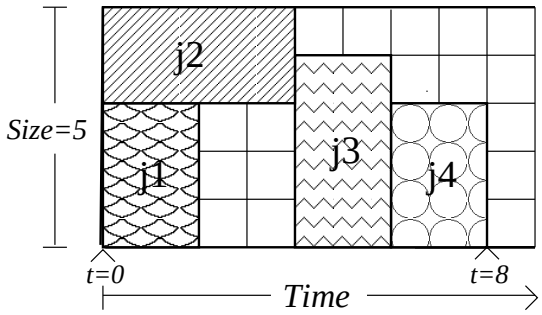


Figure 1.2: A possible schedule of the four jobs

The following values are calculated for each job j in the resulting schedule :

- *arrival time* is the time at which the job had arrived at the waiting queue.

- *start time* is the time at which the job had started executing.
- *termination time* is the time at which the job had terminated.
- $response\ time = termination\ time - arrival\ time$, often referred to as $flow\ time$ [5], is the total time the job has spent in the system, either waiting in the queue or running.
- $running\ time = termination\ time - start\ time$ is the actual runtime of the job.
- $wait\ time = response\ time - running\ time$ is the time the job had spent waiting in the queue.
- $slowdown = \frac{response\ time}{running\ time}$ is the ratio of the time it takes to run the job on a loaded system divided by the time it takes on dedicated system.

Table 1.1 shows the calculated values for the schedule in Figure 1.2

j_i	$arrival$	$start$	$termination$	$response$	$runtime$	$wait$	$slowdown$
1	0	0	2	2	2	0	1
2	0	0	4	4	4	0	1
3	0	4	6	6	2	4	3
4	0	6	8	8	2	6	4

Table 1.1: Computed values for the example schedule

Minimizing *makespan* - the difference between the termination time of the last job and the arrival time of the first job is a common global goal, primarily used for off-line[1] scheduling. As a metric it measures the performance of the system in the sense that the entire set of jobs will be scheduled in such a way that it takes as little time as possible to finish all jobs without focusing on a single job.

In our example schedule, the makespan achieved is 8.

A second and equally important global goal is to maximize the machine *utilization* which is the capacity of the machine that was utilized over its activity period. Utilization is defined as

[1]Where all jobs and their resource requirements are known in advance

$$Utilization = \frac{\sum_i j_i.size \times j_i.running\,time}{makespan \times N}$$

In our example, $Utilization = \frac{3\times2+2\times4+4\times2+3\times2}{8\times5} = 0.70$.

Makespan and utilization are highly related, but the utilization metric is also often used for on-line[2] scheduling, since it is largely dependent on the load [2]. With low loads when all jobs can be serviced, utilization is equal to the load, but as load increases and the machine saturated, the utilization is equal to the saturation point. Schedulers that focus on maximizing utilization will try to delay the onset of saturation to higher loads, but by doing so certain jobs may be starved.

Minimizing the *mean job response time* is a very common local goal especially in interactive (i.e. on-line) systems [2]. Obviously, the lower bound on the response time of a given job is its running time. The main problem with using mean job response time as a performance metric is its use of absolute values. Two jobs that had responded in one hour, but one required a full hour of computation while the other required only one second, might indicate a problem with the scheduler, but if both had been running for 50 minutes than one hour of response is pretty good.

A possible solution to this problem is to use the *mean job slowdown* metric instead, thus a job that takes twice as long to run due to system load, will suffer from a slowdown factor of 2 etc. Slowdown is widely perceived as better matching user expectations that a job's response time will be proportional to its running time. The problem with the slowdown metric is that it over emphasizes the importance of very short jobs [4]. A job with computation requirements of 100ms that had been delayed for 10 minutes will suffer from a slowdown of 6000 whereas a 10-second job delayed by the same 10 minutes has a slowdown of only 60.

To avoid such effects, Feitelson et al. have suggested the *bounded slowdown* metric [3]. The difference is that for short jobs, this measures the slowdown relative to some "interactive threshold" rather than relative to the actual runtime. Denoting this threshold by τ , the definition is

[2]Where future jobs and their resource requirements are not known in advance

$$bounded\ slowdown = max\left\{\frac{response\ time}{max\{running\ time,\ \tau\}}, 1\right\}$$

This metric behavior obviously depends on the choice of τ which typically takes the values in the range of 10 seconds to several minutes.

1.2 The Lookahead Optimizing Scheduler

Serving as a general-purpose computation core, the parallel machine is shared over a period of time by wide range of users executing jobs with various resource requirements. This mode of work is known as an *on-line* mode [4, 16] and is distinguished from an *off-line* mode in which all jobs and their resource requirements are known in advance.

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 Chapter 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 basic 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. 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. To reach these conclusions we developed

and examined a set of enhanced algorithms, built on top and conceptually similar to the basic algorithm, which in addition to maximizing the utilization of the machine, also guarantee that the chosen set of jobs will maximize or minimize a predefined merit value.

Chapter 3 provides a detailed description of the algorithm. It continues with a description of the enhanced algorithms and concludes with a discussion on complexity, followed by performance optimizations. Chapter 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, compares and analyzes LOSs' results when using any of the enhanced algorithms. Chapter 5 concludes on the effectiveness and applicability of our proposed scheduling heuristic.

Chapter 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" (the Extensible Argonne Scheduling sYstem) 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. The objective is to improve the current utilization as much as possible but the price is that execution guarantees cannot be made because it is impossible to predict how much each job will be delayed in the queue. The EASY backfilling algorithm is described in Appendix A.1. It is executed repeatedly whenever a new job arrives or a running job terminates, if the first job in the queue cannot start. In each iteration, the algorithm identifies a job that can backfill if one exists.

There are two interesting properties associated with this algorithm. First, queued jobs may suffer an unbounded delay because if a job is not the first in the queue, new jobs that arrive latter may skip it in the queue and impose delays on it, which makes predictability impossible. Second, there is no starvation because the queuing delay for the job at the head of the queue depends only on jobs that are already running since backfilled jobs will not delay it. Thus, it is guaranteed to eventually run since the running jobs will either terminate or be terminated when they exceed their estimated runtime. A detailed proof of the above two properties is found in [10].

By using aggressive backfilling EASY sacrifices predictability for potentially improving utilization. When predictability is required, one can use "Conservative" backfilling which performs all scheduling decisions upon job submittal and thus, has the capability of predicting when each job will run, giving the users execution guarantees. With conservative backfilling, users can plan ahead based on these guaranteed response times. In this version, backfilling is done subject to checking that it does not delay *any* previous job in the queue. To perform allocations, conservative backfilling maintains two data structures. One is the list of queued jobs and the time at which they are expected to start execution. The other is a profile of the expected processor usage at future times. Appendix A.2 describes the Conservative backfill algorithm. It is executed whenever a new job arrives. Note that Conservative backfilling has no danger of starvation as a reservation is made for each job when it is submitted.

Mu'alem and Feitelson [10] compared EASY backfilling to conservative backfilling.

Their simulation results show that for most cases the performance of the EASY backfilling algorithm was better than that of conservative backfilling.

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 while in conservative backfilling, all skipped jobs get reservations. The Maui scheduler [9] has a parameter that allows the system administrator to set the number of reservations. When Maui schedules, it prioritizes the jobs in the queue according to a number of factors and then orders the jobs in a highest priority-first sorted list. By default, Maui reserves only the highest priority job resulting in a most liberal and aggressive backfill. This give Maui the freedom to optimize its schedule and thus to potentially result in a better job response times and overall system utilization. While this reservation ensures that the highest priority job will not be delayed, other jobs lack a resource protection, and thus potentially could be significantly delayed. A tunable parameter, RESERVATIONDEPTH provides the ability to control how deep in the priority queue reservation should be made. In its default value, 1, Maui backfills aggressively with the purpose of maximizing utilization. As the value increases, the liberal backfilling behavior moves toward a more conservative one in which resource protection and thus predictability become available.

Srinivasan et al. [26] have studied the relative effectiveness of conservative and aggressive backfilling by grouping jobs into categories based on their size and runtime, and examining their effect on jobs in different categories. They observed that conservative and aggressive backfilling each benefit certain job categories while adversely affecting other categories. They proposed a compromise strategy called *selective backfilling* with the purpose of obtaining the best characteristics from both the conservative and the aggressive backfilling. With selective backfilling, reservations are provided selectively only to jobs whos their expected slowdown exceeds some threshold. By limiting the number of reservations the amount of backfilling is greater than conservative backfilling, but by assuring reservations to jobs after a limited wait, the disadvantage of potentially unbounded delay with aggressive backfill is avoided.

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.

Chapter 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 examines selection among alternative groups of jobs that achieve the same utilization value in the interest of improving other performance metrics. Section 3.5 analyzes the complexity of the algorithm, and finalizes the algorithm description with two suggested optimizations aimed at reducing its complexity.

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 ♣.

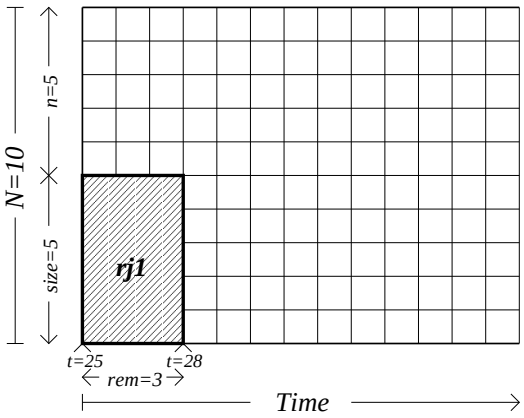
3.1 Formalizing the System State

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 3.1, 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 3.1: 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 3.2.

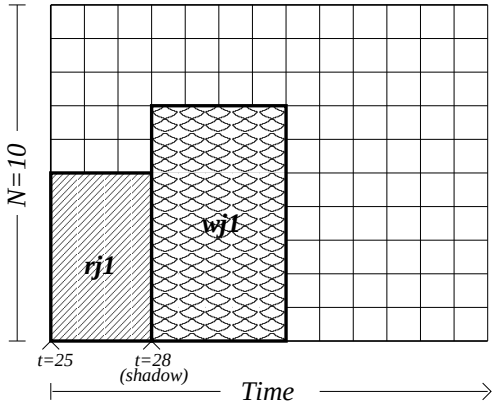


Figure 3.2: Computing the shadow time

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 3.1 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 3.1: 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.

