2008

Group-based optimization for parallel job scheduling in clusters via heuristic search

Arun Kumar Kanavallil
University of Windsor

Group-based Optimization for Parallel Job Scheduling in Clusters via Heuristic Search

by

Arun Kumar Kanavallil

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

Windsor, Ontario, Canada

2007

Your file Votre référence
ISBN: 978-0-494-42265-6
Our file Notre référence
ISBN: 978-0-494-42265-6

ABSTRACT

Job scheduling for parallel processing typically makes scheduling decisions on a per-job basis due to the dynamic arrival of jobs. Such decision making provides limited options to find globally best schedules. Most research uses off-line optimization which is not realistic. We propose an optimization on the basis of limited-size dynamic job grouping per priority class. We apply heuristic domain-knowledge-based bi-level search and branch-and-bound methods to heavy workload traces to capture good schedules. Special plan-based conservative backfilling and shifting policies are used to augment the search. Our objective is to minimize average relative response times for long and medium job classes, while keeping utilization high. The scheduling algorithm is extended from the SCOJO-PECT coarse-grain pre-emptive time-sharing scheduler. The proposed scheduler was evaluated using real traces and Lublin-Feitelson synthetic workload model. The comparisons were made with the conservative SCOJO-PECT scheduler. The results are promising - the average relative response times were improved by 18-32% while still able to contain the loss of utilization within 2%.

DEDICATION

To

My parents and family members, who guided me through the right path,
And
Friends who gave continuous encouragement

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. A. C. Sodan, for giving me an opportunity to work in a very interesting area, and for her support, guidance and encouragement throughout my graduate studies.

I would also like to thank my committee members, Dr. Chitra, Dr. Arunita and Dr. Tsin for their time and effort and their helpful comments and suggestions.

I would also like to thank my colleagues, Lawrence Barsanti and Bryan Esbaugh for providing the discrete event simulator framework and SCOJO-PECT pre-emptive time shared scheduler.

Debts of gratitude are also due to my uncle (Dr. Nandakumar) and aunt (Dr. Sreekumari) for their constant encouragement. Above all, I thank my parents and my brother Visakh Kanavallil, for their prayers and believing in me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Job scheduling in parallel systems (multi-processors and grids) [1][11][19] is a complex scenario because jobs can simultaneously occupy multiple heterogeneous resources at the same or distinct times. Being a dynamic system, scheduling is more challenging and generally comes with a cost of communication and network delays [31]. Parallel job scheduling problems can also be formulated with several constraints which make them candidates for constrained-based programming. An optimal solution for scheduling problems has been proven to be NP-Hard. With grids (interconnecting clusters and other computing devices through a high speed network) replacing clusters, scheduling demands a fault-tolerant virtual platform for running applications transparently. In addition, grids typically, comprise of multiple schedulers distributed geographically negotiating for different resources like CPU's, disks, licenses, memory etc. With the advent of information age, there has been increasing demand for high processing power. Clusters developed from off-the-shelf components are widely used to carry out data-intensive simulations and compute-intensive tasks. Optimization has a huge role to play in reducing this scheduling complexity.

- From the system administrator's point of view, optimization increases overall system utilization and load balancing.
- From the user's perspective, optimization alleviates excessive wait and faster response times for the submitted jobs.

In parallel job scheduling, typically one employs scheduling in groups [6][16][18], approximation algorithms [23][37], search based methods [10][40][41], and heuristics [4][33] to address the issue of optimization. All these approaches have one thing in common - they intelligently select optimum schedules within reasonable time. In this thesis, we undertake this task of selection using a combination of group interleaving, search methods and heuristics.

The usual metrics for testing the worthiness of a scheduling algorithm are response times (sum of wait time and runtime of the job), runtimes, wait times (duration from job submittal to start time plus additional time spend in the queue when preempted) and system utilization (percentage of machine utilization). The majority of research work accounts overall benefit only *viz.* analyzing average values or makespan (the completion time of the last job).

