2009

Backfilling with fairness and slack for parallel job scheduling

Wei Jin
University of Windsor

Backfilling with Fairness and Slack for Parallel Job Scheduling

By

Wei Jin

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2009

Your file Votre référence
ISBN: 978-0-494-57637-3
Our file Notre référence
ISBN: 978-0-494-57637-3

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Parallel jobs have different runtimes and numbers of threads/processes. Thus, scheduling parallel jobs involves a packing problem. If jobs are packed as tightly as possible, utilization will be improved. Otherwise, some resources have to stay idle. The common solution to deal with idle resources is backfilling, which schedule smaller jobs submitted later to execute earlier as long as they do not postpone the first job or all the previous jobs in the waiting queue. Traditionally, backfilling uses first fit for idle resources, according to the submission order. However, in this case, better packing of jobs could be missed. Hence, we propose an algorithm which looks further ahead if significantly improving utilization. However at the same time, this could be unfair to some jobs ahead in the queue. So we use a delay factor as a constraint to limit unfairness. We propose a branch and bound algorithm which selects jobs for backfilling which keep utilization high, while trying to stay close to First-Come-First-Served (FCFS). We evaluate relative response time and utilization and compare to other backfilling approaches. The selection of jobs for backfilling to optimize for high utilization and low delay is implemented as an extension of the existing Scojo-PECT preemptive scheduler.

To my mother, Jianhua Zhang

my father, Guoshun Jin

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my supervisor Dr. Angela C. Sodan. Without her extensive guidance and constant encouragement, I could not finish this thesis. I would also like to thank all committee members, Dr. Wai Ling Yee, Dr. Ahmed Tawfik and Dr. Peter Tsin for their valuable time and comments.

My special thanks go to my parents and my brother. Their love and care have always been with me during these years. Their trust and encouragement pulled me through hard times.

I would like to thank Xiejie Zeng, and Xiaorong Cao for the happy experience during the time I worked with them. And especially the help they gave to me during these years.

Last, but not least, I would like to thank all of my friends for all the help and support during the completion of this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In High Performance Computing (HPC) field, job scheduling is an important issue. It acts like an inter medium with one side is processors and the other side is jobs. Its role is to decide certain jobs running in certain nodes during a specific time. In this context, each jobs is referred as an application which may be composed by one process or multiple processes.

In job scheduling, there are two basic types of approaches which are time sharing and space sharing. With time sharing, processors are shared by multiple processes with time slices, so jobs are running on the same processors. In this case, jobs will be suspended even though they are not finished at the end of time slice. And new jobs are switched to these processors in the following slice time and run until the end of their time slice. Hence the most obvious advantage is that later submitted jobs could start earlier. The reason is that these later submitted jobs do not have to wait until all the previous ones finished, and they will start to run only at their own time slices. However, time sharing suffers high overload from switching processes. Another issue in time sharing is coordination which means that processes of the same job communicate with each other by sending or receiving messages. In this case, any processors can be chosen to run any processes. Their relationship is independent. Hence, when communicating among processes, there are chances that some processes of a job would have to wait for this job's other processes in different processors if these processes cannot run at the same time. With space sharing, processors are assigned exclusively to certain processes and until these processes finish running. This feature makes jobs unsuspended. Therefore, compared with time sharing, space sharing does not have overhead for switching jobs. Because of no switching, space sharing is also much easier to implement, but unfortunately, the disadvantage of space sharing is obviously that some processors are staying idle during some time periods because of imperfect packing of jobs. These idle resources sometimes show like fragments. In order to improve the performance, the size of fragment has to be decreased, and this problem is like solving bin packing problem [30] [31] [32] to some extent. A lot of scheduling approaches have been proposed to reduce the size of fragment as much as possible, such as implicit or dynamic coscheduling to time sharing, and different ways to move later smaller jobs ahead to run in these fragment areas in order to pack jobs as tightly as possible and to some extent improve the utilization of resources.

In order to evaluate the performance of a scheduling approach, there are some well-known metrics. From the perspective of system performance one metric is utilization which is usage of resources. The utilization is equal to the ratio of used resources to total resources during a period of time. Therefore, the fewer resources are idle, the higher utilization is and the better performance is. Another metric is response time which is based on the benefits of users, because it shows how fast the system finishes scheduling users' submission jobs. Response time is related to some factors which are the time a job has to wait before it starts and the time a job finishes running. Hence, it is a time period between the job's submission time and the job's finishing executing time. However, usually average response time of all jobs is calculated instead of each job's response time. Because in some situations, single user's request is treated badly, but the overall users' requests are treated very well. In addition, there is another metric, fairness, which is used to evaluate each individual job. For each job, it may be treated preferentially which means it is scheduled to start earlier than supposed to be. And also it may be treated unfairly which is just the opposite of the former. However, in fact all these metrics work contradictorily. If getting higher utilization, the fairness may become worse and average response time may increase. If getting better fairness or lower average response time, the utilization may decrease. Therefore, scheduling approaches take some metrics as main goal and keep other metrics as constraint, e.g. in the meantime of getting higher utilization, there is a slack (Chapter 4.3.1) defined to set as a threshold so that every single job's fairness is under control.

Backfilling [1] is a common solution to deal with the idle resources, which schedule smaller later submitted jobs to execute earlier to improve the resources utilization. Also it has been implemented in several production schedulers [15]. According to different rules, there are different backfilling approaches. The first one is to move later smaller jobs ahead as long as they do not postpone the first job, which is called Easy (Aggressive) Backfilling [14][1]. The other one called Conservative Backfilling [14][1] is much more constraint which only moves later smaller jobs ahead if they do not delay all the previous jobs in the waiting queue. Traditionally, backfilling uses first fit jobs for idle resources, according to some heuristics such as the submission order, or jobs' response time [17] [6]. However, in this case, better packing of jobs could be missed. Hence, we propose an algorithm which looks further ahead and do conservative backfill with slack on those jobs instead of early ones if they can significantly improve utilization. This algorithm of selecting jobs is based on branch and bound algorithm and the jobs are chosen should keep utilization high, while trying to stay close to FCFS [16]. Heuristics are used to simplify the NP-Hard problem to cut the tree's branch. However at the same time, this could be unfair to some jobs ahead in the queue and preferential to some jobs later in the queue. Hence we use a delay factor as a constraint to limit unfairness. In this situation, instead of not delaying any previous jobs when backfilling later smaller submitted jobs, the algorithm we apply allows and defines different slack to each individual job at different time so that every job can be delayed by its slack. Besides, average unfairness for both treated unfairly and preferentially jobs are calculated to compare with FCFS. Adding slack to the jobs makes conservative backfilling more flexible, and can increase the chances of backfilling more jobs.

The rest of the thesis is organized as follows. At first, in Chapter 2, background issues are discussed. Then the specification of the problem and what can be improved in Chapter 3. And the algorithm is described in details in Chapter 4. In Chapter 5, the simulation, experiments and results analysis are presented. At last, there are conclusion of the thesis in Chapter 6 and some future work about this thesis in Chapter 7.

Chapter 2: Background and Related Work

In some former work, a relaxed backfilling is used by applying a slack factor on the wait time of the highest-priority job which is always the first one in the waiting queue [11]. Priority is calculated by a number of parameters like wait time, runtime, size and importance of jobs. The approach in [11] employs EASY backfilling. Benefits in evaluation are only shown when the slack factor is very high equaling 2, which means slack is twice the wait time. A more general idea of slack was introduced in [10]. Each job is assigned a slack which is based on (dynamically changing) priority, average wait times, and a static delay-tolerance factor.

Most backfilling approaches make one-by-one decisions and use first-fit, i.e. select the first job from the waiting queue which can be moved ahead without violating the backfill guarantees. The scheduler in [10] proposes several heuristics to re-order the waiting queue within the ranges of determined slack per job. The lookahead scheduling in [7], employed in the context of EASY backfilling, applies a more global perspective and optimizes which group of jobs is best to backfill under the metric of momentary increase of utilization. If utilization is the same for different job combinations, a second metric is applied for which maximum slowdown of the selected jobs evaluated best. Looking ahead 50 jobs was found to deliver almost equally good results as looking at all jobs in the waiting queue. For a system load (overall utilization) of 85%, average relative response times are improved by about 50%.

Similar effects as relaxation can be obtained from overestimation. Overestimation leads to larger windows for backfilling smaller/shorter jobs. In [13], overestimates are investigated which are either a static factor or random (up to twice the factor). Up to 25% improvements were obtained if the overestimations were random but only if the lengthening had a very wide range (up to a factor of 20). The scheduling policy used was shortest-job-first.

Whether EASY or conservative backfilling performs better was found to largely depend on the workload and on the evaluation metric used (average response times or average relative response times) though, more often, EASY performs slightly better [5]. Investigations per job type (under basically FCFS scheduling) [9] found that long-narrow jobs benefit from EASY backfilling, whereas short-wide jobs do better with conservative backfilling, and short-narrow and long-wide jobs do equally well under both schemes. The differences become more significant under high load. Combined scheduling which permits a certain factor of delays (either using the overall running average or job-type-specific running averages) for relative response times with EASY scheduling and schedules jobs conservatively if exceeding their corresponding relative delay bound. With job-type-specific delays factors, improvements were obtained for all job types (even for long-wide jobs) and for all jobs of about 40%, while also lowering worst cases.

A consideration closely related to backfilling is fairness for which several metrics were proposed. Start times may be predicted from runtime estimates if using conservative backfilling, relative changes toward this prediction are calculated in [9] [8]. In [6], differences of larger actual start times and predicted start time in relation to the overall number of jobs are used as unfairness metric to judge the fairness of job schedulers. Another unfairness metric—suitable e.g. for time sharing—considers any positive differences between needed resources and allocated resources over the final runtime of the job. It means how many resources a job uses compared with how many resources a job is supposed to have. Both metrics are calculated in retro to compare the overall fairness of different scheduling policies. As a result, no-guarantee backfilling and SJF (shortest-job-first) rate as much more unfair under both metrics than FCFS and LXF (largest-expansion-factor-first with expansion factor being the current relative response based on the estimated runtime). Wide jobs were found to be treated extremely unfair under no-guarantee backfilling.

Slack was also applied in grid computing, mainly to deal with the problem of reservations (which can cause fragmentation) and finding scheduling slots. Reservation times may be kept flexible within certain time frames to adjust to dynamic resource availabilities, including dynamic finishing times of other jobs or insertion of other jobs for better utilization [2][4][3]. In [2], heuristics are used to re-schedule some of the reservations. This approach is suitable for workflow scheduling. With a large slack factor (applied to runtimes!) of 2, the probability of rejecting a reservation decreases by about 80%. A similar approach for slack calculation is applied in [3], where slack is calculated as average waiting time which can be a multiple of the average runtime if the load is high (and much more if the job's runtime is far below the average). A similar idea is to extend reservations for tasks in a workflow graph to deal with uncertainties and possible delays from preceding jobs overrunning their reservations [12].

Chapter 3: Space Sharing and Backfilling Strategy

Our strategy of Backfilling with Fairness and Slack for Parallel Jobs Scheduling supports space sharing which we define more precisely as follows:

Space sharing: Resources are grouped in a dedicated way. The processors which are defined as space and shared by different processes, i.e., different groups of resources are assigned exclusively to certain parallel jobs.

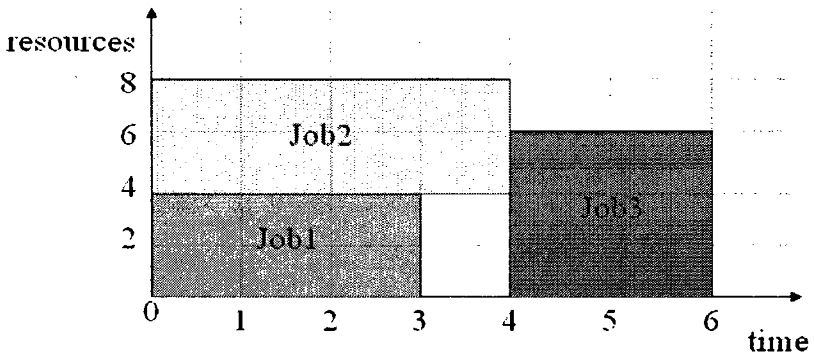


Figure 1. Space Sharing for different parallel jobs.

In Figure 1, there are eight resources just like space. At time 0, Job1 and Job2 are sharing all the eight resources, but exclusively using a group of them. And no more resources for Job3 at time 0, then Job3 has to start at time 4 after all the previous jobs finished and also use a group of resources.

In job scheduling, First-Come-First-Serve (FCFS) [16] is viewed as the fairest schedule. Take the Figure 2 as an example; the submission order is from Job1 to Job6, so the serving order is also from Job1 to Job6. However, some resources from time 0 to time t1, and some other resources from time 0 to time t2 are idle, which means no jobs are running on these resources, and this idle area is called fragment in this context. The reason why this happens is in that some large jobs come earlier could possibly prevent other later submitted jobs from using resources.