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 ← m_{i-1,j}.util
            selected ← False
            bypassed ← True
        else
            util' ← m_{i-1,j-wj_i.size}.util + wj_i.size
            if util' ≥ m_{i-1,j}.util
                util ← util'
                selected ← True
                bypassed ← False
                if util' = m_{i-1,j}.util
                    bypassed ← True
            else
                util ← m_{i-1,j}.util
                selected ← False
                bypassed ← True
```

♣ The resulting M is shown in Table 3.2. 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 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 ↑	0↑	0↑	0↑	0↑
2 (2)	0	0 ↑	2 ↖	2 ↖	2 ↖	2 ↖
3 (1)	0	0 ↑	2↑	2↑	2↑	2↑
4 (2)	0	0 ↑	2↑	2↑	2↑	2↑
5 (3)	0	0 ↑	2↑	2↑	2↑	2↑

Table 3.2: 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 terms 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 more committed to the queue FCFS policy, and is expected to receive a better score from the evaluation metrics such as average response time, slowdown etc.

Algorithm 2 Constructing S

```
S ← {}
i ← |WQ|
j ← n
while i > 0 and j > 0
    if m_{i,j}.bypassed = True
        i ← i − 1
    else
        S ← S ∪ {wj_i}
        j ← j − wj_i.size
        i ← i − 1
```

♣ The resulting S contains a single job wj_2, and its scheduling at t is illustrated in Figure 3.3. 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.

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

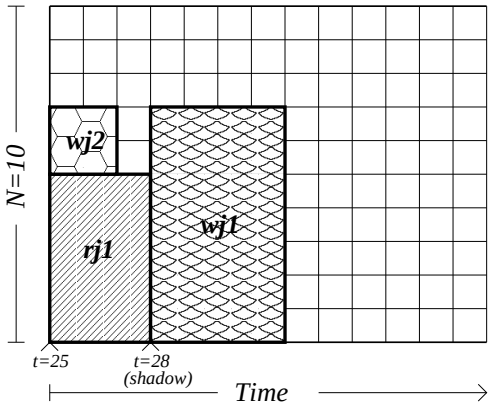


Figure 3.3: Scheduling wj_2 at $t = 25$

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 :

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

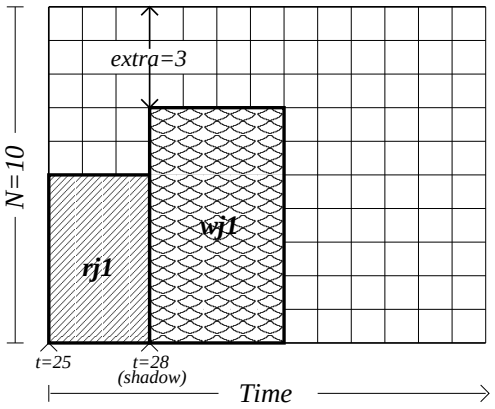
To use the extra processors, the jobs which are expected to terminate before the shadow time are distinguished from those that are expected to still run at that time, and are therefore candidates for using the extra processors. Each waiting job $wj_i \in WQ$ will now be represented by two values: its 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 3.4. 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.



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 3.4: Computing $shadow$ and $extra$, and the processed job queue

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 $(|WQ| + 1) \times (n + 1) \times (extra + 1)$.

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 3.3 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 3.3: 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 $m'_{|wq|,n,extra}$.

♣ 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 3.4. 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 for scheduling. As can be seen in Table 3.5, 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*

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 ← m'_{i-1,j,k}.util
                sutil ← m'_{i-1,j,k}.sutil
                selected ← False
                bypassed ← True
            else
                util' ← m'_{i-1,j-wj_i.size,k-wj_i.ssize}.util + wj_i.size
                sutil' ← 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' ≤ m'_{i-1,j,k}.sutil)
                    util ← util'
                    sutil ← sutil'
                    selected ← True
                    bypassed ← False
                    if util' = m_{i-1,j,k}.util and sutil' = m_{i-1,j,k}.sutil
                        m'_{i,j,k}.bypassed ← True
                else
                    util ← m'_{i-1,j,k}.util
                    sutil ← m'_{i-1,j,k}.sutil
                    selected ← False
                    bypassed ← True
```

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 3.6 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

$\downarrow i\ (size_{ssize}),\ j \rightarrow$	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	$0_0\uparrow$	$2_0\uparrow$	$2_0\uparrow$	$2_0\uparrow$	$2_0\uparrow$
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 3.4: $k = 0$ 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	$1_1\uparrow$	$2_0\uparrow$	$3_1\uparrow$	$3_1\uparrow$	$3_1\uparrow$
5 (3_3)	0_0	$1_1\uparrow$	$2_0\uparrow$	$3_1\uparrow$	$3_1\uparrow$	$3_1\uparrow$

Table 3.5: $k = 1$ plane

processors to be used by other jobs.

$\downarrow i\ (size_{ssize}),\ j \rightarrow$	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 3.6: $k = 2$ plane

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

$\downarrow i\ (size_{ssize}),\ j \rightarrow$	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 3.7: $k = 3$ plane

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 is closer to the FCFS policy.

Algorithm 4 Constructing S'

```
S' ← {}
i ← |WQ|
j ← n
k ← extra
while i > 0 and j > 0
    if m'_{i,j,k}.bypassed = True
        i ← i − 1
    else
        S' ← S' ∪ {wj_i}
        j ← j − wj_i.size
        k ← k − wj_i.ssize
        i ← i − 1
```

♣ 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 3.5. Choosing $S' = \{wj_2, wj_5\}$ is illustrated in Figure 3.6.

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 3.5) contains jobs closer to the head of the queue, thus it is more committed to the queue FCFS policy.

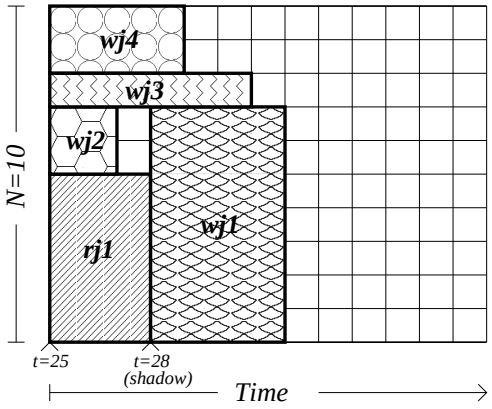


Figure 3.5: Scheduling wj_2, wj_3 and wj_4 at $t = 25$

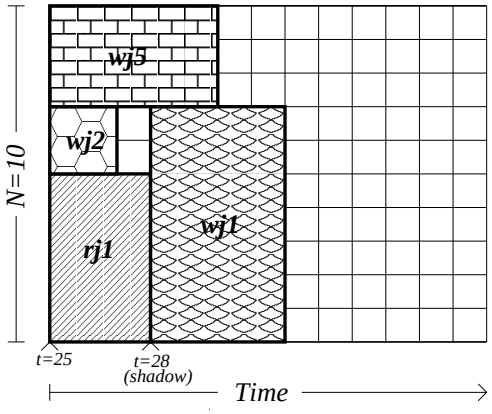


Figure 3.6: Scheduling wj_2 and wj_5 at $t = 25$.

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.

3.4 Improving Performance by Job Selection

In Section 3.2.4 we stated that in the case where both trace markers are set in a give cell, following the *bypassed* marker is expected to produce better results, since by doing so we start jobs which are closer to the head of the queue, and thus we are more committed to the queue FCFS policy. To verify this assumption we performed the following experiment: We modified algorithm 4 so it will follow the *selected* marker first, that is, whenever both

markers are set in a given cell, it will start the current job instead of bypassing it. The effect of this modification is that jobs which are closer to the *tail* of the queue are given precedence over jobs which where submitted at earlier times. We named the modified algorithm the *"Selected-First"* algorithm to distinguish it from the original *"Bypassed-First"* behavior, and describe it formally in Algorithm 5.

Algorithm 5 Constructing S'- *Selected-First* Algorithm

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

We expected the *Selected-First* algorithm to perform poorly since its produced schedule, *S'*, is no longer committed the queue's FCFS policy, but simulation results have proven the opposite and LOS performance has improved against expectations. (The simulation results are presented in Section 4.3.1).

Such surprising results have proven that basic assumptions which are often based on pure intuition, such as the one stating that selecting jobs closer to the head of the queue will improve performance, might be misleading. This opened the door to a set of experiments aimed at the purpose of improving LOS's performance and exploring the cause for the results differences. In all experiments, we enhance our three dimensional data structure which was described in Section 3.3.2 by including an additional *merit* value in every $m'_{i,j,k}$ cell, in addition to the existing, *util*, *sutil*, and the two trace markers. We also modified LOS's core algorithm for constructing M' (Algorithm 3) to consider the merit value. Whenever the same utilization value can be achieved, either by selecting or bypassing job i, a case in which both the *selected* and the *bypassed* markers were set by the original algorithm, the modified algorithm considers the merit value in order to eliminate one of the options, if possible. By doing so, the number of optional selections

is minimized and the produced schedule, *S'*, is optimized in view of the merit.

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 produced schedule, *S'*, will still maximize the machine utilization. The only difference is that now there are less cells in which both trace markers are set, and thus, less freedom to choose the set of jobs which construct *S'*. It is also important to understand that the construction of *M'* has not changed and that *M'* is still filled using the same iterative algorithm as described in Section 3.3.3, thus the use of the merit does not change the complexity of the algorithm.

We started with a simple experiment in which the merit value was simply the number of selected jobs in the path, with the purpose of choosing the set *S'* which contains the maximal number of jobs. We named this the *Maxjobs* approach and it is described in Subsection 3.4.1. Simulation results have shown that by starting the maximal number of jobs (in addition to maximizing utilization), the performance of LOS is improved. The reason for the improvement is that less jobs remain waiting and thus the mean response time and slowdown are reduced.

Next we examined various merit values such as the total jobs response time with the purpose of choosing *S'* with the maximal or minimal total response time, and total jobs slowdown with the purpose of maximizing or minimizing that factor also. Simulation results indicate that the performance of LOS has improved or reduced with respect to the chosen merit.

Peak performance was observed when the merit was the total jobs slowdown with the purpose of choosing the set *S'* which *maximizes* this factor. Again, this goes against intuition which states that when several sets of jobs exist, all of which achieve the same utilization value, it is expected that choosing the set with the minimal total slowdown will improve performance, since if we delay those jobs in the waiting queue, their slowdown (and response time) will increase. Unfortunately, intuition fails here also and performance is boosted when starting the set with the maximal total slowdown. The reason is that the slowdown metric is mostly effected by the shortest jobs and thus, a set with large total slowdown is likely to contain shorter jobs. By starting these jobs we comply with the shortest jobs first heuristic described in Chapter 2, which states that by starting

short jobs before other time consuming jobs, their response time and slowdown will be reduced, while the response time and slowdown of the longer jobs will not be severely effected and thus, the mean response and slowdown will be reduced. We named this the *Max-Slowdown* approach and describe it in Section 3.4.2. Simulation results for the *Max-Slowdown* approach are shown in Section 4.3.3.

3.4.1 Maximizing the Number of Started Jobs

The purpose of this experiment is to explore the effect of the number of started jobs in each scheduling step on the performance of LOS. In a case where both the *selected* and the *bypassed* markers are set in a given cell, following the path on which the maximal number of jobs will start, is expected to improve performance since in addition to maximizing the system utilization, fewer jobs will remain waiting and thus improvement is expected in the mean jobs response time and slowdown metrics. We refer to this as the $Maxjobs$ approach.

We enhance our three dimensional data structure which was described in Section 3.3.2 by including an additional integer value, num_jobs, in every $m'_{i,j,k}$ cell, in addition to the existing, *util*, *sutil*, and the two trace markers. The role of num_jobs is to record the number of jobs which will start when following a path through that cell. We also modified Algorithm 3 which constructs M', to consider the value of num_jobs. Whenever the same utilization value can be achieved, either by selecting or bypassing job i, the following rule is applied: If more jobs will start by bypassing job i - only the *bypassed* marker will remain set to force the bypassing of that job. On the other hand, if selecting or starting job i maximizes the number of started jobs, then leaving only the *selected* marker will force the starting of that job. This does not change any of the utilization values in any of the cells when compared to the values computed by the original algorithm, but it does limit the number of optional selections when the two trace markers are set, by eliminating one of the options — the one on which less jobs will start. Algorithm 6 formally describes this approach.