1.1 Optimization in Job Scheduling

With the right motivation in hand, our scheduling approach interleaves job groups and individual job scheduling on a course-grained, time-sharing, preemptive scheduler. When a sufficient amount of workload gets accumulated in the waiting queue, jobs are grouped based on good packing heuristics. Optimization of job groups is achieved by implementing a tree search that returns an optimal schedule. Our goal is to reduce the average relative response times for medium and long jobs while also not compromising much on utilization and response times. We include the standard approach to backfill jobs from behind the queue which can be started immediately provided they do not delay the execution of grouped jobs.

In the context of parallel job scheduling, proximity to the best solution can also be achieved using approximation methods, search methods, dynamic programming, and genetic programming. We shall discuss each one of them in detail laying emphasis on heuristics and search methodologies in Section 3.3.

2. Related Work

Several approaches [16][29][36] address static (or offline) optimization where information about the jobs are known beforehand and do not consider dynamic submission. The approach in [40] (whose approximation is referred to as bottom-line approach in Section 9.5) optimizes for minimization of average relative response times (slowdown) and total excessive wait time, i.e. the approach considers both average and worst-case behavior as separate crisp criteria, with the worst-case behavior constituting the higher-ranked criterion. To find the best schedule according to this objective, they employ a special search technique based on the largest-slowdown-first heuristic and depth-bound discrepancy. The latter considers optimizing instantaneously by increasing numbers of discrepancies from the heuristic order and bounded by a certain depth. Such optimizations do not follow a plan based approach. Follow-up work [39] includes branch-and-bound techniques and two-step optimization with additional reordering of the obtained schedule for minimizing excessive wait. In the context of job shop scheduling, Pape et.al [27] came up with an idea of combining preemptive scheduling strategies with heuristics to minimize makespan criteria. Basically jobs are divided into activities and each activity is processed by different machines. Priorities decide the activity order which can be formulated as a search tree. Combination of LDS heuristic and edge finding technique was found to provide better average results in terms of mean relative error. Closely related, optimization for backfilling was proposed in [33], considering a group of backfill candidates and using dynamic programming. The primary optimization criterion is utilization (in the space dimension), and the secondary one original queue order.

Among schedules with the same utilization, the one with jobs earlier in the queue is chosen.

There exist a number of approaches which form groups of jobs and schedule them together. Some approaches in grid scheduling allocate jobs to sites according to cost criteria or processing capability and simply group the next jobs sent to a site to reduce transfer cost [25]. This implies that the decision for allocation to a specific site is then fixed for this group. The approach in [11] is slightly more advanced and combine jobs according to size for better fit (packing potential) within the groups that are allocated to each site but miss to consider the variations in runtime which significantly contribute to the packing problem.

Our approach is built on top of the course-grain timesharing scheduler developed by Esbaugh and Sodan [7]. Varying time slices (resource allocation policies) are allotted to different job classes (long, medium and short) enabling multiple virtual machines to execute synchronously. Controlled preemption for long slices, suspending jobs to disk at the end of slice and smart backfilling from other slices resulted in 88% decrease in bounded slowdowns and 31% increase in average response times. Preemption can be approached in two major ways: (a) Migrating jobs to new resources (incurs communication and data transfer costs) was still found to fare well in giving benefits in utilization and response times [28]. However, migration is accompanied by the problem of check-pointing (which is application centric). (b) Secondly, gang scheduling creates global time slices and causes jobs to be preempted at the end of each slice. Gang scheduling has been investigated earlier to provide better average response times and bounded slowdown [22]. However, the drawback is that it requires preempted jobs to be memory-resident during execution. Course-grain timesharing also provided similar benefits as shown in [31] and [42].

Dutot et. al [6] presents an approach for local job scheduling which constitutes a mix of static and dynamic scheduling. The approach additionally considers adaptive job sizes, i.e., jobs can be started with different job sizes which subsequently lead to different runtimes. The jobs that are initially in the waiting queue are grouped according to their runtimes being, under certain sizes, possible within increasingly long time frames. Next, the actual jobs for the next group and their sizes need to be determined. This decision is made by optimizing with objective of priority for maximum sizes (that still let the runtime fit into the interval) and of the best combination of jobs (treating the problem as a knapsack optimization and solving it via dynamic integer programming). Dynamically arriving jobs are considered in the next group to be optimized. However, scalability of jobs is not considered here.