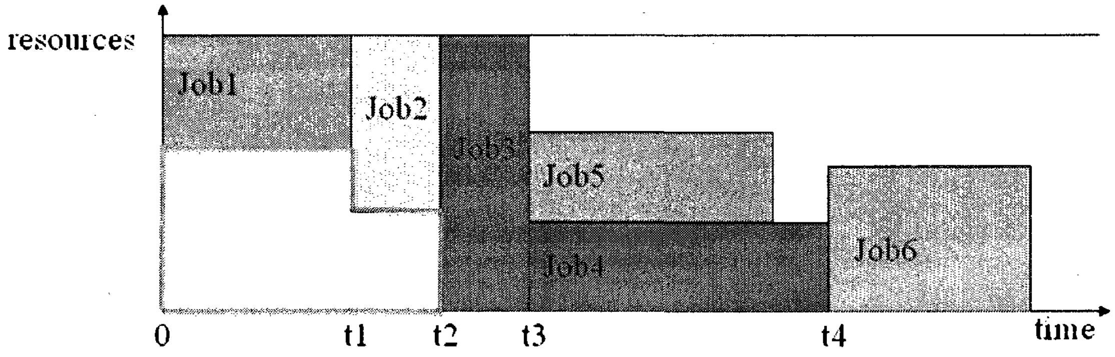


Figure 2. FCFS job scheduling.

In the figure above, Job3 submits earlier than Job4, Job5, and Job6. And Job3 uses all of the resources, so Job4 and Job5 have to wait until Job3 finished and use a group of resources exclusively. Obviously, FCFS has poor utilization and bad response time (Chapter 1), though it is the fairest schedule.

Hence, from the explanation and figure above, in order to improve resources utilization and decrease response time, the size of fragment should shrink as much as possible. To deal with this problem, intensive research has been done to decrease fragment and to optimize the space sharing strategy [27]. Backfilling [28] is one important approach of these different strategies. The main purpose of backfilling is to allow moving later smaller submitted jobs ahead to fill the fragment. There are basically two kinds of backfilling. One is easy backfilling [29], which moves later smaller jobs submitted ahead only if they do not delay the first job versus its normal scheduling time in the job queue. The other one is conservative backfilling [28], which moves later smaller submitted jobs ahead but do not delay any previous jobs in the job queue versus their normal scheduling time [26]. In this context, all the backfilling are applied based on conservative backfilling. And conservative backfilling is explained in details as follows:

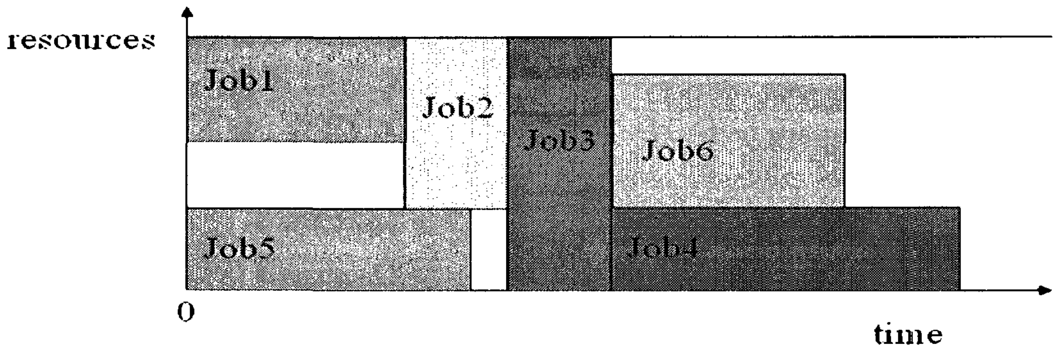


Figure 3. Conservative Backfilling.

Comparing Figure 3 with Figure 2, Job 5 starts at time 0 instead of time t3. When a job is firstly submitted, it will check all the fragments to find a fragment which is large enough to schedule the job. The fragment which is large enough means the number of free resources is at least the same as the size of job, and the idle time for these resources should at least equal to the runtime of this job. In Figure 3, at time 0, the number of resources is bigger than the size of Job4, but the idle time of these resources is less than the runtime of Job4. If backfilling Job4, then Job3 will be delayed versus its normal scheduling time which disobey the definition of conservative backfilling. Hence, Job4 has to be scheduled after all of its previous jobs finished. However for Job5, the situation is different, because the idle time of these resources is larger than the runtime of Job5. If Job5 is backfilled, all jobs are not delayed versus their scheduling time. Therefore, Job5 is scheduled at time 0 and sharing a group of resources exclusively sharing with Job1. Comparing Figure of 2 with Figure 3, the size of fragment is much smaller than in Figure 3. In another word, the utilization of resources increases, and in the mean time the response time of jobs decreases. Therefore, the changes of both metrics stand that the performance is improved.

From the description of conservative backfilling above, there is obviously a limitation of backfilling which it is job's runtime. The job runtime is usually obtained based on user's estimation or history date of running this job, however it is not accurate. Besides, it is necessary that one more thing should be obtained before doing conservative backfilling which is every job's estimate start time. Traditionally, the estimate start time of every job is calculated based on FCFS and conservative backfilling [1]. At first, it simulate conservatively backfilling all the previous jobs and get the whole fragments. Then conservatively backfill again on the new submit job, and the time for this job to start running is this jobs' estimate start time.

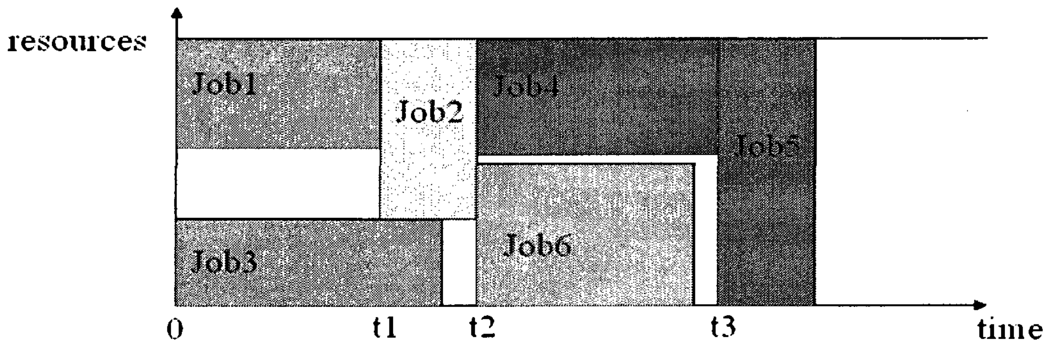


Figure 4. Job's Estimate Start Time.

In Figure 4, all jobs get their estimate start time based on FCFS and conservative backfilling. Such as Job1's estimate start time is at time 0, Job2 has to start running after Job1 and its estimate start time is t1. Then Job3 is submitted, which can be conservatively backfilled at time 0. Hence Job3's estimate start time is also at time 0 which is the same as Job1. Job4 and Job5 have to schedule at time t2 and t3 respectively. Job6 performs similarly with Job3. Job6 is conservatively backfilled at time t2, so Job6's estimate start time is at time t2. However, Backfilling with Fairness and Slack for Parallel Jobs Scheduling does not simply adopt conservative backfilling, and it makes some changes to the original conservative backfilling strategy. Hence, the way to calculate every job's start time is different and is explained in detail in Chapter 4.3.1.

Chapter 4: Backfilling with Fairness and Slack for Parallel Jobs Scheduling

This algorithm of Backfilling with Fairness and Slack for Parallel Jobs Scheduling is implemented as an extension of the existing coarse-grain Scojo-PECT preemptive scheduler [18]. In coarse-grain Scojo-PECT preemptive scheduler, the rule always chooses the first fit jobs to backfill according to jobs' submission order which can be viewed as FCFS. The backfilling in Scojo-PECT can be conservative backfilling or easy backfilling, and the jobs chosen to backfill cannot delay any previous jobs or the first job according to different backfilling strategies. However, Backfilling with Fairness and Slack for Parallel Jobs Scheduling, instead of choosing jobs to backfill according to jobs' submission order, looks further ahead and chooses later submitted jobs in the queue to backfill if they can improve utilization significantly. However, this could be unfair to some jobs ahead of the queue. Therefore at the same time, this backfilling strategy adds slack to each job. Hence, the slack can act as a controller so that unfairness of each job would be limited and no individual job is delayed severely. And meanwhile, backfilling is also set to be more flexible, in which each job can be delayed by its slack other than no delay at all.

4.1 Assumptions and Goals

1. The Backfilling with Fairness and Slack for Parallel Jobs Scheduling algorithm is based on the following assumptions.

- In each node, there is only one processor, and at each time, only one process is scheduled in each processor.
- Instead of threads, jobs are composed by processes, with which jobs are more flexible to schedule on different processors, e.g. MPI applications.
- The job's size is the number of processes. And also the number of processors a job needs is the same with the number of processes, because of the first assumption.
- Besides the job's size, job's runtime and other characteristics information sufficiently to calculate unfairness should be known when a job is submitted. For a job's size, users can provide it. For a job's runtime and other characteristics, one way is based on history data of scheduling this job. The other way is to simulate running this job with a sample input. This sample input is actually the same used in the testing environment. Then get runtime and characteristics from the execution.
- Mostly there are three kinds of jobs based on the situation of processes they need. The first one is rigid jobs, which means the size of this kind of job cannot be changed with fixed number of processes. The second one is moldable jobs, and their size can be changed only at the time they begin to run. And the last one is malleable jobs. The size of this kind of job can be changed at the start running time and even during the execution. However only rigid jobs are considered in this implementation.
- From the perspective of jobs' property, jobs can be serial which only need one processor or parallel jobs with power-of-two size which need at least two processors. This can be found in many practical parallel system logs [19]. In this workload, most jobs are serial and some are parallel.

2. The Backfilling with Fairness and Slack for Parallel Jobs Scheduling is designed with following goals:

- Provide a flexible conservative backfilling takes advantage of every jobs' slack (Chapter 4.2). Slack defines that every job can start running later than they are supposed to be. However, this time period of each job is different. The jobs with longer wait time (Chapter 4.2) get smaller slack factor. In contrary, the jobs with shorter wait time get bigger slack factor. And there is also a range of all the jobs' factor, which means the factor cannot be too big or too small.
- Lookahead in the queue to find best packing of jobs to backfill, instead of backfilling first fit jobs according to submission order. There are two goals we want to check. For the primary goal, the jobs selected to backfill should stay close to FCFS, but keep utilization high and unfairness low. For the opposite goal, the jobs selected to backfill increase utilization significantly, but also should stay reasonably close to FCFS and keep unfairness low.
- The order of jobs will be changed under the backfilling strategy compared to the FCFS order. So the average unfairness is applied to all jobs to make sure that no individual job is treated too unfairly or too preferentially.
- Improve system utilization, typically during the high-load phases. Reduce average relative response time to provide better services to users.

4.2 Scheduling Objectives

The objective of the scheduler is to select best packing of jobs from all possible packings in order to obtain higher utilization in phases of high-load, and decrease jobs' average response time while keeping unfairness low. In [20], this paper points out that higher utilization in high-load phases seems to get better average response time. Only some jobs in high-load phases will be delayed when the schedulers with less utilization. However, these delayed jobs will be scheduled when the following low-load phases comes. Therefore, from the whole process, all the jobs could not queue-up. In addition, when the schedulers have high utilization, they would not delay jobs during high-load phases and keep the resources idle or very lowly loaded during low-load phases. Note: as long as jobs could not be queued-up and the schedulers can deal with the workload, overall utilization almost does not change under different scheduling policies if the workload remains the same.

Thus we use both job average response time and utilization improvement as primary objectives while every job's delay factor is a constraint. This requires us to formally define high-load phases, resources utilization, unfairness, and response time. Terms used through the thesis are listed and explained in Table 1.

S_i	Size (whole process number) of Job i
T_i	Runtime of Job i when scheduled individually with one process per node
PPN_i	Number of processes for Job I per node
$T_{makespan}$	Total runtime of the whole workload
N	Number of nodes in the cluster (machine size)
$U_{resouerce}$	Resource utilization
FST_i	Fair start time of Job i when scheduled in FCFS with conservative backfilling
AST_i	Actual run time of Job i when scheduled in this context's strategy
F_{slack}	Slack factor used for fairness check
$Phase_H$	High-load phases
N_{wait}	The number of jobs in the waiting queue
N_H	A threshold for rating as high load
Nup_i	The number of used processors during a time period
Tp_i	A time period

Table 1. Terms used throughout the formulas in the thesis.

During some periods, the number of jobs in waiting queue is bigger than a defined threshold, then these periods are defined as high-load phases and $Phase_H$ are defined as:

$$Phase_{H,l} = [t_l \text{ with } N_{wait,tl} \geq N_H \;\&\&\; N_{wait,tl-1} < N_H \text{ for } t_{l-1}, \;\&\&$$
$$t_j \text{ with } N_{wai,tj} < N_H \;\&\&\; \text{any } t_k \text{ between l and j has } N_{wait,tk} \geq N_H] \quad (1)$$

t_l, t_j and t_k are certain time points of status changes. Node utilization U_{node}, is the percentage of used-nodes time over the makespan:

$$U_{resource} = \Sigma_{i \text{ in all time periods}} (Tp_i * Nup_i) / (T_{makespan} * N) \quad (2)$$

Similarly, $U_{resource}$ during a high-load phase is the percentage of used-resources time over high phases.