The final phase is to construct *S'* as described in Section 3.3.4. Since *M'* is only enhanced with a single integer value in each cell, constructing *S'* does *not* require any

modifications and Algorithm 4 (or 5) remains the same. The only difference is in the final result — this time the number of jobs in the resulting S' is maximized, that is $\mid S' \mid \rightarrow MAX$.

Simulation results presented in Section 4.3.2 show that the *Maxjobs* approach indeed improves LOS performance and thus our assumption was proven to be correct.

Algorithm 6 Constructing M' - The *Maxjobs* Approach

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

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

3.4.2 Maximizing the Total Slowdown

In this experiment we took the enhanced three dimensional data structure, *M'*, which was described in Section 3.4.1, and replaced the num_jobs counter in every each $m'_{i,j,k}$with a $tot\_slowdown$ accumulator. $tot\_slowdown$ records the accumulated slowdown values of jobs when following a path through that cell.

It is important to understand how the algorithm computes a jobs' slowdown. Slowdown was defined in Section 1.1 to be the ratio of the time it takes to run the job on a loaded system divided by the time it takes on dedicated system, formally $slowdown = \frac{response\,time}{running\,time}$. Since $response\ time = wait\ time + running\ time$ and the jobs' actual $running\ time$ is unknown at the time the slowdown is calculated, we use the user-estimated runtime for that job instead, since it is the best (and only) indication to how long that job will run. Thus for each considered job wj_i, its slowdown is computed as follows:

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

We also took the $Maxjobs$ algorithm as a basis and modified it to consider the value of $tot\_slowdown$ in the following matter: When the same utilization can be achieved either by selecting or bypassing job i, then if the total slowdown achieved by selecting job i is greater than the total slowdown achieved by bypassing that job, then only the *selected* marker remains set. On the other hand if the total slowdown is greater by bypassing job i then setting the *bypassed* marker will force job i to be bypassed. We named this the *Max-Slowdown* approach and formally describe it in Algorithm 7.

Simulation results presented in Section 4.3.3 have shown that LOS's performance is boosted when using this approach, compared to all other tested merit values.

Algorithm 7 Constructing M' - The *Max-Slowdown* Approach

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

for $k = 0$ to $extra$
  for $i = 1$ to $|WQ|$
    for $j = 1$ to n
      if $wj_i.size > j$ or $wj_i.ssize > k$
        $util \leftarrow {m'}_{i-1,j,k}.util$
        $sutil \leftarrow {m'}_{i-1,j,k}.sutil$
        $tot\_slowdown \leftarrow {m'}_{i-1,j,k}.tot\_slowdown$
        $selected \leftarrow False$
        $bypassed \leftarrow True$
      else
        $util' \leftarrow {m'}_{i-1,j-wj_i.size,k-wj_i.ssize}.util + wj_i.size$
        $sutil' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.sutil + wj_i.ssize$
        $tot\_slowdown' \leftarrow m'_{i-1,j-wj_i.size,k-wj_i.ssize}.tot\_slowdown +$
                $(t - wj_i.arrival + wj_i.time)/wj_i.time$
        if $util' > {m'}_{i-1,j,k}.util$ or
          ($util' = {m'}_{i-1,j,k}.util$ and $sutil' \leq {m'}_{i-1,j,k}.sutil$)
            $util \leftarrow util'$
            $sutil \leftarrow sutil'$
            if $util' > {m'}_{i-1,j,k}.util$ or $sutil' < {m'}_{i-1,j,k}.sutil$
                $tot\_slowdown \leftarrow tot\_slowdown'$
                $selected \leftarrow True$
                $bypassed \leftarrow False$
            else
                if $tot\_slowdown' < m'_{i-1,j,k}.tot\_slowdown$
                    $tot\_slowdown \leftarrow m'_{i-1,j,k}.tot\_slowdown$
                    $selected \leftarrow False$
                    $bypassed \leftarrow True$
                else
                    $tot\_slowdown \leftarrow tot\_slowdown'$
                    $selected \leftarrow True$
                    if $tot\_slowdown' = m'_{i-1,j,k}.tot\_slowdown$
                        $bypassed \leftarrow True$
                    else
                        $bypassed \leftarrow False$
        else
            $util \leftarrow {m'}_{i-1,j,k}.util$
            $sutil \leftarrow {m'}_{i-1,j,k}.sutil$
            $tot\_slowdown \leftarrow {m'}_{i-1,j,k}.tot\_slowdown$
            $selected \leftarrow False$
            $bypassed \leftarrow True$

3.5 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. Both n and $extra$ are bounded by N — the size of the machine, which is a constant. Since each ${m'}_{i,j,k}$ cell is computed in a constant time and there are maximum $|WQ| \times N \times N$ cells to compute, the time complexity of the algorithm for constructing M' and thus for producing the optimal schedule is:

$$(1) \quad O(|WQ| \times N \times N) = O(|WQ| \times N^2)$$

It is important to understand that the algorithm is *not* polynomial in the size of its input — the list of jobs sizes and in fact, there is an exponential relationship between the size of the input and the algorithm runtime. 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 $\log x$, and thus the length of encoding any of the waiting jobs' sizes is $\log wj_i.size$. If wj_l is the largest waiting job, than the size of encoding the entire input is:

$$(2) \quad O(|WQ| \times \log wj_l.size)$$

At this point we can use the fact that N is at-most the sum of all waiting jobs sizes, otherwise all jobs can be started and the solution becomes trivial. Since wj_l is the largest of all waiting jobs, we can safely state that $N \leq |WQ| \times wj_l.size$. Thus by substituting N in (1) we find that the time complexity of the algorithm is:

$$(3) \quad O(|WQ|^3 \times (wj_l.size)^2)$$

Since $\log wj_l.size$ in (2) and $wj_l.size$ in (3) hold an exponential relationship and not a polynomial one, it is clear that the time complexity is not polynomial in the size of the input alone. In fact, it is polynomial in the size of the input and the size of the largest waiting job.

Such algorithms which have their runtime bounded by a polynomial in the size in the input and the *value* of any integer in the input are known as *pseudo-polynomial* algorithms. 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 theoretical analysis. In our case, it is the restriction on N which allows the optimal schedule to be produced in a "reasonable" time, feasible for practical implementation.

3.5.1 Runtime Optimizations

As mentioned in Section 3.5, the construction of M' depends on three parameters: $|WQ|$ — the length of the waiting queue, n — the machine's free capacity at t, and $extra$ — the shadow free capacity. Since the values of these three parameters change from one scheduling step to the other, understanding the factors which effect each of the parameters is useful if one wishes to predict LOSs' runtime in upcoming scheduling steps.

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 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.

$|WQ|$ on the other hand, depends on the system load. On heavy loaded systems the mean waiting queue length can reach tens of jobs with peaks reaching sometimes hundreds — a fact that significantly increases the runtime of the algorithm. Two enhancements can be applied in the pre-processing phase. Both result in a shorter waiting queue $|WQ'| < |WQ|$ and thus improve LOS runtime 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 results.

The second 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 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 LOSs results is examined in Section 4.4.

♣ Looking at our initial waiting queue described in the table in Figure 3.4, 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 3.8.

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

Table 3.8: 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 3.5. By limiting the lookahead we improved the algorithm runtime and achieved the same results.

Chapter 4

Experimental Results

4.1 The Simulation Environment

We implemented all aspects of the algorithm including the optimizations mentioned in Section 3.5.1, 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 parallel supercomputers (See Appendix B - Workload Characteristics for details) 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,

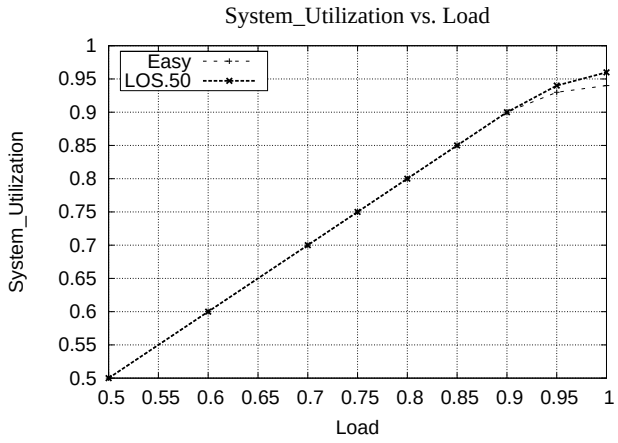
the simulator records its response time, bounded slowdown (applying a threshold of $\tau = 10$ seconds), and wait time.

4.2 Improvement over EASY

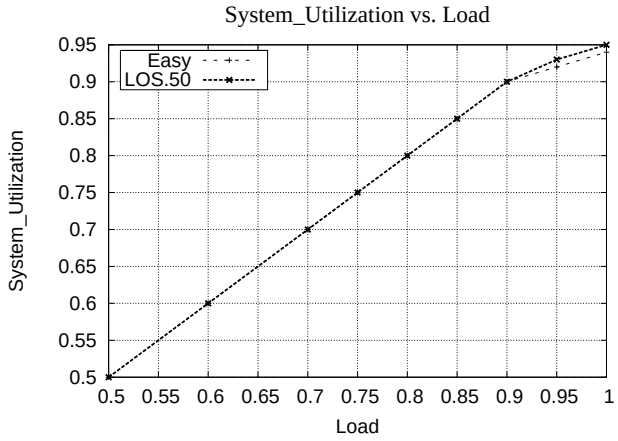
We used the framework mentioned above to run simulations of the EASY scheduler [24, 25], and compared its results to those of LOS which was limited to a maximal lookahead of 50 jobs. By comparing the achieved utilization vs. the offered load of each simulation, we saw that for the CTC and SDSC workloads (Figures 4.1(a,b)) a discrepancy occurs at loads higher than 0.9, whereas for the KTH workload (Figure 4.1(c)) 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.