There are many approaches that optimize allocation of groups of serial jobs, e.g., taking the next set of jobs from a FIFO queue and allocating them to different nodes in a grid, while optimizing (via genetic algorithms) the makespan of the group [16][19][21][26]. In [11], a weighted sum of makespan, excess wait time (over deadline), and utilization were used as the objective. In [20], genetic algorithms are combined with different local optimization approaches to optimize both makespan and wait time via Pareto fronts. In some cases, local time sharing per machine is considered. In [4], genetic algorithms applied to groups of jobs are compared to heuristics that make decision on a per-job basis. The relatively simple min-min heuristic (selecting a task-machine pair with the task being completed earliest among all tasks on that machine) was found to perform almost as good (within 12% of the makespan) as the much more costly genetic algorithms.

A different approach is to use heuristics for slack-based backfilling [35] which has the basic idea to permit that jobs are moved backward to some—controllable—extent. Jobs may be flexibly reordered during backfilling to improve utilization, while resulting delays for individual jobs are considered via their slack, i.e. maximum possible delay. However, only average wait time is considered and not the current load on the machine.

In [2], a heuristic is proposed for adaptive size selection that considers the context of all currently waiting and future (predicted) jobs and determines a size that balances the interest of the candidate job with the interests of those other jobs by applying the same relative size modification to all jobs. However, the size decision is finalized only for the candidate job and is newly made for the other jobs when the scheduling decision is made for them. Similarly, [15] considers a balanced workload over all resources CPU, memory, and disk when deciding which job to schedule next and on which most underutilized resources. Other approaches simulate different schedules over all jobs that are currently in the waiting queue and then make the decision about job size, either based on the best response time for the candidate job [5] or the best average response time [30]. However, simulation cannot take future submissions of potentially higher priority jobs into account. Correspondingly, job sizes of running jobs may be adjusted at runtime if jobs are malleable by considering the context of all jobs and giving more resources to jobs with high efficiency [29].

3. Parallel Job Scheduling Optimization Approaches

This section focuses various optimization methodologies prevalent in literature.

3.1 Approximation algorithms

Approximation algorithms are solely designed to contain the time complexity of non-polynomial time (NP-Hard) problems by mathematically proving that a solution close to optimal could be discovered in polynomial time. A performance guarantee (ρ) is always associated with such algorithms and is defined as the ratio of the approximated solution to the optimal one. Alternatively, they are expressed using bounded error ($\varepsilon > 0$) if the approximated solution lies within bound. They are generally applied for optimizing problems where no polynomial time algorithms exist. Best examples are scheduling problems, vertex covering etc. Advantages of approximation algorithms are:

- They are mathematically stable and can be verified
- Such algorithms provide near-optimal (within a specific percentage of optimal) solutions in polynomial time

Formulating a non-preemptive malleable parallel task scheduling (MPTS) as a linear program, Fishkin et.al [8] has shown a polynomial time approximation scheme (PTAS) to achieve a performance bound of $1+\varepsilon$ where ($\varepsilon > 0$) for makespan. Mounie et.al [23] dealt the same problem using dual approximation techniques [6] for generating a non-preemptive schedule. The main contribution of [23] was a worst-case performance guarantee of makespan to be 1.73 as against a guarantee of 2 for non-malleable parallel jobs in [37]. Interestingly, [8] discusses the idea of task grouping using special task profiling and dynamic programming to develop an approximation scheme for unrelated parallel machines.

3.2 Search Metrics

Many search metrics have been proposed and developed for job scheduling. For instance, MET (Minimum Execution Time), MCT (Minimum Completion Time), Min-Min, Max-Min, Suffrage and slack metric. Some of these (MET, MCT, Min-Min, Max-Min) are applied for independent task scheduling [4]. The following are some commonly used metrics:

- Minimum Execution Time (MET) [4]: MET is a commonly used heuristic which assigns each job to the resource that gives the minimum expected execution time for the job, regardless of that machine's availability. Clearly, MET could give really bad wait times on heavy workloads.