According to the calculation of jobs' unfairness, there are many different considerations in the literatures [22] [23] [24] [6]. We consider any difference between job's fair start time (Chapter 4.3.1) and its actual start run time. The fair start time is calculated based on FCFS and conservative backfilling with slack. The actual start time is the time this job is scheduled. Since estimation of fair start time via simulating slack conservative backfilling at submission time is possible explained in Chapter 4.3.1, we record the fair start time (FST_i) for job i. When the job is actually running on one or some nodes, we record the actual start time (AST_i) for job i. Unfairness is then provided by calculating the start running time changes (FST_i compared to AST_i) on average of all the jobs from the waiting queue which happens due to our effort for utilization improvement and by permitting limited delays on every job. If the AST − FST is less than zero, the job was given preferential treatment; if it equals to zero the job was treated fairly; and if it is greater than zero, it was treated unfairly.. In the equations below, n is the number of all jobs that are considered. For the jobs are treated unfairly, $\text{OverallUnfairness}_n$ is defined as below:

$$\text{OverallUnfairness}_n = \sum_{i \in jobs} \max((AST_i - FST_i),0) \,/\, \sum_{n \in jobs} 1 \qquad (3)$$

And, for the jobs are treated preferentially, this unfairness is defined as SkipUnfairness_n:

$$\text{SkipUnfairness}_n = \sum_{j \in jobs} \max((FST_j - AST_j),0) \,/\, \sum_{n \in jobs} 1 \qquad (4)$$

The overall average value of OverallUnfairness and SkipUnfairness are defined as the sum of the OverallUnfairness and SkipUnfairness divided by the number of jobs separately. Therefore, jobs which are given unfair treatment or preferential treatment cannot bring down the metric, as we only sum over unfairly treated or preferentially treated jobs. Also, we divide the sum by the total number of jobs. So if only one job is treated unfairly by T, the overall unfairness is T/N. While, for an example, a scheme where N jobs are treated unfairly by T_1,

$T_2, \ldots T_n$, then will have an overall unfairness of $(T_1+T_2+ \ldots +T_n) / N$. And it is the same with jobs treated preferentially.

4.3 Scheduling Algorithm

The original backfilling strategy sorts jobs in submission order and chooses the first fit job. However, as explained before, this could miss more optimized combinations of jobs. Hence, we choose to look further ahead and try more combinations of jobs, and then backfill more than one job. Two criterions are used in job selection, which are fair start time and slack.

4.3.1 Job Fair Start Time

The Scojo-PECT (Chapter 4.4) scheduling predicts every job's start time by simulating its scheduling in the future. The basic rule is FCFS. However, if a job can be conservatively backfilled, then backfill it and the time is this job's estimate start time (Chapter 3). Scojo-PECT also employs FCFS and typical conservative backfilling, in which every previous job is not delayed versus their normal scheduling time due to backfilling. However, in our backfilling strategy, every waiting job is assigned a delay slack to make the conservative backfilling more flexible. A job can be backfilled when its start time is estimated, but cannot be backfilled when it is actually scheduled. Hence, the original method of start time prediction is not suitable any more. Therefore, we introduce another way to estimate the job's fair start time from [24]. There are two cases in estimating the fair start time of jobs:

- Case 1: the waiting job is the job whose fair start time needs to be estimated.

  In this case, add this job in the back of possible fragments (free resources during certain time in Chapter 3) to schedule and determine its fair start time.

- Case 2: the waiting job is not the target of fair start time estimation.

  In this case simulate all of this kind of jobs to start based on conservative backfilling.

From the descriptions of these two cases, when calculating a job's fair start time, this job is just scheduled in FCFS order. But jobs submitted before it are conservatively backfilled. The reason is that a job can be conservatively backfilled into a fragment when estimating its start time but possibly may not be conservatively backfilled again in actual scheduling later. Hence, for calculating the fair start time, it should include this situation. So, FCFS is employed to calculate its fair start time. However, our backfilling still adopts conservatively backfilling. Therefore those submitted earlier jobs should be conservatively backfilled. Under these two cases, the fair start time obtained in Backfilling with Fairness and Slack for Parallel Jobs Scheduling is suitable to calculate unfairness (Chapter 4.2).

The pseudo code for calculating the fair start time is shown in Figure 5:

```
freeQueue = initBackfillOptions();            //find all the fragments that can
                                              be used to backfill jobs
if(this job is needed to determine its FST){  // this is case 1
    FST = add_lastframent;                    // add this job in the back of
                                              fragments and get fair start time
}
else{                                         // this is case 2
    simulateFillWithConservativeBackfill;     //conservative backfill this job
}
```

Figure 5. Pseudo code for calculating fair start time.

4.3.2 Job Slack

From the description of conservative backfilling (Chapter 3), we can tell that conservative backfilling is very restricted. This restriction is used to ensure fairness. However, on the other hand, it could possibly prevent backfilling jobs which can fit into free resources but their runtimes are larger than resources' idle time. Therefore, applying a slack to every job can relax the restriction, however slack must be controlled to avoid the violation of fairness.

According to the Lublin-Feitelson workload model [19], jobs are generated at a time but cannot be scheduled on the resources immediately. Hence, some jobs have to wait for enough resources. From the description above, Scojo-PECT is able to estimate job's start time and record it. When a job is scheduled to run, Scojo-PECT is also able to retrieve the time and record it. Therefore, every job's waiting time can be obtained by calculating the difference between a job's submission time and the time it starts running, which is defined as T_{wait}. When a job has already been waiting for a long time, the user can not afford to wait any longer. On the contrary, a job that has only been waiting for a short time, the user can afford to wait longer. Therefore, the slack factor F_{slack} defined as follows:

$$F_{slack} = \min\{S_{up,} K2 * \text{Math.exp}(T_{wait} * K1) + S_{down}\} \qquad (5)$$

Every job's slack factor is related to job's wait time. K_1 and K_2 are two parameters to control slack factor. S_{down} and S_{up} are the minimum and maximum values of slack factor range.

In Formula 5, the longer the wait time is, the smaller the slack factor is, and vice verse. Also the decrement is non-linearly, which means the slack factor decreases more rapidly when T_{wait} is small, and more slowly when T_{wait} is large. Figure 6 shows the slack factor curve.

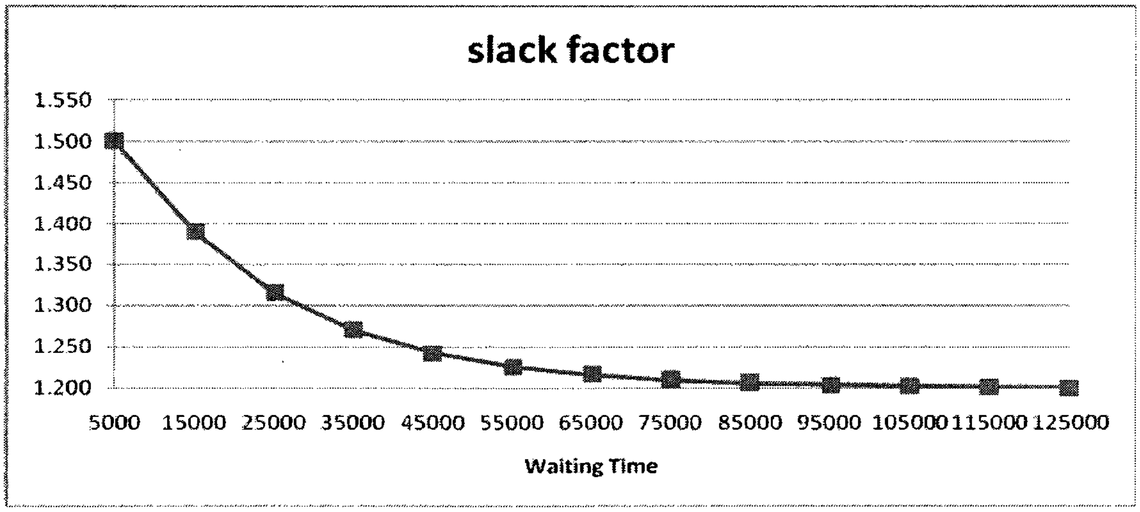


Figure 6. Slack Factor with different waiting time.

In Figure 6, the horizontal axis is the waiting time, and its unit is seconds. The vertical axis is slack factor each waiting time can get, and it is in a specific range. In the beginning, when the waiting time varies from 5000 seconds to 65000 seconds, the slack factor drops from almost 1.50 to 1.21 which is 0.29. On the other hand, the waiting time changes from 65000 seconds to 125000 seconds, the slack factor drops from 1.21 to 1.20 which is only 0.01.

Obviously, for the same amount of change in waiting time, the decrement of the slack factor is much smaller for larger values of waiting time.

However, how to assign this slack at each time is another important issue. A job only can use a fraction of its slack at each round of backfilling, so that every job cannot use up all its slack at the very first time, and there is no delay any more.

The wait time is used to determines how much can be used at each time. We define $T_{areadywait}$ as the time this job has already waited, $F_{partslack}$ is the fraction of slack factor the job can get at this time, $F_{mininalslack}$ is the minimal slack factor which is a constant number but would be changed in tests. So the fraction of slack is calculated using Equation 6:

$$(F_{partslack} - F_{mininalslack}) / (F_{slack} - F_{mininalslack}) = T_{areadywait} / T_{wait} \qquad (6)$$

In Equation 6, we can see that the ratio of $(F_{partslack} - F_{mininalslack})$ to $(F_{slack} - F_{mininalslack})$ is the same as the ratio of $T_{areadywait}$ to T_{wait}, and the fraction slack a job can get changes at each time. The longer a job has waited the larger the fraction of its slack that can be used. Therefore, in this case, a job could not use all its slack at the very first time, and cannot be pushed back any more. Transferring Equation 6, we can get the calculation of $F_{partslack}$ as below:

$$F_{partslack} = (T_{areadywait} * (F_{slack} - F_{mininalslack}) / T_{wait}) + F_{mininalslack} \qquad (7)$$

4.3.3 Job Selection

In recent years, some strategies [33] [34] [37] have been introduced to select jobs for backfilling into idle resources and many optimization algorithms have been proposed [38] [40]. However, in this context, instead of backfilling first fit jobs according to their submission order, it looks further ahead into job queue and backfills jobs that can improve utilization significantly and also can stay close to FCFS. There are 2^n possible combinations of jobs; n is the number of waiting jobs that will be considered. It is a NP-Hard problem. So when the n is very large, this problem becomes very difficult. Hence, it is hard to get the optimal solution, but we can get an approximate one based on some heuristics.

Backfilling with Fairness and Slack for Parallel Jobs Scheduling is implemented via branch and bound algorithm [35] [36] [39]. The algorithm description is generalized to work with three kinds of backfilling (Chapter 4.4). The details of algorithm steps are explained as below:

- Step 1: check whether this algorithm can be applied or not.

  There are two conditions that determine whether this algorithm can be applied. The first one is: that the remaining number of waiting jobs is more than zero. Otherwise, the algorithm is meaningless. The second one is: that the amount of processed waiting jobs should be less than the defined number. The define number is the maximum number that would be considered to lookahead. This condition restricts lookahead a certain amount of waiting jobs.

- Step 2: start building an incomplete tree based on the branch and bound algorithm.

  The procedure of finding possible combination of jobs is based on the branch and bound algorithm idea with some changes.

  For every waiting job, there are two nodes. The left node means this job is selected to be backfilled, and the right one means this job is not to be backfilled. This covers all the 2^n possible combinations of jobs.

  There are two cases. Case 1: if the job is the first waiting job from the waiting queue, then create two nodes for it. The left node is with this job selected and the right node is without. Otherwise, in case 2, there are already some nodes in this tree, hence continue creating two children of each node. The left child is with next job selected, and right child is without.

- Step 3: check if there are left node or left children.

  If the amount of left node or left children is larger than zero, then continue with step2 of case two. If not, try next waiting jobs and continue with step 2 with case 1. We only need to check every left node or left children. The reason for this is that, in fact, the right child of next layer of tree is the same situation with the left child of previous layer.

- Step 4: sum up the sizes of selected jobs and compare with the amount of free resources at current time.

  This is one condition for cutting the tree. If the sum of sizes of selected jobs is larger than the number of free resources at the current time, then obviously there are not enough resources to schedule this combination of jobs. Hence this combination of jobs can be deleted and the whole branch of this node as well.

- Step 5: compare the delays of waiting jobs that are not selected with their corresponding slacks.

  Another constriction to cut the tree is the slack of all other not selected waiting jobs. When backfilling waiting jobs, this could delay some other not selected waiting jobs. Therefore, we need to check all the other not selected waiting jobs that should not exceed their slack at each time. If any of these not selected waiting jobs' slack is exceeded, then this combination of job is deleted, and the whole branch of this node included.

- Step 6: save the jobs combination as temporary results, and the combination of jobs as a node of this incomplete tree.