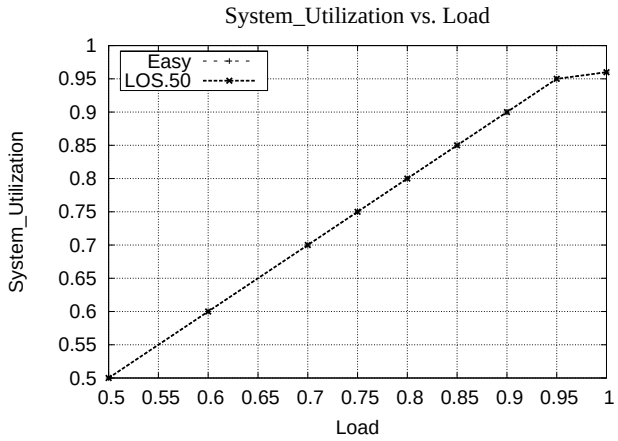
The results for response time are shown in Figure 4.2 , and for bounded slowdown in Figure 4.3. The results for wait time are the same as those for response time, because we are looking at differences. In all the plots, the mean job differential response time (or bounded slowdown) is positive across 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

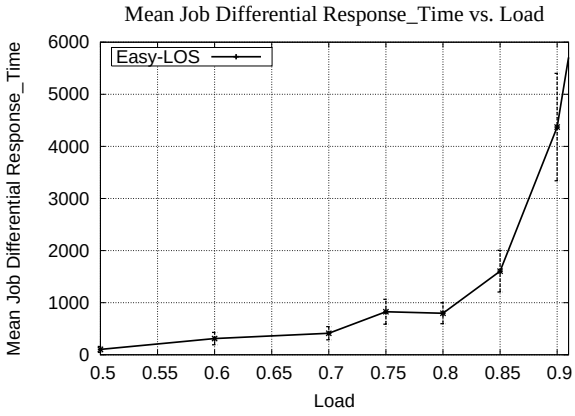


(b) SDSC Log

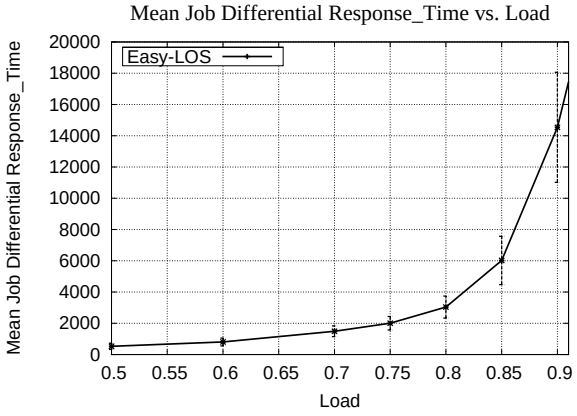


(c) KTH Log

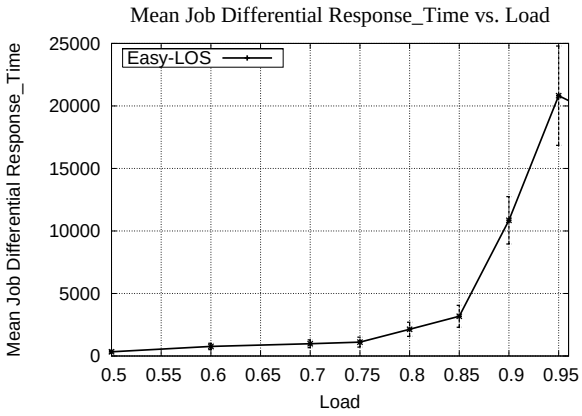
Figure 4.1: System Utilization vs. Load



(a) CTC log

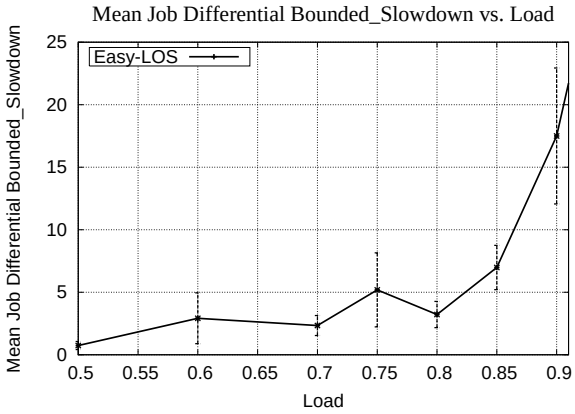


(b) SDSC log

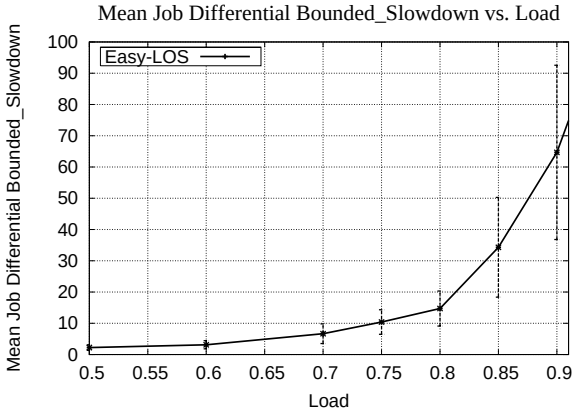


(c) KTH log

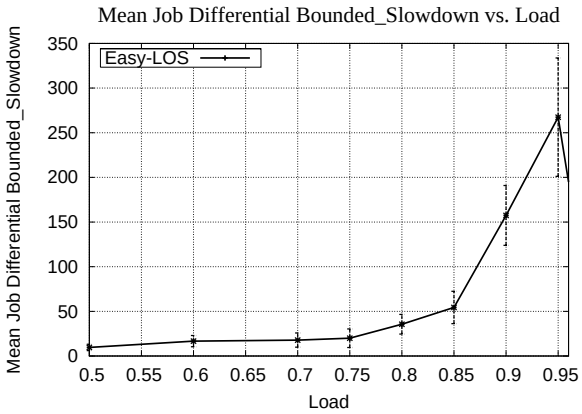
Figure 4.2: Mean job differential response time *vs* Load



(a) CTC log



(b) SDSC log



(c) KTH log

Figure 4.3: Mean job differential bounded slowdown *vs* Load

4.3 Job Selection Effect on Performance

4.3.1 Selecting Instead of Bypassing

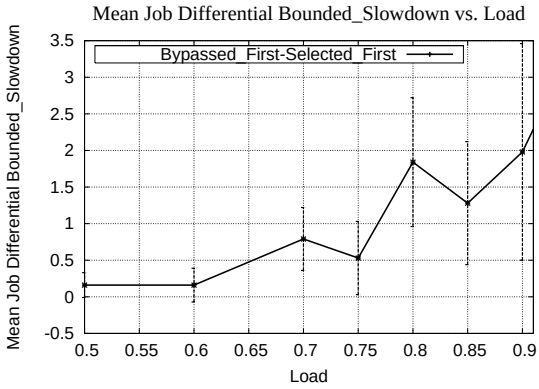
In Section 3.4 we introduced the *Selected-First* algorithm which is a modification of the original algorithm for constructing *S'*. Unlike the original, *Bypassed-First* algorithm which selects jobs closer to the head of the queue, the *Selected-First* algorithm favors jobs at the queue tail.

To compare the two algorithms we used the framework described in Section 4.1 and ran two simulation of LOS which was limited to a maximal lookahead of 50 jobs. The only difference between the two runs is that in the first, LOS used the unmodified *Bypassed-First* algorithm, and in the second, the *Selected-First* algorithm was used to construct *S'*.

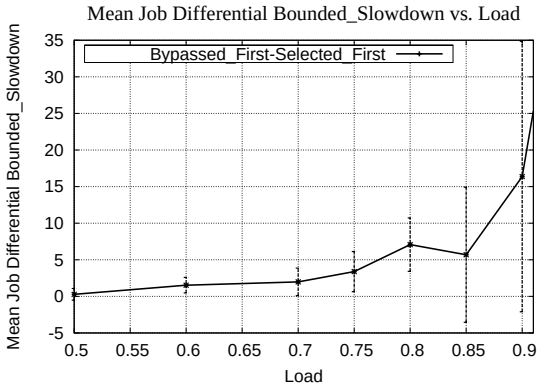
Following the explanation in Section 4.2, we applied the "common random numbers" variance reduction technique [29]. For each job in the workload file, we tabulate the *difference* between its bounded slowdown in the first and the second runs, computed confidence intervals on these differences and plotted the results. We decided to focus our analysis on the mean job bounded slowdown metric since it does not use absolute values (See Section 1.1 - The Goals of the Job Scheduler), and thus more accurately reflects the differences between the two algorithms.

The results are shown in Figure 4.4. We see that for all three workloads, the mean job bounded slowdown difference is positive across the entire load range — a clear indication that the *Selected-First* algorithm outperforms the original *Bypassed-First* with respect to this metric. On the other hand if we compare the resulting plots to those of Figure 4.3 where LOS was compared to EASY, we see that the curves here are significantly lower and in fact some of the lower foundries of the 90% confidence interval bars fall below the load axis. For example, the mean job differential bounded slowdown at 90% load for the CTC workload in Figure 4.4(a) is 2, while in Figure 4.3(a) it is about 18. For the SDSC workload in Figure 4.4(b) it is 16 while in Figure 4.3(b) it is 65 etc. The reason for the low curves is the fact that unlike Section 4.2 where we compared LOS to a conceptually different scheduling algorithm, we now compare two versions of the same scheduler, both which focus and achieve the exact same maximal utilization, but only differ in the set of

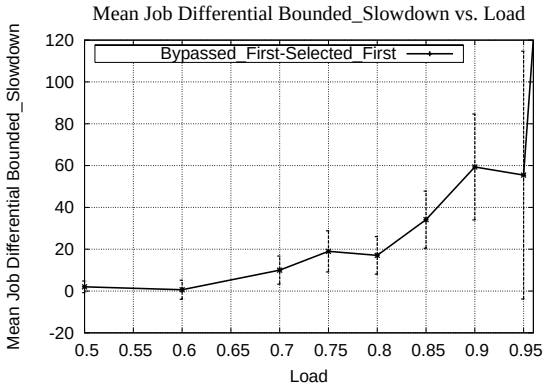
jobs which construct the final schedule. Therefore we can expect the performance gaps to be smaller, but still we see that choosing different sets jobs effect LOSs performance, a positive improvement in our case.



(a) CTC log



(b) SDSC log



(c) KTH log

Figure 4.4: Mean job differential bounded slowdown time *vs* Load - *Selected-First* algorithm

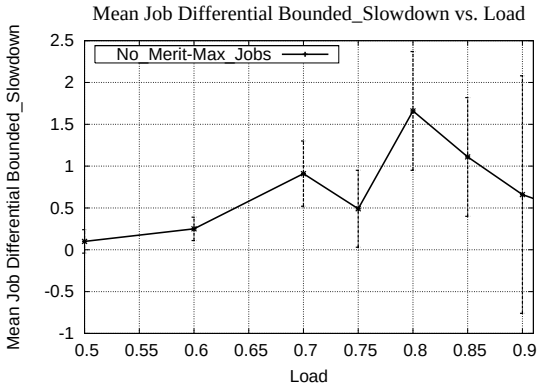
4.3.2 Maximizing the Number of Started Jobs

In Section 3.4.1 we introduced the *Maxjobs* approach. We stated that considering the number of jobs which will start, and selecting the path on which this number is maximized, is expected to improve LOS's performance since less jobs will remain waiting.

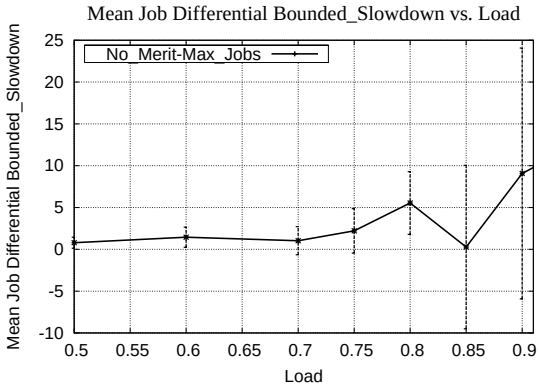
We followed the simulation paradigm of Section 4.3.1 and ran two simulations of LOS. In the first, LOS used the unmodified algorithm for constructing *M'* (Algorithm 3). In the second, this algorithm was replaced with Algorithm 6 which encapsulates the *Maxjobs* approach. Again, for each job we tabulated the *difference* between its bounded slowdown in the first and the second runs, computed confidence intervals on these differences and plotted the results.

The results are shown in Figure 4.5. The curve title "No_Merit - Max_Jobs" indicates that the differences are between the original algorithm where no merit computation was involved and the *Maxjobs* algorithm which starts the maximal number of jobs.

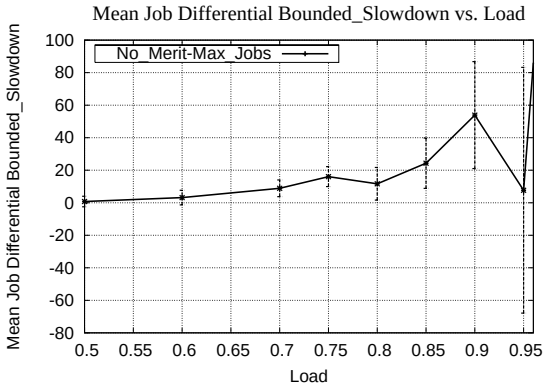
The fact that for all three workloads and for the entire load range, the mean job differential bounded slowdown remain positive indicates that the *Maxjobs* algorithm outperforms the original algorithm for constructing *M',* and since the time complexity of both algorithms is identical, it is the preferable choice in view of its results.



(a) CTC log



(b) SDSC log



(c) KTH log

Figure 4.5: Mean job differential bounded slowdown time *vs* Load - *Maxjobs* approach

4.3.3 Maximizing the Total Slowdown

In Section 3.4.1 we introduced the *Max-Slowdown* approach in which the set *S'* is chosen in a way that its overall total slowdown is maximized. We followed the paradigm of Sections 4.3.2 and 4.3.1, and run two simulations of LOS, one with the unmodified algorithm for constructing *M'* and the second with the modified Algorithm 7 which considers the jobs slowdown.

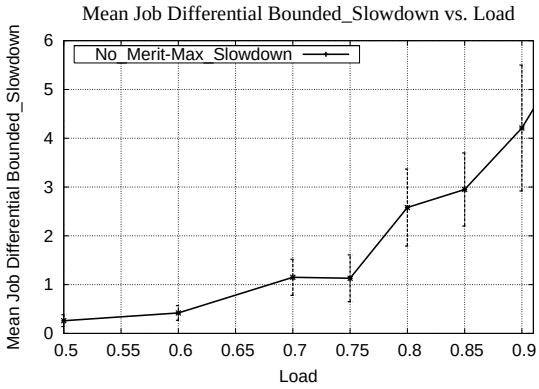
The results in Figure 4.6 show that LOS performance is boosted when using the *Max-Slowdown* approach compared to the results achieved when the original algorithm for constructing *M'* was used. In addition, the results far exceeds those of the *Maxjobs* approach in Figure 4.5 and the *selected-first* algorithm in Figure 4.4.

Just for comparison, the maximal differential bounded slowdown for the KTH workload in Figure 4.5(c) is 50, in Figure 4.4(c) it is 60 while for the *Max-Slowdown* in Figure 4.6 it is 90. Similar observation hold for the CTC and SDSC workloads.

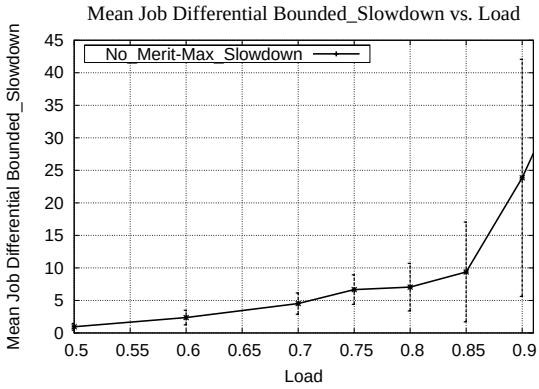
To complete the performance evaluation we compared LOS when using the *Max-Slowdown* approach, to the EASY scheduler. We followed the simulation paradigm of Section 4.2 and plotted the mean job differential bounded slowdown curve in Figure 4.7. We then compared the results to Figure 4.3, where the unmodified algorithm for constructing *M'* was used. As can be seen, for all three workloads and for the entire load range, the mean job differential bounded slowdown curves in Figure 4.7 are higher than the curves in Figure 4.3. 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 means the *Max-Slowdown* approach had further reduced the jobs slowdown and thus it outperforms the original algorithm for constructing *M'*.

It is also interesting to see how does the *Max-Slowdown* approach effect other metrics such as jobs' response time. We plotted the mean job differential response time under EASY and LOS in Figure 4.8 and compared the resulting curves to Figure 4.2. For all three workloads the curves of the *Max-Slowdown* approach are higher than those of the unmodified algorithm for constructing *M'* which means that this approach outperforms the original algorithm with respect to other metrics as well. As can be seen in Subfigures 4.8(b) and 4.8(c), in 90% load (95% in KTH) there is a slight advantage for the

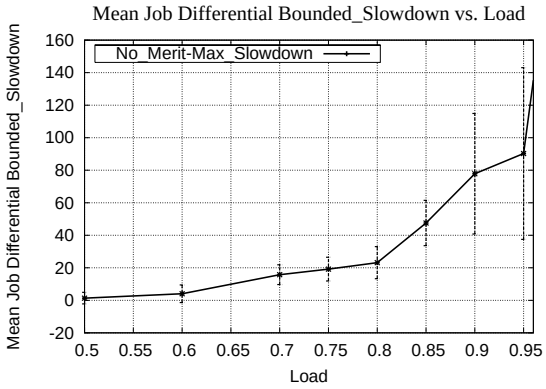
unmodified algorithm. This does not mean the *Max-Slowdown* has failed to perform and in fact a positive mean response difference of 14000 (20000 in KTH) is a major improvement over EASY. What this means is that on extremely high loads when the machine almost saturates, a change in the heuristic may be considered if the scheduler target is to minimize the response time of the jobs.



(a) CTC log

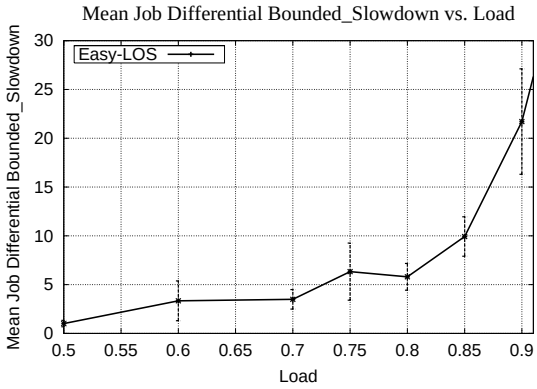


(b) SDSC log

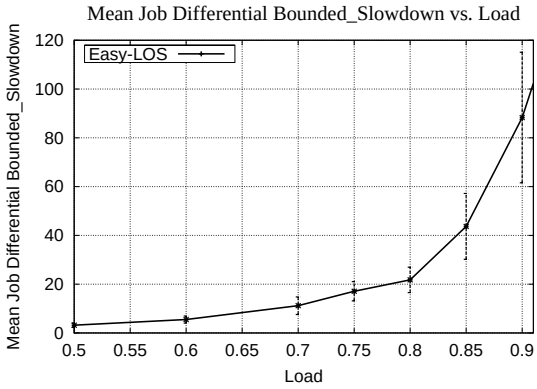


(c) KTH log

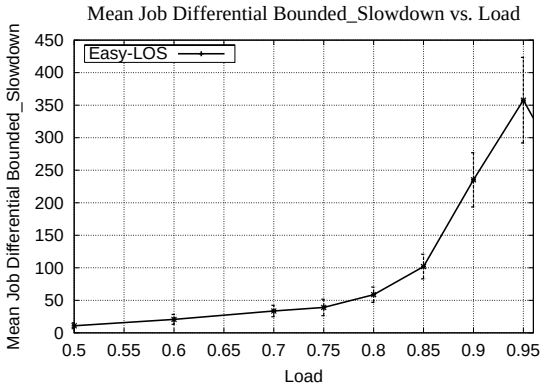
Figure 4.6: Mean job differential bounded slowdown time *vs* Load - *Max-Slowdown* approach



(a) CTC log

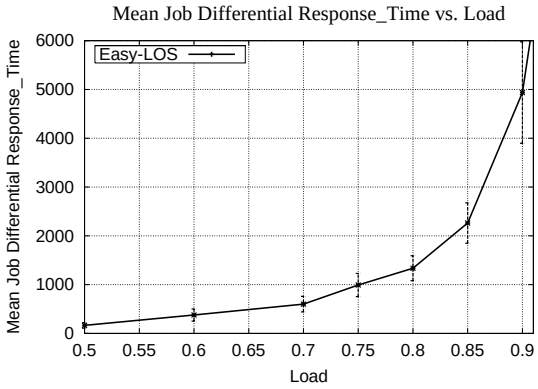


(b) SDSC log

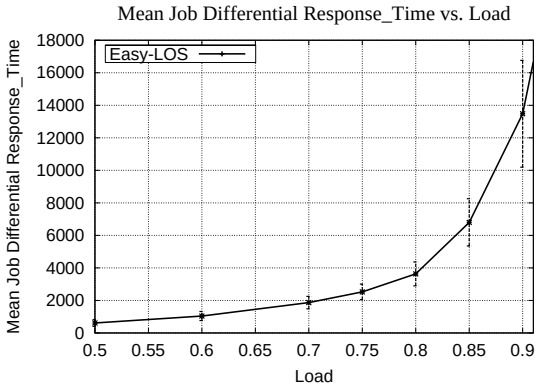


(c) KTH log

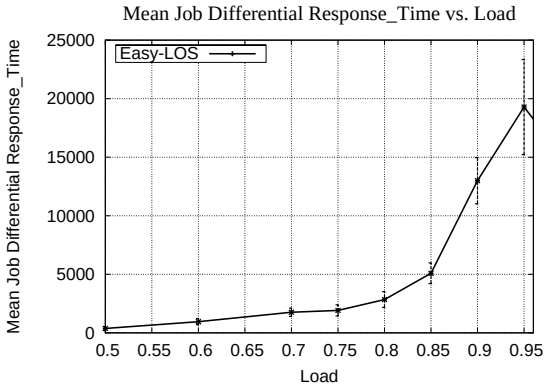
Figure 4.7: Mean job differential bounded slowdown *vs* Load - EASY - LOS, *Max-Slowdown* approach



(a) CTC log



(b) SDSC log



(c) KTH log

Figure 4.8: Mean job differential response time *vs* Load - EASY - LOS, *Max-Slowdown* approach

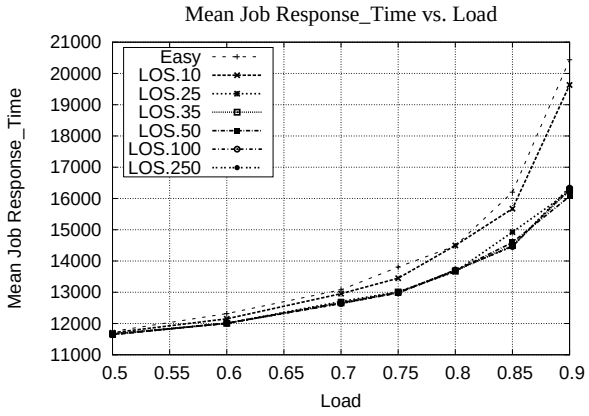
4.4 Limiting the Lookahead

Subsection 3.5.1 proposed an enhancement called *limited lookahead* aimed at improving the runtime performance 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. Figure 4.9 presents the effect of the limited lookahead on the mean job response time. Figures 4.10 and 4.11 present its effect on the mean job bounded slowdown and mean job wait time respectively.