  Only both of two constrictions are satisfied, then this combination of jobs can be the results that would be backfilled and saved as temporary results. Also, this node is saved and will be the parents of the tree to create new children in step 2 with case 2.

- Step 7: finish trying all possible combinations of jobs, then decide to schedule which backfilling strategy.

  When there are no more waiting jobs or the amount of processed waiting jobs is larger than the number defined as the maximum lookahead number, then try to schedule waiting jobs. If the sum of all possible combinations of jobs is zero, which means there are no combinations of jobs satisfying both of two constrictions, then choose the original backfilling strategy. Otherwise if there are some combinations of jobs, then try Backfilling with Fairness and Slack for Parallel Jobs Scheduling.

- Step 8: if backfill with fairness and slack for parallel jobs scheduling is selected, then decide which packing of jobs could be chosen to backfill.

  There are two goals in backfilling of fairness and slack for parallel jobs scheduling. Goal 1 is to choose the packing of jobs which stay as close to FCFS as possible but keep utilization high. Goal 2 is to choose the packing of jobs which increase utilization greatly but stay reasonably close to FCFS.

- Step 9: if the remaining number of combination of jobs is more than zero, then find the best packing of jobs to backfill according to every packing of jobs' OverallUnfairness, SkipUnfairness (Chapter 4.2) and utilization.

  Goal 1: compare each packing of jobs' OverallUnfairness, SkipUnfairness and utilization with the smallest OverallUnfairness of all combinations of jobs, SkipUnfairness and utilization of the combination with the smallest OverallUnfairness as below:

  OverallUnfairness / smallest OverallUnfairness <= max OverallUnfairness range && SkipUnfairness / (SkipUnfairness with smallest OverallUnfairness) <= max skipfairness range &&

  Utilization / (utilization of smallest unfairness) >= min utilization range

Goal 2: compare each combination of jobs' utilization, OverallUnfairness, and SkipUnfairness with the highest utilization of all combinations of jobs, the OverallUnfairness and SkipUnfairness of the combination with the highest utilization as below:

highest utilization / utilization <= minmum utilization range &&

(OverallUnfairness with highest utilization) / OverallUnfairness >= max OverallUnfairness range &&

(SkipUnfairness with highest utilization) / SkipUnfairness >= max SkipUnfairness range

- Step 10: if satisfying either one of these two goals' conditions, the comibnation can be saved.

Goal 1: SkipUnfairness and Utilization of next packing are compared with the latest saved packing's, but OverallUnfairness is still the smallest one. The reason is the aim of Goal 1, which always chooses the jobs which stay as close to FCFS (always compare with the smallest OverallUnfairness) and improve utilization significantly (always compare with the higher utilization).

Goal 2: OverallUnfairness and SkipUnfairness of next packing are compared with the latest saved packing's, but Utilization is still the highest one for the reason of Goal 2's aim, which always choose the jobs which improve utilization significantly (always compare with the highest utilization) and stay reasonably close to FCFS (always compare with the smaller OverallUnfairness).

- Step 11: If no packing of jobs can satisfy the conditions of these two goals:

  Goal 1: the packing of jobs with smallest OverallUnfairness is selected.

  Goal 2: the packing of jobs with highest Utilization is selected.

- Step 12: if the remaining number of combination of jobs is zero, then backfill the final selected waiting jobs on the free resources at current time.

The core of this scheduling algorithm is shown in Figure 6 and Figure 7.

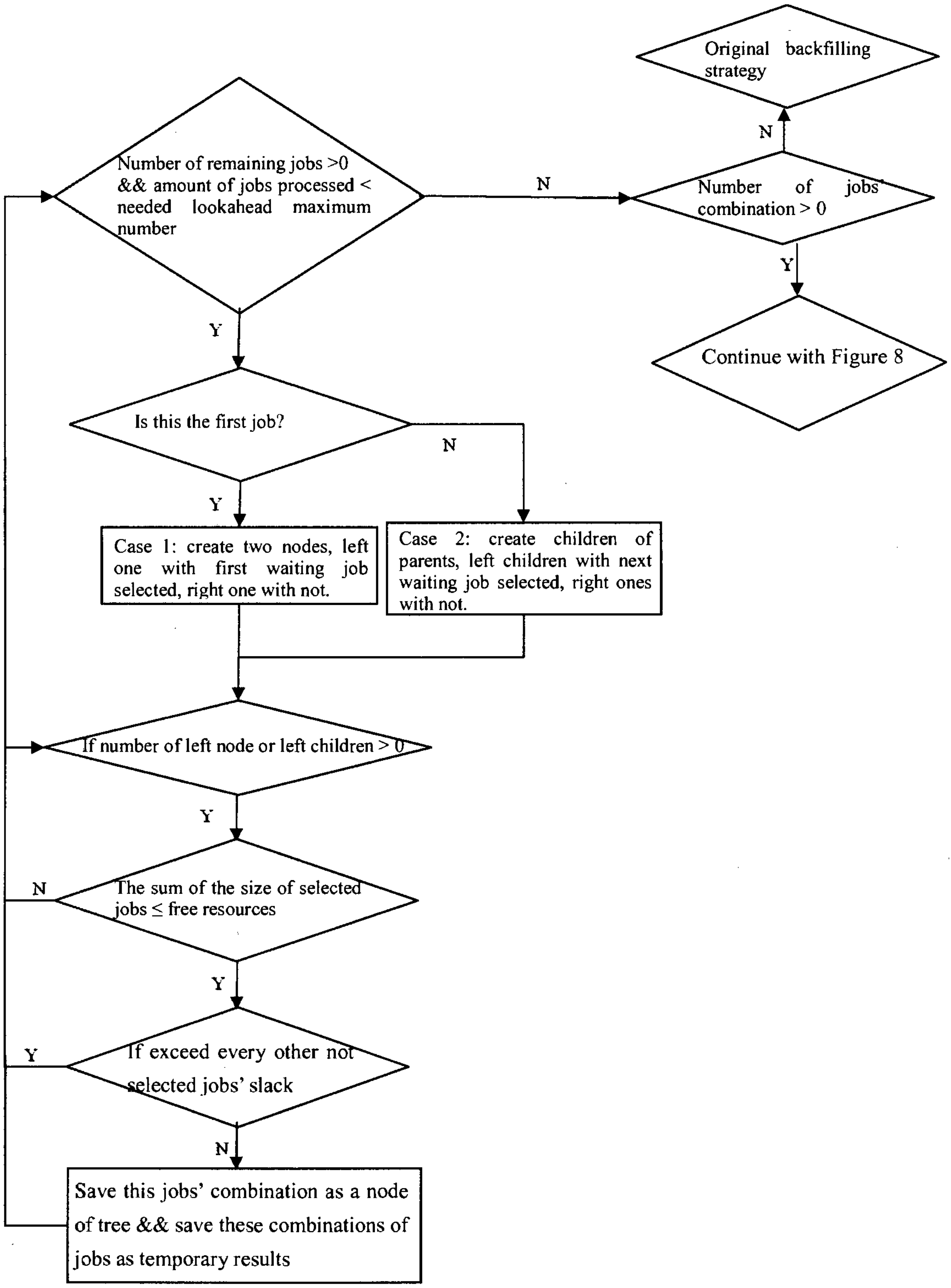
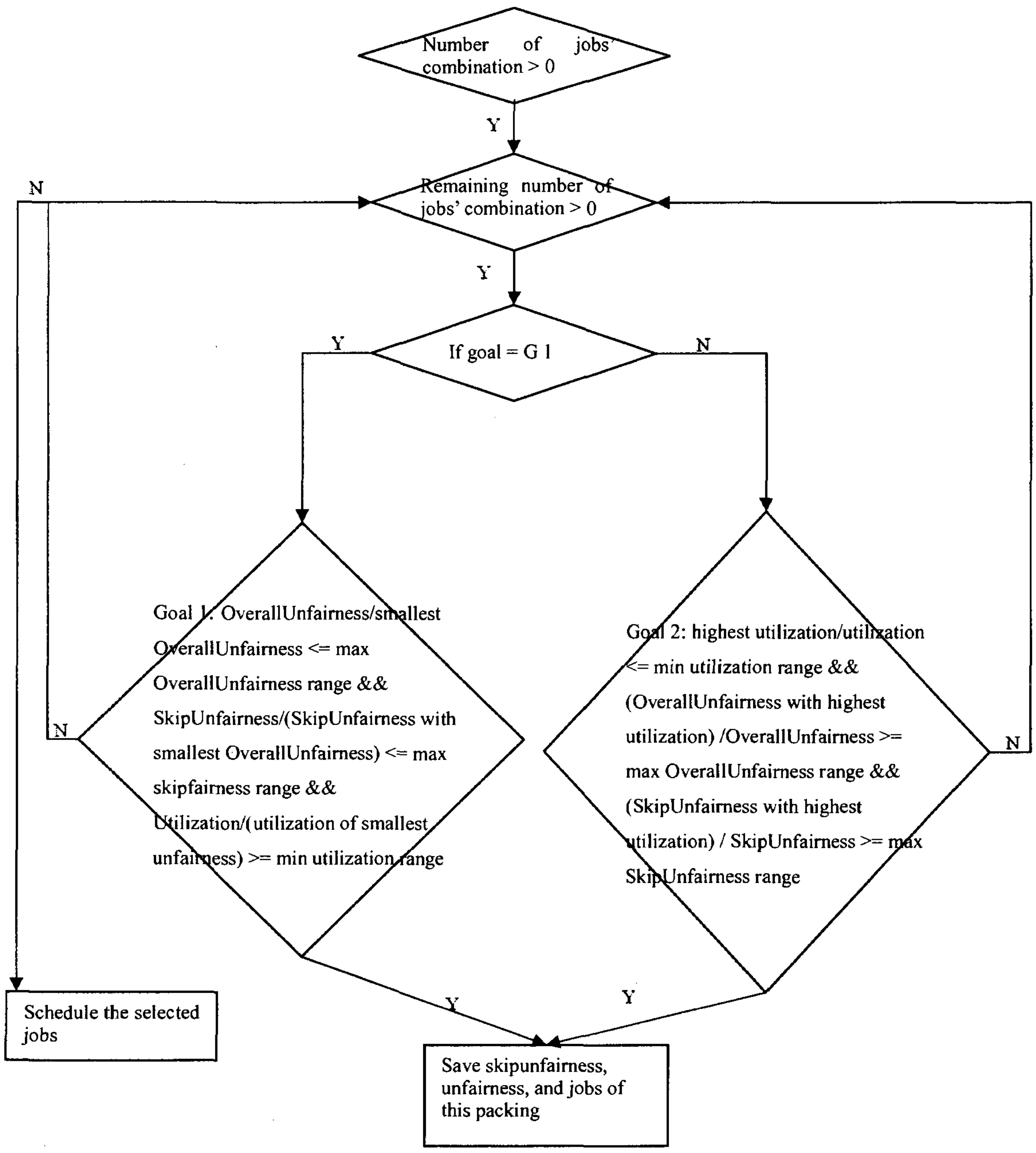

Original backfilling strategy
N
Number of remaining jobs >0 && amount of jobs processed < needed lookahead maximum number
N
Number of jobs' combination > 0
Y
Continue with Figure 8
Y
Is this the first job?
N
Y
Case 1: create two nodes, left one with first waiting job selected, right one with not.
Case 2: create children of parents, left children with next waiting job selected, right ones with not.
If number of left node or left children > 0
Y
N
The sum of the size of selected jobs ≤ free resources
Y
Y
If exceed every other not selected jobs' slack
N
Save this jobs' combination as a node of tree && save these combinations of jobs as temporary results

Figure 7. Partial Flow chart for core scheduling algorithm.



Number of jobs' combination > 0
Y
N
Remaining number of jobs' combination > 0
Y
Y
If goal = G 1
N
Goal 1: OverallUnfairness/smallest OverallUnfairness <= max OverallUnfairness range && SkipUnfairness/(SkipUnfairness with smallest OverallUnfairness) <= max skipfairness range && Utilization/(utilization of smallest unfairness) >= min utilization range
N
Goal 2: highest utilization/utilization <= min utilization range && (OverallUnfairness with highest utilization) /OverallUnfairness >= max OverallUnfairness range && (SkipUnfairness with highest utilization) / SkipUnfairness >= max SkipUnfairness range
N
Schedule the selected jobs
Y
Y
Save skipunfairness, unfairness, and jobs of this packing

Figure 8. Partial Flow chart for core scheduling algorithm.

4.4 Incorporation into Scojo-PECT

Scojo-PECT [18] provides a framework of job scheduler provided by Dr. Sodan and her graduate students. Scojo-PECT is a coarse-grain preemptive scheduler [8] with service guarantees and predictability. With coarse-grain, the length of time slice in Scojo-PECT is tens-of-minutes range instead of seconds. And preemptive allows jobs being preempted at the end of time slice so that all memory space are freed and used by the next running jobs, while in the mean time, it can avoid the situation of gang scheduling [25] about overhead and memory pressures. Scojo-PECT makes constraint all the preempted jobs have to be restarted on the same resources so that no hard-to-support checkpointing [1] is needed at all.