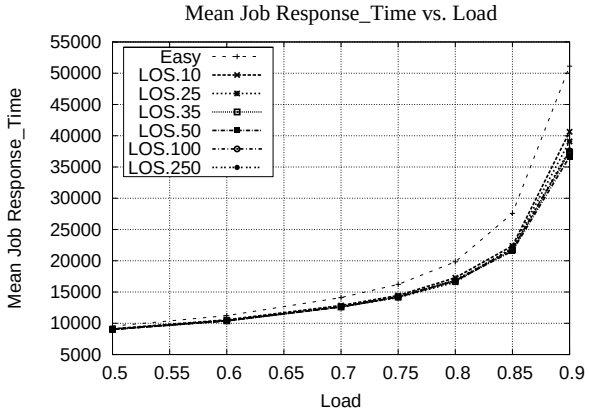
The notation *LOS.X* is used to represent LOS's result curve, where X is the maximal number of waiting jobs that LOS was allowed to examine on each scheduling step (i.e. its lookahead limitation). We also plotted EASY's result curve to allow a comparison. We observe that for the CTC log in Figure 4.9(a) and the KTH log in Figure 4.9(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 Figures 4.10(a,c) and to the mean job wait time as shown in Figures 4.11(a,c). As most clearly illustrated in figures 4.9(a), 4.10(a) and 4.11(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 4.9(b), 4.10(b) and 4.11(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 all three metrics are relatively similar for lookahead in the range of 25–250 jobs. Thus we can safely use a bound of 50 on the lookahead, thus bounding the complexity of the algorithm.

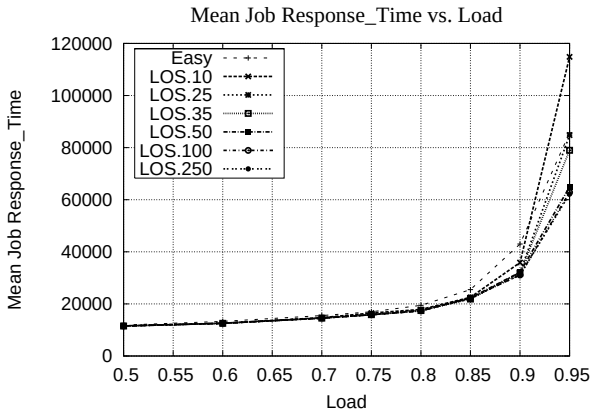
The explanation is that at most of the scheduling steps, especially under low loads, the length of the waiting queue is kept small, so a lookahead of hundreds of jobs has no effect



(a) CTC log



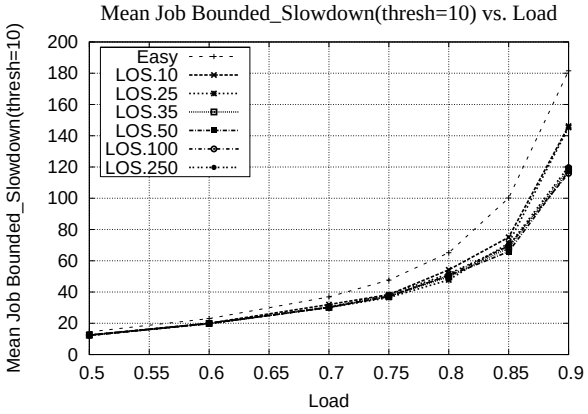
(b) SDSC log



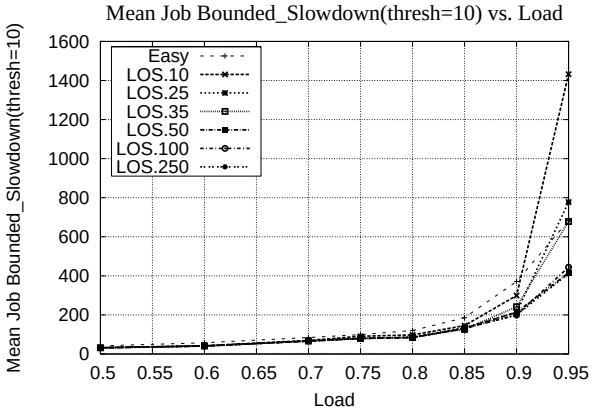
(c) KTH log

Figure 4.9: Limited lookahead affect on mean job response time

(a) CTC log

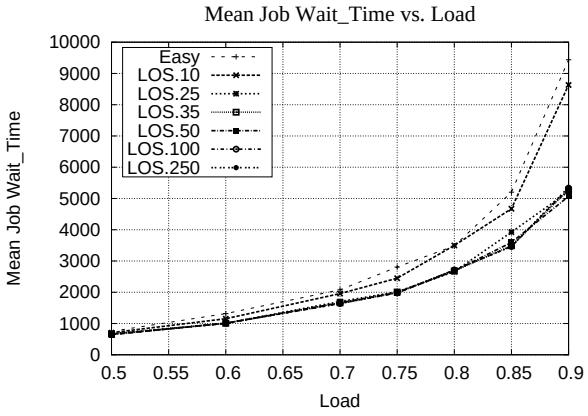


(b) SDSC log

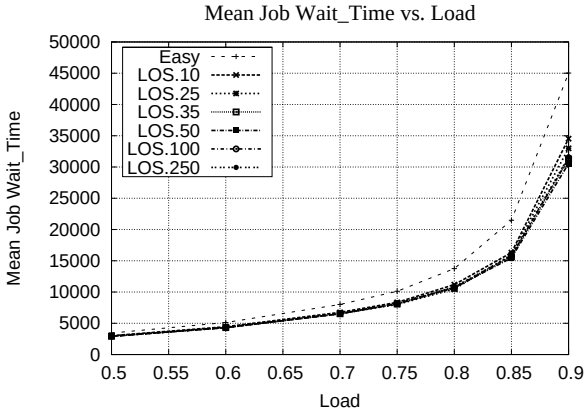


(c) KTH log

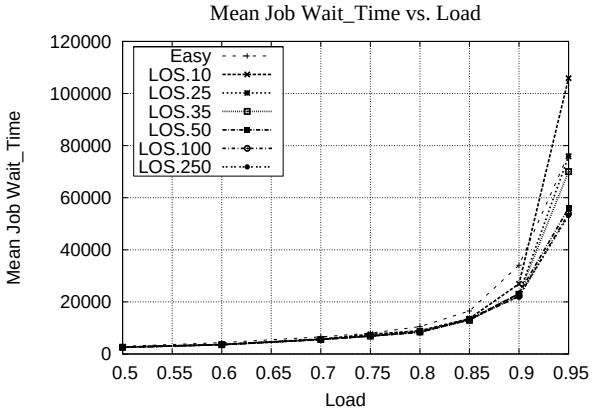
Figure 4.10: Limited lookahead affect on mean job bounded slowdown



(a) CTC log



(b) SDSC log



(c) KTH log

Figure 4.11: Limited lookahead affect on mean job wait time

in practice. As the load increases and the machine advances toward its saturation point, the number of waiting jobs increases, and the effect of changing the lookahead is more clearly seen. Figure 4.12 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 the results. First, with LOS, the mean queue length is actually shorter compared to EASY, and the reason is its efficiency in packing jobs, which allows them to terminate faster. The second observation is that only for the KTH log in Sub-figure 4.12(c), the mean number of waiting jobs exceeds the lookahead limitation of 50 jobs, and this happens only at loads higher than 90%. The problem with Figure 4.12 is that the mean queue length provides only a brief summary to what happened over the entire simulation and that time-dependent information such as peaks in which the queue length can reach hundreds of jobs are absorbed in the mean calculation.

A detailed time-dependent analysis of the queue behavior is presented in Section 4.5 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 manages to keep the queue length well below 50 jobs at times when it reaches a length of hundreds under EASY.

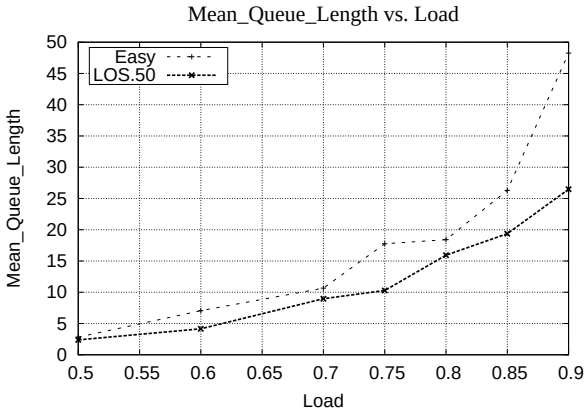
4.5 The LOS Scheduler and Users Satisfaction

In Section 3.5.1 we stated that on a heavily loaded system the waiting queue length can reach tens of jobs, so a scheduler capable of maintaining a shorter queue across a large fraction of the scheduling steps, increases the users' satisfaction with the system.

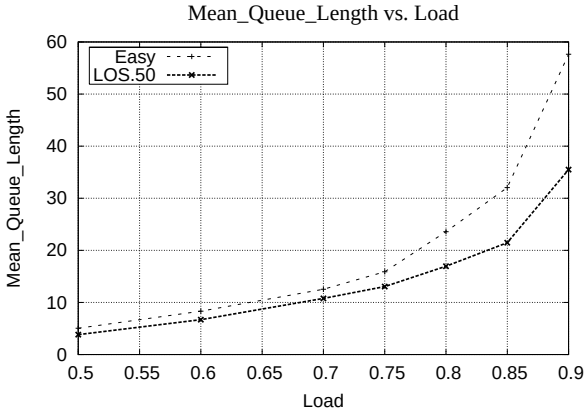
It is interesting to examine if and how LOS manages to maintain a shorter waiting queue when compared to EASY. It is also interesting to examine the queue behavior when the *Max-Slowdown* approach is used by LOS.

To explore the queue length behavior, we instrumented the simulator with a waiting-queue length counter and recorded its changing value on every scheduling step (i.e arrival or termination of jobs) across the entire simulation.

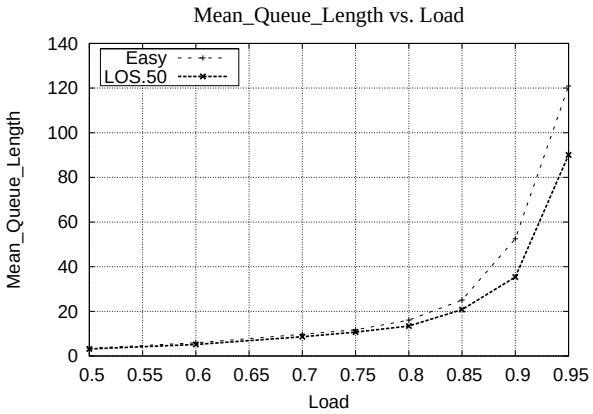
As such instrumentation produces large amounts of data, we enabled it only for the two most high offered loads on which the simulation is still stable, that is, for the SDSC



(a) CTC log



(b) SDSC log



(c) KTH log

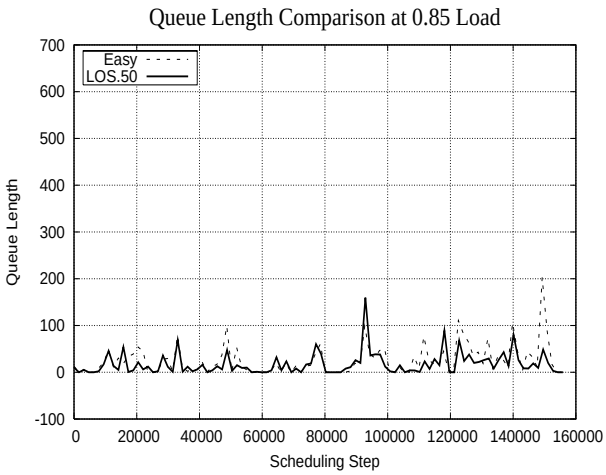
Figure 4.12: Mean queue length *vs* Load

and CTC workloads, counter recording has been enabled for 0.85 and 0.90 loads and for the KTH workload results were recorded for 0.90 and 0.95 loads.