Jobs are separated into three types according to their run time in Scojo-PECT, which are short job, medium job, and long job. Scojo-PECT also divides time in each interval into three kinds of slices, short, medium and long time slice. Each job type is scheduled in a virtual machine and run via time slices. Our scheduler uses space sharing per virtual machine. Hence in each job type, jobs are scheduled in FCFS order in their own time slices/virtual machines. However different kinds of backfilling as described below are applied in Scojo-PECT, short jobs are always backfilled in other time slices. Hence, short time slice is only provided if there are short jobs. Therefore the share of time slices in each interval is decided based on potential dynamic adjustment. In the context of this thesis, the relative time slices are kept static.

In Scojo-PECT typical backfilling is applied, in which later smaller jobs can be moved ahead if they do not delay any other jobs. Scojo-PECT supports conservative and easy backfilling. But in the presented work, we use conservative backfilling. Since the jobs are separated into different types, it is likely to increase the idle resources because of job sizes and runtime tend to be correlated [19], hence Scojo-PECT adds safe non-type slice backfilling applied on preempted and waiting jobs. It requires that preempted or waiting jobs may be backfilled into other time slice and the backfilling is only valid until the end of slice, only if they do not delay any jobs of this time slice type jobs or of their own type according to the backfilling approach applied. If all time slices (resource shares) get the same service, medium and long jobs get similar service as standard space sharing with priorities, but serve short jobs better, then Scojo-PECT improves overall response times by about 50% [18].

Scojo-PECT is implemented and evaluated via discrete event simulation. There are five kinds of events which are job-submit event, job-finish event, slice-begin event, slice-end event and load-finish event. The system's status can only be changed by these events. For example, job-submit event means a new job is submitted and put in the waiting queue; job-finish event means a job has finished running and the resources occupied by the job are free; slice-begin event means a new time slice is created and set the system's status to be this slice; slice-end event means this slice ends, set the system's status to be none and all current running jobs are preempted; load-finish event means a job is finished loading into memory and it is in memory now.

The implementation of Backfilling with Fairness and Slack for Parallel Jobs Scheduling is based on the basic framework of Scojo-PECT. Scojo-PECT is applied per virtual machine or time slice and job type, and it is to resume or schedule jobs after one of the job-submit, job-finish and slice-begin events happens. Backfilling with Fairness and Slack for Parallel Jobs Scheduling needs to compare every job's fair start time and actual start time, and the predicted start time is relative time to current time. So in order to compare the fair star time and actual start time, here we transfer the relative time to absolute system time. Hence, in order to get the absolute system time, we need to know when the new slices begin in future based on system's workload and interval and time slices settings as explained above. Therefore, calculating job's fair start time starts when job-submit event happens.

Besides, the Backfilling with Fairness and Slack for Parallel Jobs Scheduling is applied to all three kinds of backfilling including typical conservative backfilling and two safe non-type slice backfilling. It is to resume or schedule jobs to run after one of the job-submit event, job-finish event, and slice-begin events happen.

Chapter 5: Experiments and Results Analysis

5.1 Experimental Set-up

The evaluation is performed on the basis of discrete event simulation. Scojo-PECT uses the Lublin-Feitelson statistical workload model [19] which is the best-available synthetic workload model. Every test is performed with different workloads and seeds, and results are averaged. Table 2 shows the characteristics of the workloads. There are several workloads we tested. Workload W1 is high workload, workload W2 is normal workload, workload W3 is lighter than workload W2 and workload W4 is much heavier than W1. In Workload W1, it sets the α parameter in the inter-arrival time distribution to a smaller value and subsequently creates shorter inter-arrival times. In workload W4, the α parameter is set to an even smaller value, while the α parameter is set to a larger value in workload W3.

From the table, obviously workload W3 time interval is bigger and workload W4 time interval is much smaller. Seed is used to generate random number for job's characteristics such as job's runtime, job's size. We selected 11 seeds. Some seeds generate about 10% higher workload than normal workload (W2), like seeds are 7, 31, 35, 70, 71 and 73. Some seeds generate about 10% to 15% lower workload than normal workload (W2), such as seed are 3, 13, 23, 99 and 103.

W1 (high load)	$\alpha = 10.23$
W2 (normal load)	$\alpha = 10.33$
W3 (light load)	$\alpha = 10.43, 10.73$
W4 (very heavy load)	$\alpha = 10.05$
seed	s= 71,3,7,13,23,31,35,70,73,99,103
Machine size M	128
Short jobs N_S	64%
Medium jobs N_M	19.5% (54% of Medium and Long)
Long jobs N_L	16.5% (46% of Medium and Long)
Work of short jobs W_S	0.5%
Work of medium jobs W_M	26.0%
Work of long jobs W_L	73.5%
Serial jobs	24%
Power-of-two size among parallel jobs	75%

Table 2. Workload characteristics.

In Scojo-PECT, there are three types of jobs according to their runtime. Short jobs are with the runtime less than ten minutes. Medium jobs are with the runtime less than three hours. And others are all long jobs. Each type of job has its own time slice. Short time slice is only created if there are short jobs. Therefore, one hour interval is set as time slice length for one short time slice if there is a short time slice, one medium time slice and one long time slice. The medium time slice and long time slice are set as 30% relative time share and 70% relative time share respectively. The time overhead for switching jobs when each time slice ends is 0 for all workloads.

To evaluate the performance of our algorithm, we mainly compare with original Scojo-PECT scheduling with original conservative backfilling. Table 3 shows scheduler parameters used in our experiments.

SLACK_UP	1.5	Maximum slack factor that a job can reach
SLACK_DOWN	1.2	Minimum slack factor that a job should have
SLACK_MININAL	1.1	Minimum slack factor when calculating how much part slack a job can obtain
K1	-0.5	One parameter to control how much partial slack a job can get at each time
K2	0.4	Another parameter to control how much partial slack a job can get at each time
SELECTWAITINGJOB_NUM	20	Number of waiting jobs defines how many jobs are lookaheaded
GOAL_FAIRNESS	True	Decide to select jobs as close to FCFS and improve utilization as goal G1, or higher utilization and stay as reasonably close to FCFS as goal G2

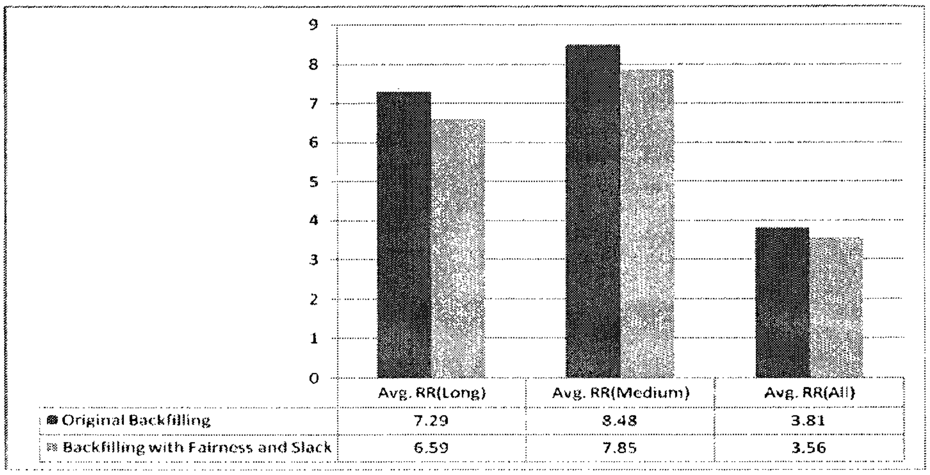
Table 3. Scheduler parameters and values used in the experiments.

We used the following metrics for comparison:

- Average relative response time (RR): pure runtime plus waiting time considering time slices in relation to pure runtime which is the time without time slicing while using cut-offs for very short jobs (only relevant for all-job evaluation)

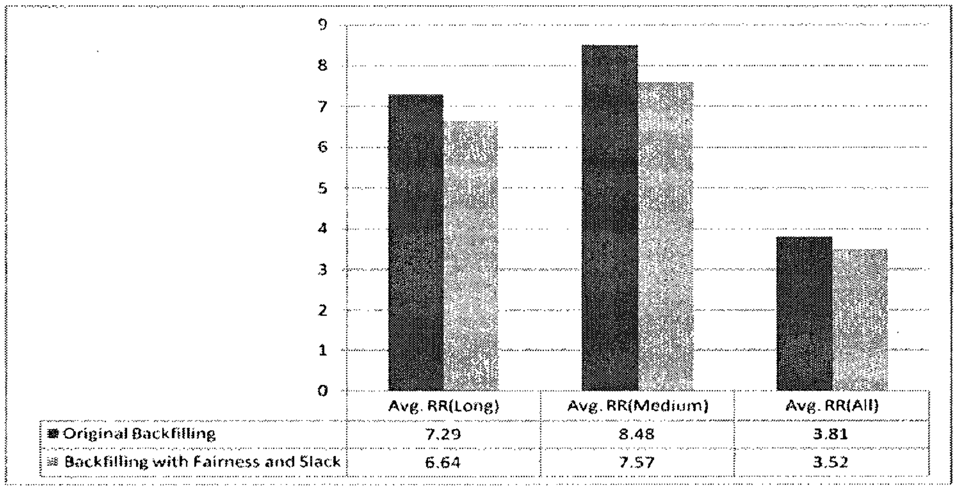
5.2 Performance Results

The performance results are tested under Backfilling with Fairness and Slack for Parallel Jobs Scheduling and compared with original Scojo-PECT scheduling using conservative backfilling. The Figure 8 and Figure 9 show average relative response time for both scheduling strategy.



	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	7.29	8.48	3.81
▨ Backfilling with Fairness and Slack	6.59	7.85	3.56

Figure9. RR for original scheduler and backfill with fairness and slack scheduling with Workload W1 and G1.



	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	7.29	8.48	3.81
▨ Backfilling with Fairness and Slack	6.64	7.57	3.52

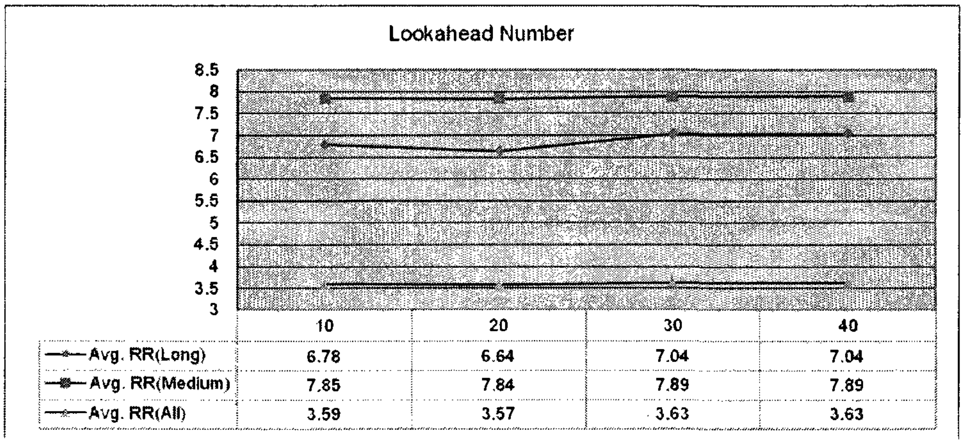
Figure10. RR for original scheduler and backfill with fairness and slack scheduling with Workload W1 and G2.

Figure 9 and 10 are the results of improvement comparing our backfilling with original backfilling. It shows Backfilling with Fairness and Slack for Parallel Jobs Scheduling performs better than original Scojo-PECT by 10.49% for long jobs, 7.89% for medium jobs, and 6.92% for all jobs for Goal 1. For Goal 2, it shows improvement by 9.73% for long jobs, 11.99% for medium jobs, and 8.13% for all jobs. Note long jobs and medium jobs perform similarly. Hence this scheduling can benefit both long and medium jobs. The scheduling with goal G2 performs a little better than the scheduling with goal G1.

The whole running time of our backfilling is about 10 minutes on average for all the seeds and 10000 jobs, and it is about 7 minutes for the original one. Our backfilling runs only 3 minutes longer which is about 40%. Therefore, the cutting of tree works well. It shows that Backfilling with Fairness and Slack for Parallel Jobs Scheduling finds the best packing of jobs effectively.

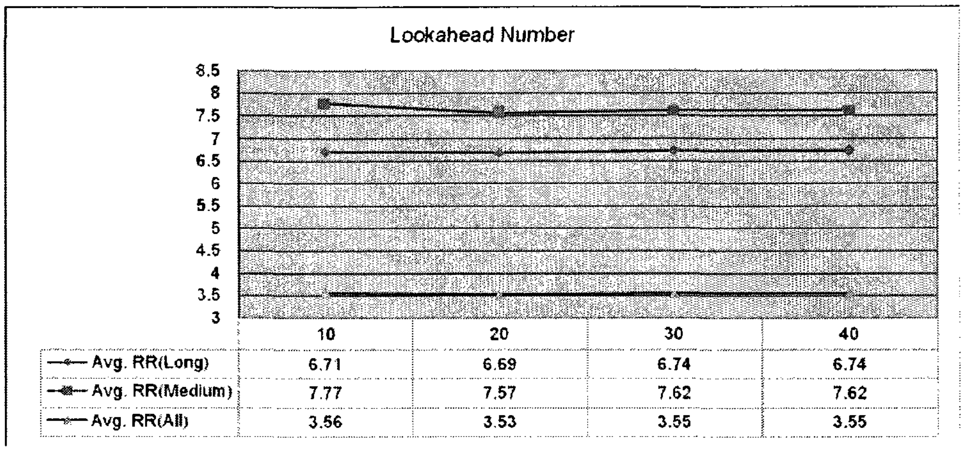
5.3 Impact of Different Parameters

We tested the parameters with different numbers, which define how many jobs will be lookaheaded and checked with which number the scheduler gets the best performance. In the Figure 11 and Figure 12 below, the workload is W1, and when the number is 20, then it is the best results. And if lookahead even more jobs, the results almost remain the same. Therefore lookahead more jobs makes no sense and is not needed.



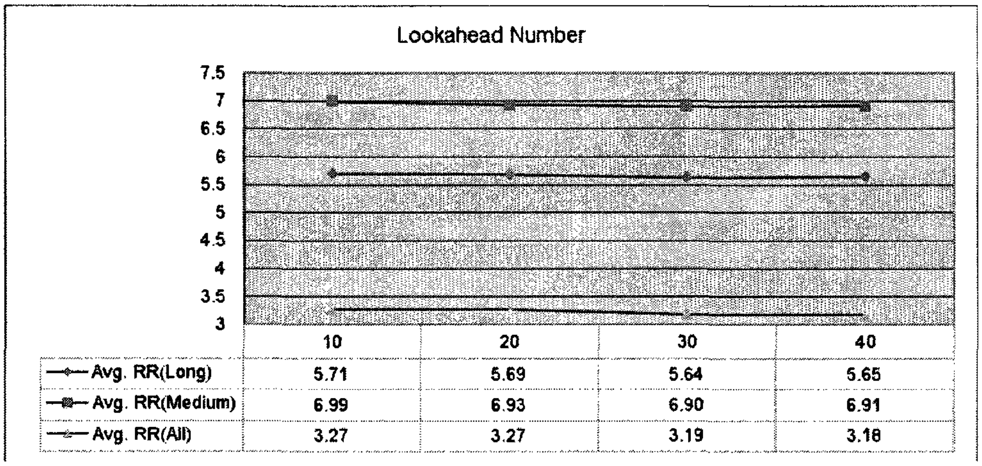
	10	20	30	40
Avg. RR(Long)	6.78	6.64	7.04	7.04
Avg. RR(Medium)	7.85	7.84	7.89	7.89
Avg. RR(All)	3.59	3.57	3.63	3.63

Figure11. RR for different lookahead number for backfilling with fairness and slack scheduling with Workload W1 and Goal G1.



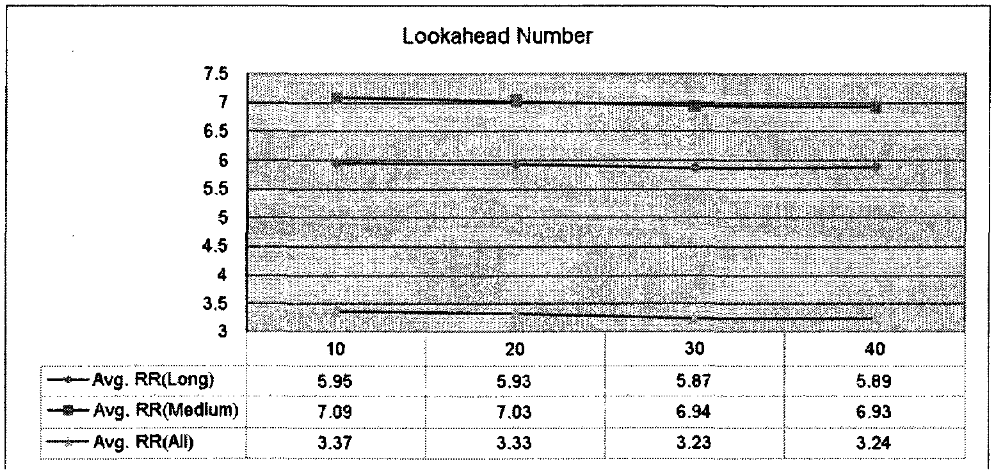
	10	20	30	40
Avg. RR(Long)	6.71	6.69	6.74	6.74
Avg. RR(Medium)	7.77	7.57	7.62	7.62
Avg. RR(All)	3.56	3.53	3.55	3.55

Figure12. RR for different lookahead number for backfilling with fairness and slack scheduling with Workload W1 and Goal G2.



	10	20	30	40
Avg. RR(Long)	5.71	5.69	5.64	5.65
Avg. RR(Medium)	6.99	6.93	6.90	6.91
Avg. RR(All)	3.27	3.27	3.19	3.18

Figure13. RR for different lookahead number for backfilling with fairness and slack scheduling with Workload W2 and Goal G1.

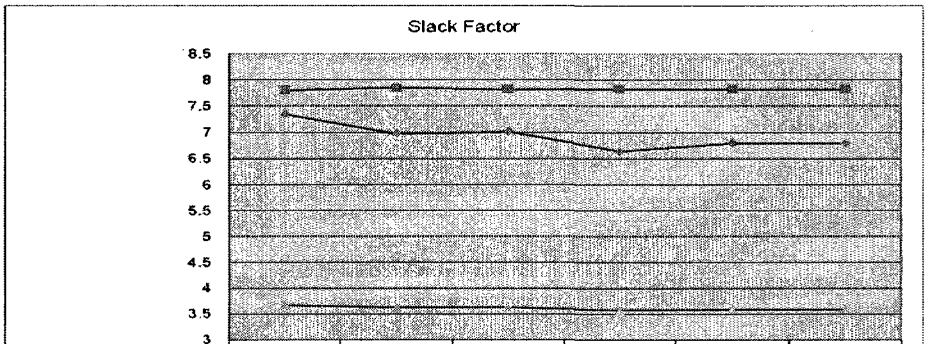


	10	20	30	40
Avg. RR(Long)	5.95	5.93	5.87	5.89
Avg. RR(Medium)	7.09	7.03	6.94	6.93
Avg. RR(All)	3.37	3.33	3.23	3.24

Figure14. RR for different lookahead number for backfilling with fairness and slack scheduling with Workload W2 and Goal G2.

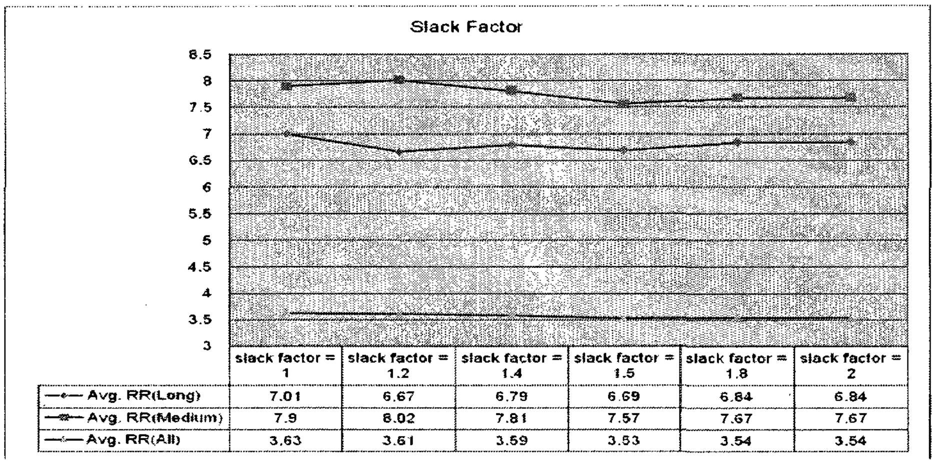
In the Figure 13 and Figure 14, the workload is normal workload W2. Lookahead 30 jobs is enough and can get the best results. Lookahead more jobs makes no sense and is not needed either.

We also test this scheduling strategy with different slack factors, and compare the results. The upper range of slack factor of more than 1.5 is only applied on smaller medium jobs which are with runtime less than one hour, and wait time less than two hours. We do not apply high slack factor too all jobs, because i.e. the job has been waiting for 3 hours, then it would wait for 6 hours with slack factor can go up to 2 which is very unfair. And for all the different slack factors, the lower range is 1.2, except the lower range is 1 when slack factor equals 1. The result graphs are shown as below:



	slack factor = 1	slack factor = 1.2	slack factor = 1.4	slack factor = 1.5	slack factor = 1.8	slack factor = 2
◆ Avg. RR(Long)	7.35	6.98	7.02	6.64	6.8	6.8
■ Avg. RR(Medium)	7.81	7.86	7.82	7.84	7.82	7.82
▲ Avg. RR(All)	3.67	3.63	3.63	3.57	3.59	3.59

Figure15. RR for different slack factor for backfilling with fairness and slack schdeduling with Workload W1 and Goal G1.



	slack factor = 1	slack factor = 1.2	slack factor = 1.4	slack factor = 1.5	slack factor = 1.8	slack factor = 2
Avg. RR(Long)	7.01	6.67	6.79	6.69	6.84	6.84
Avg. RR(Medium)	7.9	8.02	7.81	7.57	7.67	7.67
Avg. RR(All)	3.63	3.61	3.59	3.53	3.54	3.54

Figure16. RR for different slack factor for backfilling with fairness and slack scheduling with Workload W1 and Goal G2.

From the Figure 15 and 16, we can tell that when the slack factor is small, average relative response time is higher. The reason is when slack factor is small, then every job's slack is also small, thus more jobs could be possibly rejected to be backfilled because there is not enough time on idle resources. Hence, jobs have higher chances to be backfilled when increasing the slack factor. And the average relative response time decreases. However, until a certain value of slack factor, the average relative response time stays almost the same or even worse. The explanation could be that all possible combinations of jobs have been considered under a certain value of slack factor. In the Figure 14 and 15, when slack factor reaches 1.5, scheduling gets the best performance. All the results are tested when slack factor is in the range of 1.2 to 1.5.

We also tested another four parameters, which are SLACK_DOWN, SLACK_MININAL, K1 and K2, and assigning different values to these parameters does not change the performance significantly. The reason may be that changing the value of these parameters does not make enough change to every job's slack, and the number of possible combinations of jobs is almost the same.

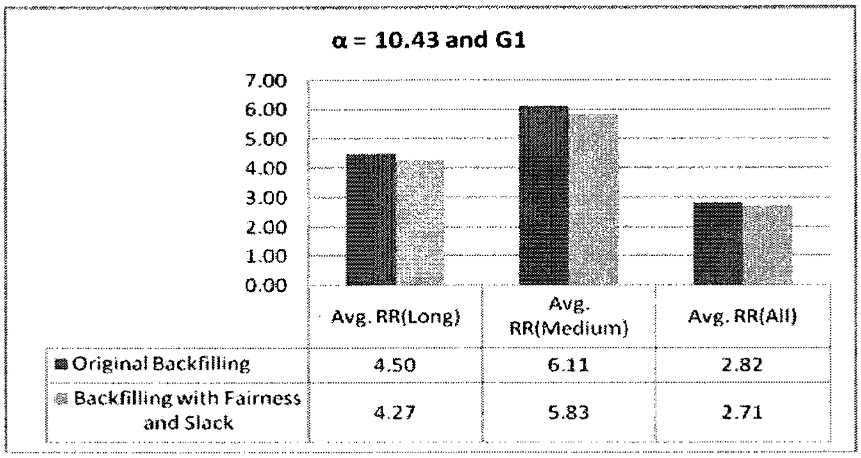
5.4 Impact of Different Workloads

We also tested our backfilling under different workload. Comparing the utilization among workload W3, workload W4 and workload W2 in Table 4 below, we find utilizations of α equaling 10.43 and 10.73 both are smaller than the normal workload with α equaling 10.33 and the utilization of workload with α equaling 10.05 is higher than the normal workload with α equaling 10.33. That is because workload W3 is light workload, and workload W4 is heavy workload.

workload	α = 10.33	α = 10.43	α = 10.73	α = 10.05
utilization	76.43%	71.90%	63.29%	85.29%

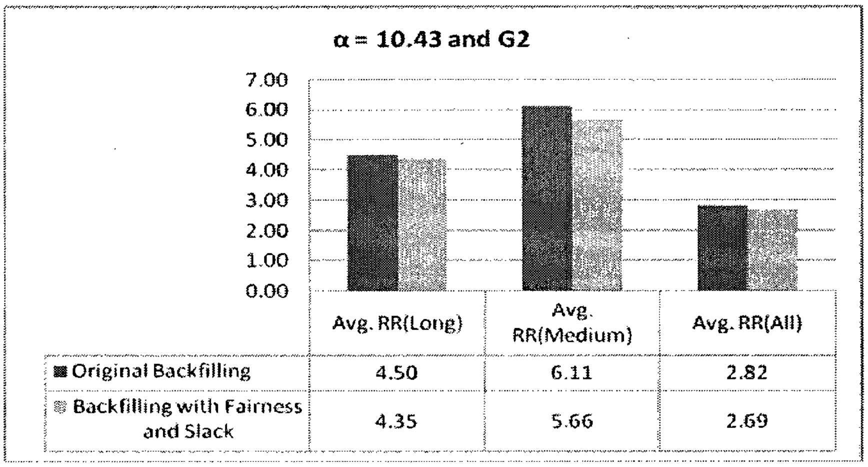
Table 4. Utilizations of different time interval.