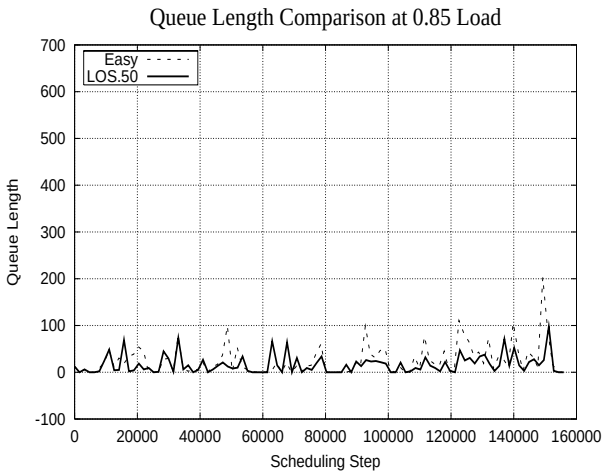
A detailed comparison of the queue length behavior of the EASY scheduler and LOS which was limited to a lookahead of 50 jobs is presented in Figures 4.13 to 4.15. Figure 4.13 presents the simulation results for the CTC workload. Figures 4.14 and 4.15 presents the results for the SDSC and KTH respectively. In all figures, the curves in Sub-figures (a) and (c) are those of the unmodified algorithm and in Sub-figures (b) and (d) are for the *Max-Slowdown* algorithm.

As clearly seen in all figures, LOS maintains a shorter queue compared to EASY across a dominating portion of the scheduling steps. The reason is LOS's efficiency in packing jobs. As jobs are packed in an optimal manner, more processors are utilized and thus more computation is performed. As a result more jobs will terminate which allows waiting jobs to start, thus the waiting queue length is reduced.

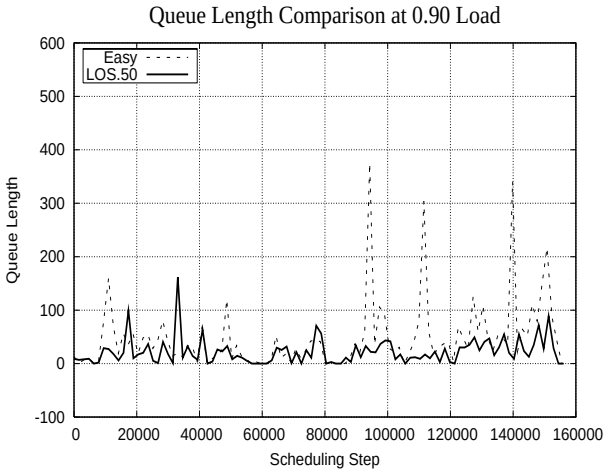
Following the results analysis we can safely state that in addition to improving specific metrics such as response time or slowdown , LOS will also increase users satisfaction when compared to the EASY scheduler.



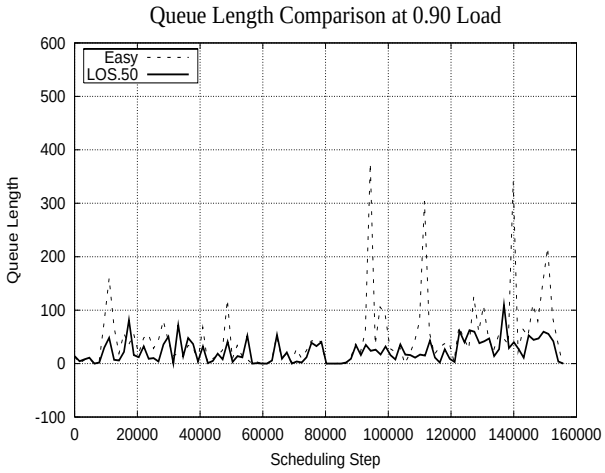
(a) Unmodified Algorithm



(b) The Max-Slowdown Approach

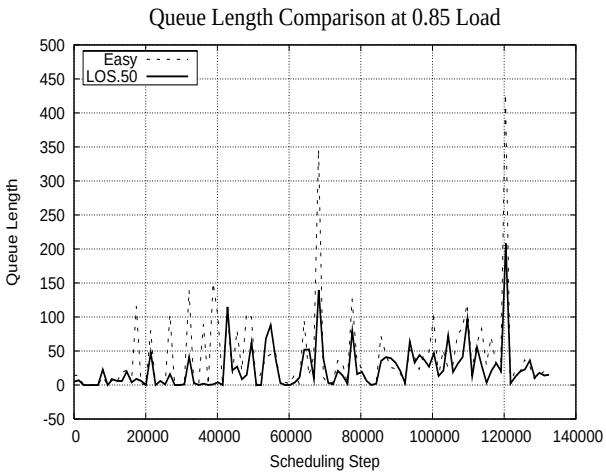


(c) Unmodified Algorithm

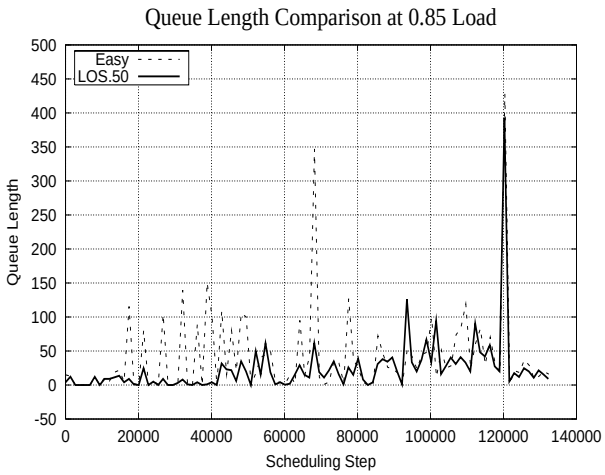


(d) The Max-Slowdown Approach

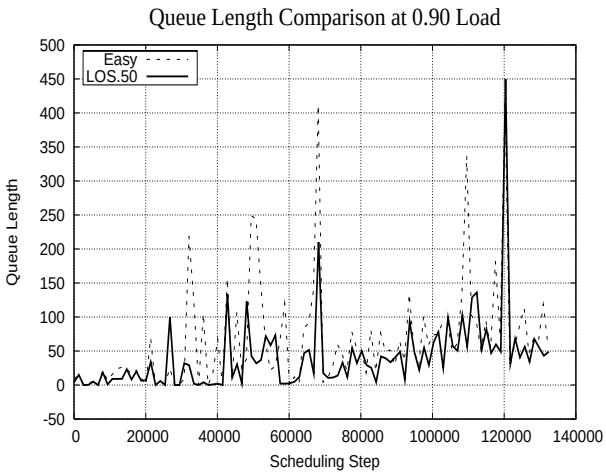
Figure 4.13: Queue Length Behavior Comparison- CTC log



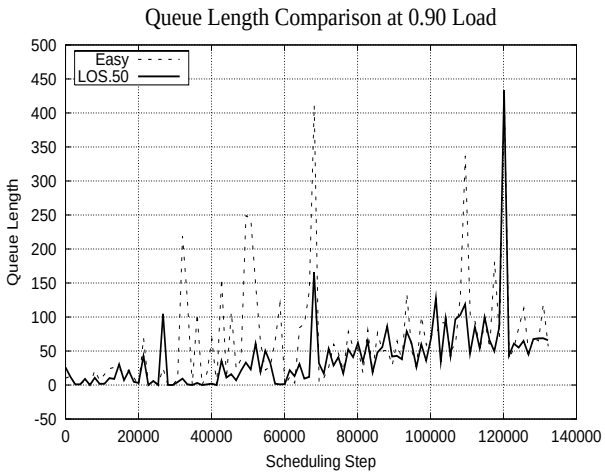
(a) Unmodefied Algorithm



(b) The Max-Slowdown Approach

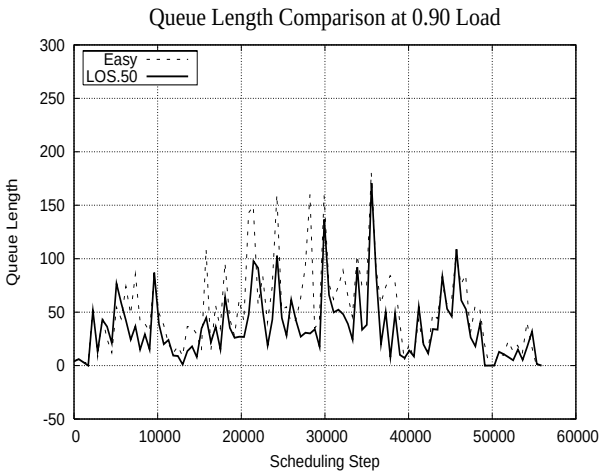


(c) Unmodified Algorithm

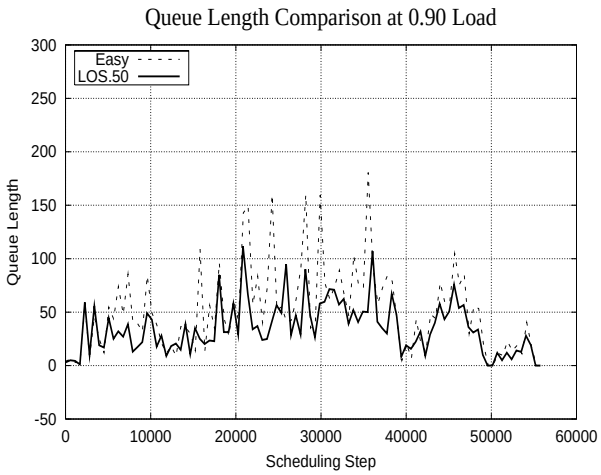


(d) The Max-Slowdown Approach

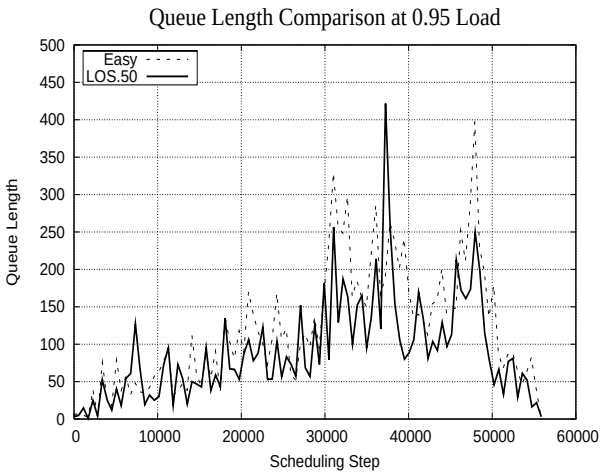
Figure 4.14: Queue Length Behavior Comparison - SDSC log



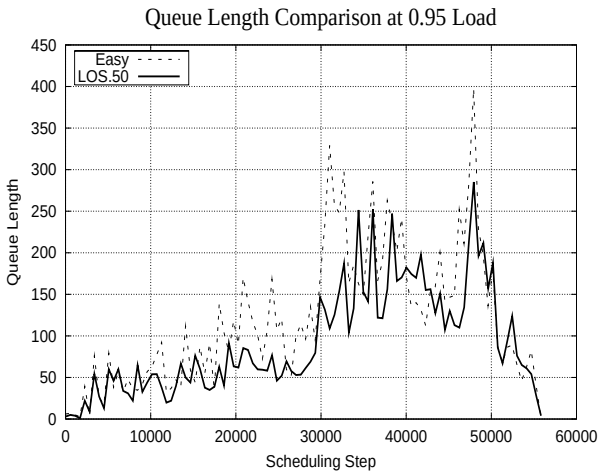
(a) Unmodified Algorithm



(b) The Max-Slowdown approach



(c) Unmodified Algorithm



(d) The Max-Slowdown Approach

Figure 4.15: Queue Length Behavior Comparison - KTH log

Chapter 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 maintained shorter across large fraction of the scheduling steps and this increases the users' satisfaction with the system.