In the Figure 17, comparing with original Scojo-PECT scheduler, Backfilling with Fairness and Slack for Parallel Jobs Scheduling with α equals 10.43 improves average relative response time by 5.32% for long jobs, 4.7% for medium jobs, and 3.74% for all jobs.



	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	4.50	6.11	2.82
▨ Backfilling with Fairness and Slack	4.27	5.83	2.71

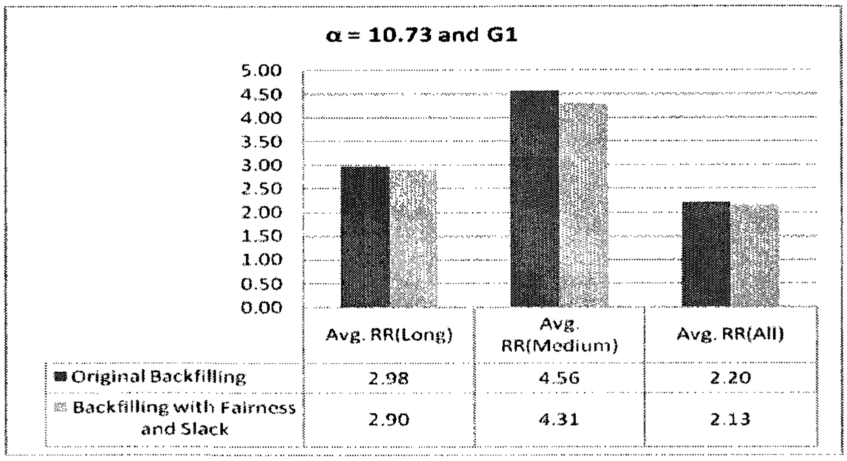
Figure17. RR for original scheduler and backfill with fairness and slack scheduling with α = 10.43 and G1



	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	4.50	6.11	2.82
▨ Backfilling with Fairness and Slack	4.35	5.66	2.69

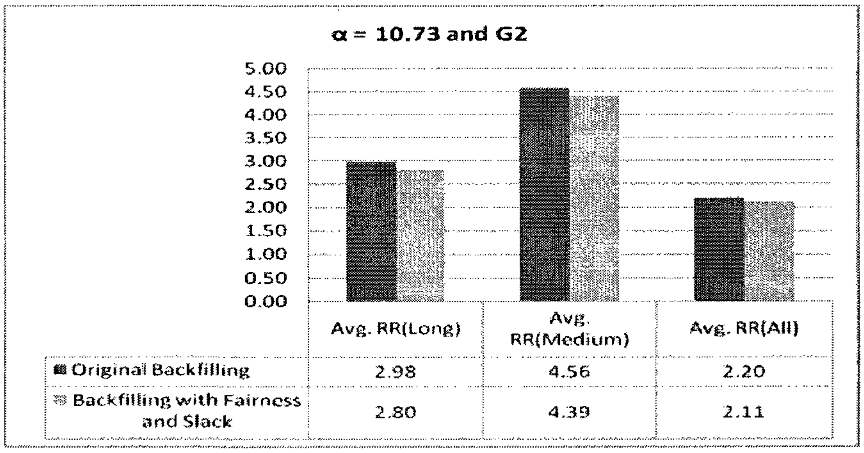
Figure18. RR for original scheduler and backfill with fairness and slack scheduling with α = 10.43 and G2

In the Figure 18, with the same light workload but with G2, the improvement is 3.31% for long jobs, 7.9% for medium jobs, and 4.51% for all jobs.



	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	2.98	4.56	2.20
▨ Backfilling with Fairness and Slack	2.90	4.31	2.13

Figure19. RR for original scheduler and backfill with fairness and slack scheduling with α = 10.73 and G1

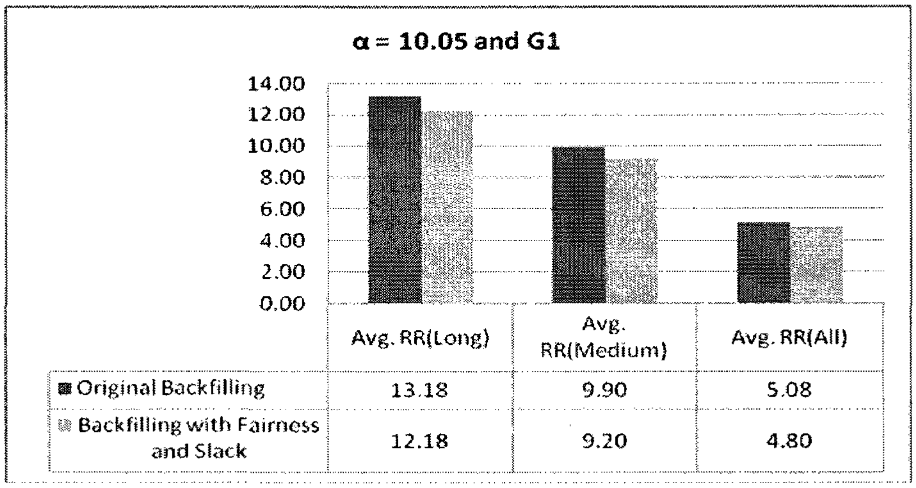


	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	2.98	4.56	2.20
▨ Backfilling with Fairness and Slack	2.80	4.39	2.11

Figure20. RR for original scheduler and backfill with fairness and slack scheduling with α = 10.73 and G2

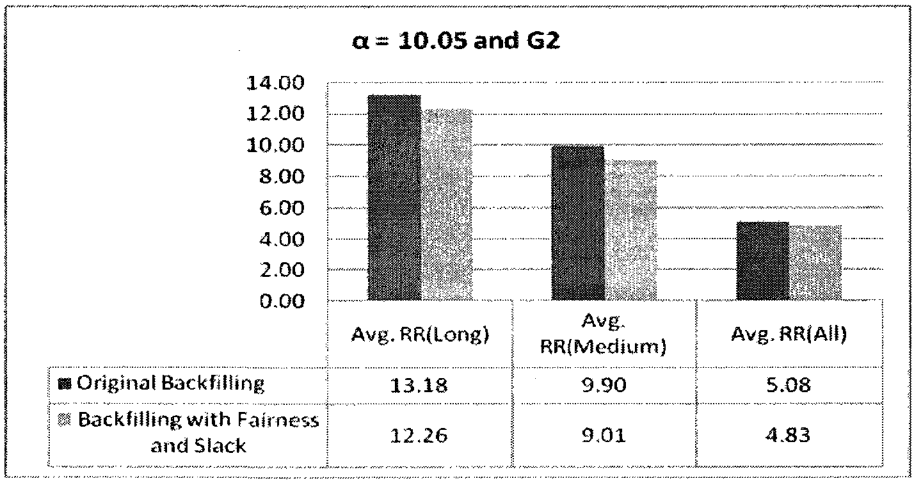
In the Figure 19 and 20, the workload is even lighter with the α equaling 10.73. In this cases, with G1, the average relative response time is improved by 3.12% for long jobs, 7.32% for medium jobs, and 3.78% for all jobs. With G2, the improvement is 4.69% for long jobs, 8.76 % for medium jobs, and 4.76% for all jobs. Comparing these results of light workload with the results of workload W1, the improvement is not as much as workload W1. The reason is probably because there are not enough jobs in the queue, so not so many jobs can be looked further ahead.

The testing results on the very heavy workload W4 are shown in the Figure 21 and 22 as below:



	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	13.18	9.90	5.08
▨ Backfilling with Fairness and Slack	12.18	9.20	4.80

Figure21. RR for original scheduler and backfill with fairness and slack scheduling with α = 10.05 and G1



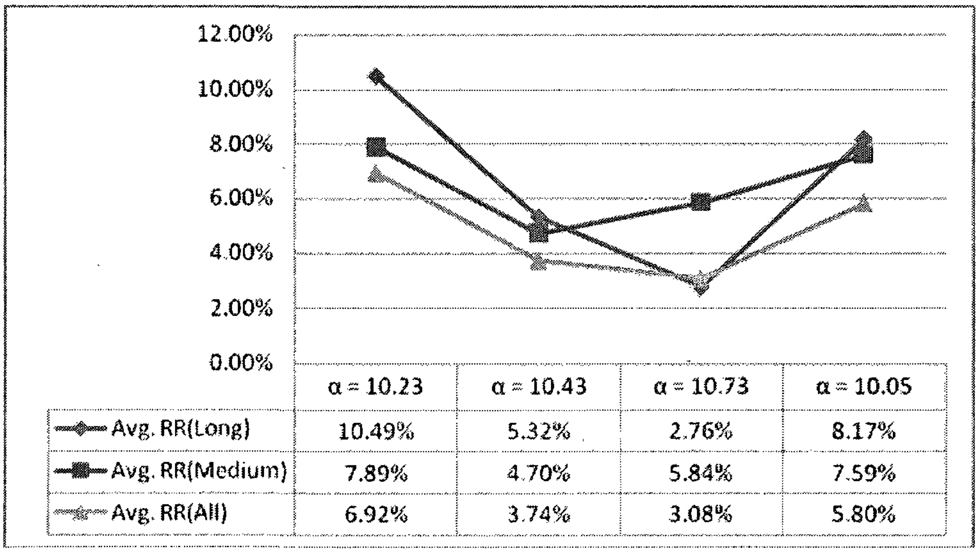
	Avg. RR(Long)	Avg. RR(Medium)	Avg. RR(All)
■ Original Backfilling	13.18	9.90	5.08
▨ Backfilling with Fairness and Slack	12.26	9.01	4.83

Figure22. RR for original scheduler and backfill with fairness and slack scheduling with α = 10.05 and G2

In the Figure 21, compared with original Scojo-PECT, the average relative response time of our backfilling strategy improved 8.17% for long jobs, 7.59% for medium jobs, and 5.8% for all jobs with G1 and the α equaling 10.05. In the Figure 22, with G2, the improvement is 7.48% for long jobs, 9.86% for medium jobs and 5.05% for all jobs.

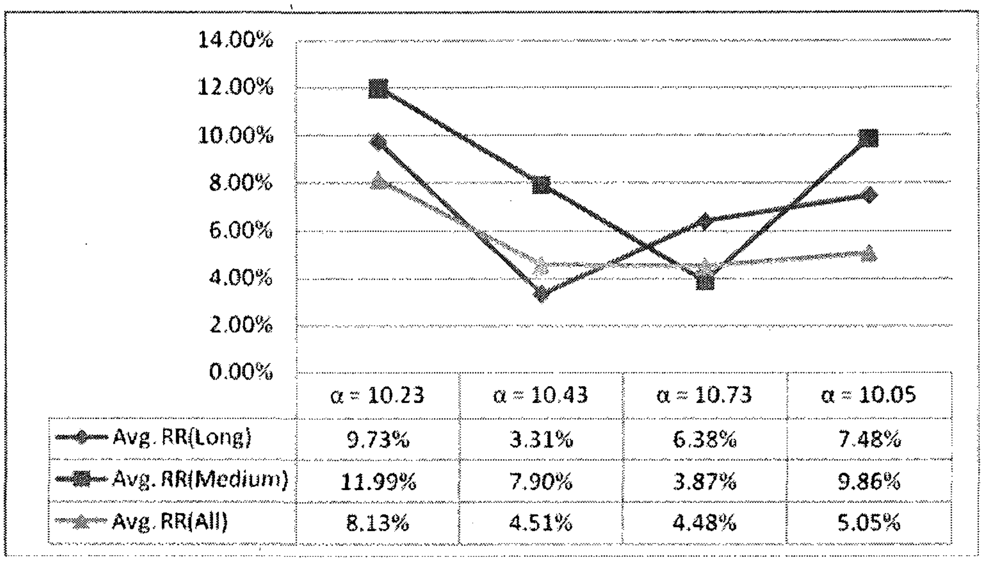
As seen from Figure 17 to 22, the improvement is higher with α equaling 10.05 comparing with α equaling 10.43 or 10.73, which means the higher workload gets better results. The reason is just opposite to the one why no so much improvement when the workload is light. When the workload is heavy, more jobs are created and submitted in the waititng queue, thus more combinations of jobs are tried. So there is a higher chance that a better packing of jobs can be obtained to be backfilled, which improve utilization signicantly and also keep unfairness low.

However, comparing with the high workload W1, light workload W3, and very heavy workload W4 in Figure 23 and 24, we found that it is not the higher workload is, the better performance is.



	α = 10.23	α = 10.43	α = 10.73	α = 10.05
◆ Avg. RR(Long)	10.49%	5.32%	2.76%	8.17%
■ Avg. RR(Medium)	7.89%	4.70%	5.84%	7.59%
★ Avg. RR(All)	6.92%	3.74%	3.08%	5.80%

Figure23. Improvement for different workload for backfilling with fairness and slack scheduling with Goal G1.