There is often more than a single way to pack jobs and achieve the same utilization values. We explored the various alternatives by including merit calculation in the lookahead process and choosing the set of jobs which maximizes or minimizes the merit value. We show that performance can improve or reduce with respect to the chosen merit. Surprisingly, performance is boosted when choosing the set of jobs, S', with the *maximal* total slowdown. The reason is the nature of the slowdown metric which is mostly effected by the shorter jobs and thus, a set with a large total slowdown is likely to contain the shortest jobs. By starting these jobs ahead of longer ones, the mean job response time is reduced and performance much improve.

A future study should further explore the various ways to optimize the algorithm runtime. In Section 3.5.1 we suggested two enhancements which result in a shorter waiting queue and showed that runtime performance can improve without effect on the results. Calculating the utilization in an on-going fashion and stopping the construction of M' when utilization reaches a certain threshold is another improvement which we suggest as a study case. In addition, 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.

Bibliography

[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 BlueGene/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, *"Selective Reservation Strategies for Backfill Job Scheduling"*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson, L. Rudolph and U. Schwiegelshohn, Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2537, pp. 55-71, 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.

Appendix A

Backfilling Algorithms

A.1 The EASY Backfilling Algorithm

The EASY backfilling algorithm is executed repeatedly whenever a new job arrives or a running job terminates, if the first job in the queue cannot start. In each iteration, the algorithm identifies a job that can backfill if one exists.

Algorithm 8 EASY Backfill

1. Find the shadow time and extra nodes:
   (a) Sort the list of running jobs according to their expected termination time.
   (b) Loop over the list and collect nodes until the number of available nodes is sufficient for the first job in the queue.
   (c) The time at which this happens is the shadow time.
   (d) If, at this time, more nodes are available than needed by the first queued job, the ones left over are the extra nodes.
2. Find a backfill job:
   (a) Loop on the list of queued jobs in order of arrival.
   (b) For each one, check whether either of the following conditions hold:
      i. It requires no more than the currently free nodes and will terminate by the shadow time, or
      ii. It requires no more than the minimum of the currently free nodes and the extra nodes.
   (c) The first such job can be used for backfilling.

A.2 The Conservative Backfilling Algorithm

Conservative backfilling maintains two data structures. One is the list of queued jobs and the time at which they are expected to start execution. The other is a profile of the expected processor usage at future times. The algorithm is executed whenever a new job arrives.

Algorithm 9 Conservative Backfill

1. Find anchor point:
   (a) Scan the profile and find the first point where enough processors are available to run this job. This is called the anchor point.
   (b) Starting from this point, continue scanning the profile to ascertain that the processors remain available until the job's expected termination.
   (c) If not, return to (a) and continue the scan to find the next possible anchor point.
2. Update the profile to reflect the allocation of processors to this job, starting from its anchor point.
3. If the job's anchor is the current time, start it immediately.

Appendix B

Workloads Characteristics

B.1 The CTC Workload

The Cornell Theory Center (CTC) log contains 79302 job records submitted to a 512 nodes IBM SP2 System. Log recording started at Wednesday, 26 Jun 96, 16:06:00 and ended at Saturday, 31 May 97, 22:11:26.

Information :

1. http://www.tc.cornell.edu
2. http://www.cs.huji.ac.il/labs/parallel/workload

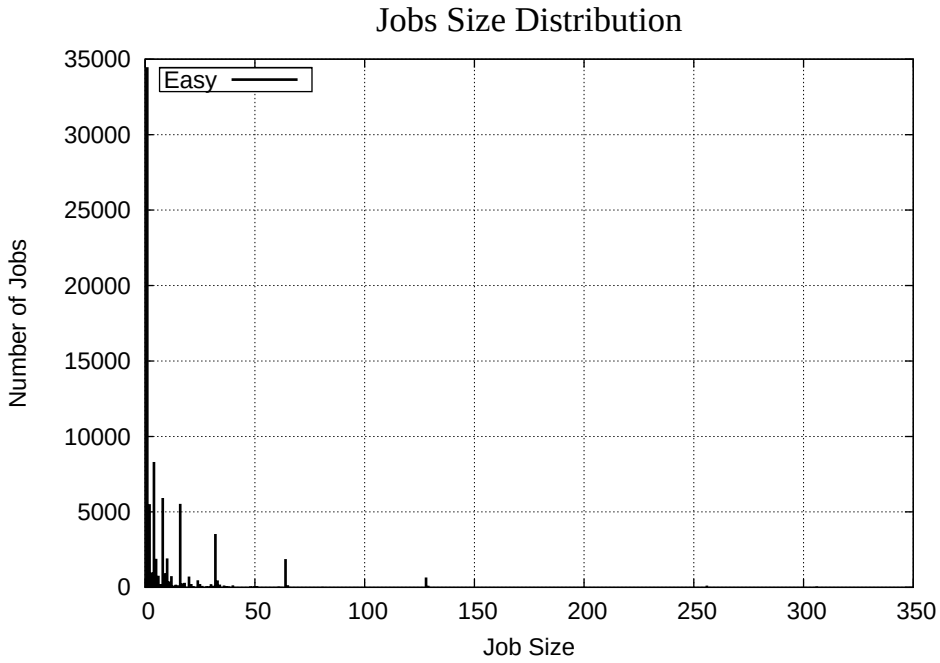


Figure B.1: Jobs size distribution - CTC log

B.2 The SDSC Workload

The San Diego Supercomputer Center (SDSC) log contains 67667 job records submitted to a 128 nodes IBM SP2 System. Log recording started at Wednesday, 29 Apr 98, 16:05:28, 16:06:00 and ended at Sunday, 30 Apr 00, 04:08:32.

Information :

1. http://joblog.npaci.edu
2. http://www.cs.huji.ac.il/labs/parallel/workload

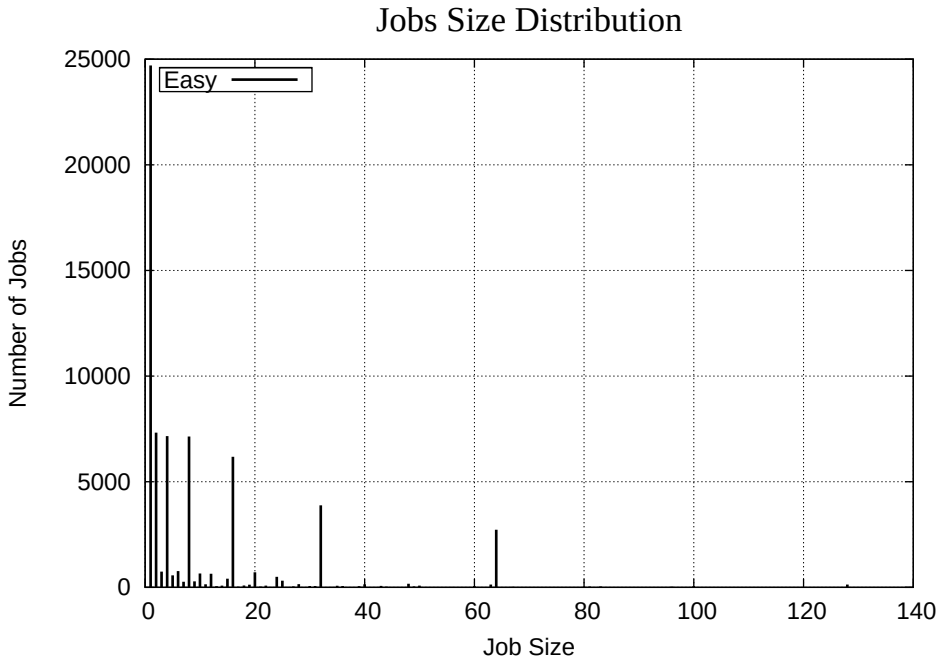


Figure B.2: Jobs size distribution - SDSC log

B.3 The KTH Workload

The Swedish Royal Institute of Technology (KTH) log contains 28490 job records submitted to a 100 nodes IBM SP2 System. Log recording started at Monday, 23 Sep 96, 12:00:31 and ended at Friday, 29 Aug 97, 08:55:01.

Information:

1. http://www.pdc.kth.se
2. http://www.cs.huji.ac.il/labs/parallel/workload

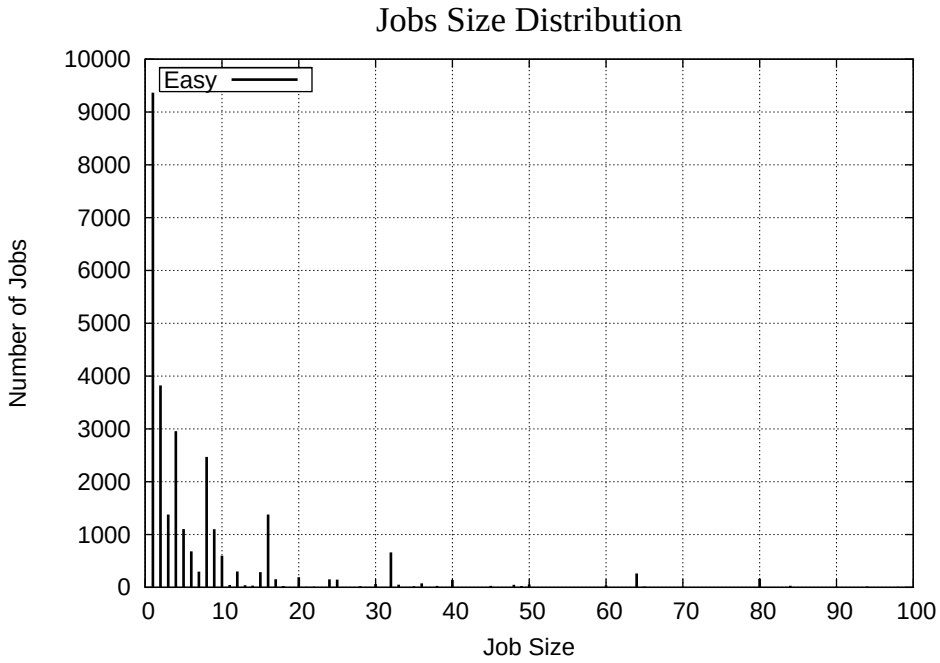


Figure B.3: Jobs size distribution - KTH log