	α = 10.23	α = 10.43	α = 10.73	α = 10.05
◆ Avg. RR(Long)	9.73%	3.31%	6.38%	7.48%
■ Avg. RR(Medium)	11.99%	7.90%	3.87%	9.86%
▲ Avg. RR(All)	8.13%	4.51%	4.48%	5.05%

Figure24. Improvement for different workload for backfilling with fairness and slack scheduling with Goal G2.

The results with workload W2 are better than the ones with workload W4. This may mainly because the slack, which defines how much each individual job can be delayed. If there are more jobs are in the waiting queue, then more slack should be satisfied when decide to backfill this packing of jobs or not. Therefore, even though there are a number of jobs in the waiting queue, some combinations of jobs could be rejected. Hence, on the other side, this also proves that lookaheading a number of jobs gets the best results somehow, and there is no need to look even futher ahead.

The average unfairness of all jobs which are treated unfairly under different parameters stays in around 160 seconds which is about 3 minutes and the number is about 800 jobs out of 10000 jobs. In the 800 jobs, there are about 500 jobs are medium jobs. This time represents that the scheduling time of 800 jobs out of 10000 is about 3 minutes later than FCFS scheduling time, and 3 minutes is totally tolerable. The average of skip unfairness of all jobs which are treated preferentially under different parameters is about 20,000 seconds and it is 5 hours and the number is about 50 jobs out of 10000 jobs. The skip unfairness is high in that Scojo-PECT supports non-type safe backfilling, in which jobs can be backfilled into other type of time slices (Chapter 4.4).

Chapter 6: Summary and Conclusion

We have presented the Backfilling with Fairness and Slack for Parallel Jobs Scheduling which incorporates space sharing on coarse-grain preemptive Scojo-PECT scheduler. With space sharing, jobs are sharing the whole resources exclusively. The decision of choosing which jobs to be backfilled is made according to the jobs' slack, jobs' fairness, and utilization gain at current time. Specifically, the thesis has following contributions:

- Every job gets its individual slack factor according to its waiting time. Slack factor has two characteristics:
    a) The slack factor decreases when the estimated waiting time increases. The shorter the time a job is expected to wait, the larger slack factor it will get. In other words, if a job is waiting very long, it cannot afford to be pushed back as much as other jobs which are expected to only wait for a short time. To some extent, this decision is also very fair to all jobs.
    b) The slack factor decreases non-linearly, which means the assigned slack decreases much more with shorter expected waiting time than with longer expected waiting time. (Chapter 4.3.1)
- Each job cannot use all its slack immediately. Therefore, every job can be pushed back many times, instead of only the very first time. How much a job can be pushed back is different based on elements such as how long this job has originally to wait, how long this job has now been waiting, and how much this job's slack factor is.

- This algorithm proposes a modification of [24] to calculate fair start time. It conservatively backfills jobs which are submitted earlier at first, then schedule the next job in FCFS order and estimate its fair start time.
- A number of heuristics are applied when trying to choose packing of jobs since it is an NP-Hard problem of selecting jobs to backfill among multiple waiting jobs.
- When the workload is high, lookahead 20 jobs to backfill is enough. And lookahead 30 jobs is enough for low workload.
- Two metrics are used to control job's fairness:
  a) Assign each individual job a slack. And from the results, it is not the larger the slack factor is the better performance it is. The slack factor in a range less than 1.5 of each individual job's waiting time gets the best results.
  b) For all jobs, average unfairness check is done to avoid serious delay which measures how many jobs are delayed and how many jobs are advanced based on the differences between their fair start time and actual run time (Chapter 4.2). And it is also used to select optimal combination of jobs to backfill with utilization gain at current time.

Backfilling with Fairness and Slack for Parallel Jobs Scheduling is integrated with the coarse-grain preemptive Scojo-PECT scheduler. The experiments show that our scheduler improves average response times by about 10% compared to the original backfilling scheduler which only backfill one job each time and according to their submission order and without considering jobs' slack or fairness.

Chapter 7: Future Work

One of the tasks for future work is to select jobs to backfilling based on the globe improvement instead of current one. From the testing, the decrement of relative response time is not so much as predicted. The main reason is, at each time, the jobs are chosen based on the utilization gain and unfairness at current time. Hence, the jobs that are selected to backfill now are the best. However from the whole progress, they may not, and may affect performance when new jobs are submitted in the future.

From the whole progress, jobs are chosen to backfill into fragment would push back other jobs. New fragments could possibly be created which can also be used to backfill jobs too. If trying to backfill jobs in these new fragments, newer fragments are created again. And this recursive backfilling will continue. Hence, we need to investigate in detail the behavior of this recursive backfilling works, and find the rule to deal with recursive backfilling so that we can always choose the jobs to backfill that can improve the performance globally.

References

[1] A.C. Sodan, "Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey", *Concurrency & Computation: Practice & Experience*, 17(15), Dec. 2005, pp. 1725-1781.

[2] U. Farooq, S. Majumdar, E.W. Parsons, "A Framework to Achieve Guaranteed QoS for Applications and High System Performance in Multi-Institutional Grid Computing", *Proc. Internat. Conf. on Parallel Processing (ICPP)*, 2006.

[3] C. Hu, J. Huai, and T. Wo, "Flexible Resource Reservation Using Slack Time for Service Grid", *Proc. Internat. Conf. on Parallel and Distributed Systems (ICPADS)*, 2006.

[4] N.R. Kaushil, S.M. Figueira, and S.A. Chiappari, "Flexible Time-Windows for Advance Reservation Scheduling", Proc. 14th IEEE Internat. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2006.

[5] A. Mu'alem and D. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," in *IEEE Trans. on Parallel and Distributed Systems*, 12(6), June 2001.

[6] G. Sabin and P. Sadayappan, "Unfairness Metrics for Space-Sharing Parallel Job Schedulers", *Proc. JSSPP*, Cambridge MA, USA, June 2005, Springer, LNCS 3834, pp. 238-256.

[7] E. Shmueli and D.G. Feitelson, "Backfilling with Lookahead to Optimize the Packing of Parallel Jobs", *J. of Parallel and Distributed Computing* 65, 2005, pp. 1090-1107.

[8] A. Sodan and B. Esbaugh, "Service Control and Service Prediction with the Preemptive Parallel Job Scheduler Scojo-PECT", submitted to journal.

[9] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Selective Reservation Strategies for Backfill Job Scheduling", *Proc. JSSPP*, Springer, LNCS 2537, 2002, pp. 55-71.

[10] D. Talby and D. Feitelson, "Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack Based Backfilling", *Proc. IPPS/SPDP*, 1999.

[11] W. Ward, C.L. Mahood, and J.E. West, "Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy", *Proc. JSSPP*, Springer, LNCS 2537, 2002, pp. 88-102.

[12] H. Zhao and R. Sakellariu, "Advance Reservation Policies for Workflows", *Proc. JSSPP 2006*, Springer, LNCS 4376, 2007.

[13] D. Zotkin and P.J. Keleher, "Job-Length Estimation and Performance in Backfilling Schedulers", *Proc. 8th IEEE Internat. Symp. on High Performance Distributed Computing*, August 1999.

[14] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of Backfilling Strategies for Parallel Job Scheduling", *Proc. Internat. Conf. on Parallel Processing Workshops (ICPPW'02)*, 2002, pp. 1530-2016.

[15] D. Jackson, Q. Snell, and M. J. Clement, "Core Algorithms of the maui scheduler", *In JSSPP*, 2001, pp. 87-102.

[16] U. Schwiegelshohn, and R. Yahyapour, "Analysis of First-Come-First-Serve Parallel Job Schedluling", *In Proc. Of the 9th SIAM Symposium on Discrete Algorithms*, 1998.

[17] Z. Liu, and E. Sanlaville, "Preemptive Scheduling with Variable Profile, Precedence Constraints and Due Dates", *Discrete Applied Mathematics*, April 1995, pp. 253-280.

[18] B. Esbaugh, and A.C. Sodan, "Coarse-Grain Time Slicing with Resource-Share Control in Parallel-job Scheduling", *High Performance Computing and Communication (HPCC),* Houston, LNCS 4782, Springer Verlag, Sept. 2007.

[19] U. Lublin, and D.G. Feitelson, "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs", *Journal of Parallel and Distributed Computing*, 2003, pp. 1105-1122.

[20] A.C. Sodan, "Adaptive Scheduling for QoS Virtual-Machines under Different Resource Availability – First Experiences", *Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2009.

[21] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and Practice in Parallel Job Scheduling". *In D. Feitelson and L. Rudolph, editors, 3rd Workshop on Job Scheduling Strategies for Parallel Processing, volume 1291 of LNCS, Springer-Verlag*, 1997, pp. 1-34.

[22] N. Bansal, and M.H. Balter, "Analysisi of SRPT Scheduling: Investigating Unfairness", *ACM SIGMETRICS Performance Evaluation Review*, 2001, pp. 279-290.

[23] G. Sabin, G. Kochhar, and P. Sadayappan, "Job Fairness in Non-Preemptive Job

Scheduling", *Proc. Intern. Conf. on Parallel Proc. IEEE*, 2004.

[24] G. Sabin, V. Sahasrabudhe, "On Fairness in Distributed Job Scheduling Across Multiple Sites", *CLUSTER IEEE*, 2004.

[25] S. Setia, M. Squillante, and V. Naik, "The Impact of Job Memory Requirements on Gang-Scheduling Performance", *Performance Evaluation Review*, vol. 26, no. 4, 1999, pp. 30-39.

[26] Y. Zhang, "Scheduling and Resource Management for Next Generation Clusters", *Ph.D thesis dissertation*, Department of Computer Science and Engineering The Pennsylvania State University, August 2002.

[27] D. G. Feitelson, "A survey of Scheduling in Multiprogrammed Parallel Systems", *Research report rc 19790 (87657),* IBM T. J. Waston Research Center, Yorktown Heights, Second Revision, February 1997.

[28] D. G. Feitelson and A. M. Weil, "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling", *In 12th International Parallel Processing Symposium*, April 1998, pp. 532-546.

[29] D. Lifka, "The ANL/IBM SP Scheduling System", *Proc. Job Scheduling Strategies for Parallel Processing (JSSPP)*, Lecture Notes in Computer Science, Springer Verlag, Vol. 949, 1995.

[30] A. Lodi, S. Martello, and D. Vigo, "Recent Advances on Two-Dimensional Bin Packing Problems", *Discrete Applied Mathematics*, Vol. 123, 2002, pp. 373-390.

[31] A. Lodi, "Algorithm for Two-Dimensional Bin Packing and Assignment Problems—A survey", Vol. 141, 2002, pp. 241-252.

[32] L. Horchani, and M. Bellalouna, "The 2-Dimensional Probabilistic Bin Packing Problem: An Average Case Analysis", *International Journal of Mathematics and Computers in Simulation*, Issue 1, Vol. 2, 2008.

[33] A. C. Sodan, A. Kanavallil, and B. Esbaugh, "Group-Based Optimization for Parallel Job Scheduling with Scojo-PECT-O", *IEEE, Quebec City*, June 2008.

[34] S. Vasupongayya, S. H. Chiang, and B. Massey, "Search-based Job Scheduling for Parallel Computer Workloads", *IEEE, Boston City*, September 2005.

[35] G. Belov and G. Scheithauer, "A Branch-and-Cut-and-Price Algorithm for

One-Dimensional Stock Cutting and Two-Dimensional Two-Stage Cutting", *European Journal of Operational Research,* Vol. 171, May 2006, pp. 85-106.

[36] D. R. Ulm, J. W. Baker and M. C. Scherger, "Solving a 2D Knapsack Problem Using a Hybrid Data-Parallel / Control Style of Computing", Proc. Of The 18th International Parallel and Distributed Processing Symposium (IPDPS), 2004.

[37] P. Bonzon, "Necessary and Sufficient Conditions for Dynamic Programming of Combinatorial Type", *Journal of the ACM (JACM)*, ISSN: 0004-5411, Issue 4, Vol. 17, 1970, pp675-682.

[38] J. Bentley, "Programming Pearls, Second Edition", *Addison-Wesley, Inc.*, ISBN 0-201-65788-0, 2000.

[39] G. Fred, and T. Lee, "Dynamic Strategies for Branch and Bound", *The Int. Journal of Mgmt Sci. (OMEGA). Issue 5*, Vol.4, 1976, pp: 571-576.

[40] K. Brucker, and S. Knust, "Complex Scheduling", *Springer Berlin Heidelberg New York*, ISBN: 978-3-540-29545-7, 2006.

Vita Auctoris

NAME:	Wei Jin
PLACE OF BIRTH:	JiangXi, P.R. China
YEAR OF BIRTH:	1985
EDUCATION:	The 1th High School, FuLiang, JiangXi, China
	1999 – 2002
	JiangXi Normal University, NanChang, China
	2002 – 2006
	University of Windsor, Windsor, Ontario, Canada
	2007 – 2009