- Minimum Completion Time (MCT) [4]: Intuitively MCT performs better than MET and OLB as it strikes a balance between the above two heuristics. By preventing circumstances where OLB and MET are likely to perform poor, MCT heuristic assigns jobs in arbitrary order to the resource with the minimum expected completion time for that job.

- Min-Min [4]: The condition for min-min to work is to have a set of waiting jobs in the beginning. Unlike MCT, Min-Min considers the set of jobs and computes their completion times on all resources. The idea behind min-min is to schedule that job from the job set on the corresponding resource with the best minimum completion time. Eleven static scheduling heuristics using makespan criteria were compared Braun et.al [4] proved Min-min metric to significantly perform better than others.

- Suffrage: As the name suggests, Suffrage heuristic [18] prioritizes or schedules job based on the degree of disturbance caused on the job if its scheduling is delayed. The suffrage value is commonly chosen as the difference between the first and second MCT's. Jobs that suffer more get more priority. X-Suffrage is an extended version where suffrage value also accommodates the state of the system.
- Slack metric: Here the term 'slack' means allowing some degree of compromise per job (measured in units of time) for it to be scheduled later. The slack measure attributed to a job is generally defined as a function of its priority and characteristics (job size, expected runtime, slowdown etc.). First introduced in slack based backfilling [35], the average wait times of priority scheduling were reduced by 15% relative to EASY backfilling.
- Branch-and-Bound (BnB) metric [39]: A monotonically increasing function value is chosen as the upper bound for BnB metric. The metric chosen would be problem specific. Basically intended to prune search trees, BnB provides significant benefits depending on its formulation.

3.3 Search based optimization methods

Since scheduling problems are inherently NP-Hard, search based approaches in scheduling comes handy when search space can be reduced using high-quality problem specific heuristics. Basically, search algorithms can be divided into two: local search and complete (global) search. Local search algorithms foray within its specified neighborhood to select a better solution and advance forward with this new solution. Hill climbing, simulated annealing (SA), ant-colony optimization (ACO), tabu search and iterated local search (ILS) etc. fall under this category. On the other hand, complete search algorithms generate the schedule corresponding to the goal node by a global search. Genetic algorithms, discrepancy search, evolutionary computation are some examples of complete search. Collectively they include (but not restricted to) what can be termed as 'meta-heuristics'. Heuristics have emerged as a powerful strategy to partially circumvent the problems of complexity and intractability. Being intuitive, they have the ability to surpass, judge, predict, or compare different potential solutions. Heuristics in general must be robust, application specific and goal oriented in order to significantly prove beneficial for its purpose. Static scheduling heuristics basically takes scheduling decisions for job(s) after they have been submitted into the waiting queue. On the other hand, dynamic heuristics perform decisions about scheduling, load balancing and resource selection dynamically with no wait. Dynamic heuristics have to be efficient and less complicated to reduce the scheduling overhead. Employing more than one heuristic and dynamically swapping them based on current system states have received new attention recently due to scheduling applications on heterogeneous environments that are themselves dynamic. Some search based heuristics used in job scheduling are:

3.3.1 A* Heuristic:

A* [4] requires the problem to be formulated as a search tree with nodes having a cost function associated with them. Basically the cost function of node *n* is calculated as:

$$f(n) = g(n) + h(n)$$

g(n) represents the cost of scheduling *1..n-1* jobs (node *0* being null) and *h(n)* represents a heuristic estimate of the cost of getting to the goal node. At times, *h(n)* can even be a set of heuristics.

3.3.2 Global Search: Limited Discrepancy Search

Discrepancy based search [40][41] has been in the limelight for solving combinatorial optimization problems. The degree of optimality achieved however heavily depends on the selection of a high-quality heuristic. In its elementary form, discrepancy search can be visualized as taking place in a tree of permutations. Heuristics guide and perpetuate the search to prominent regions. In other words, heuristics bias the search pattern in the following ways:

a) It provides a good initial solution

b) Secondly, it allows some degree of flexibility (or discrepancies, as they are called) to explore other permutations that do not follow the heuristic.

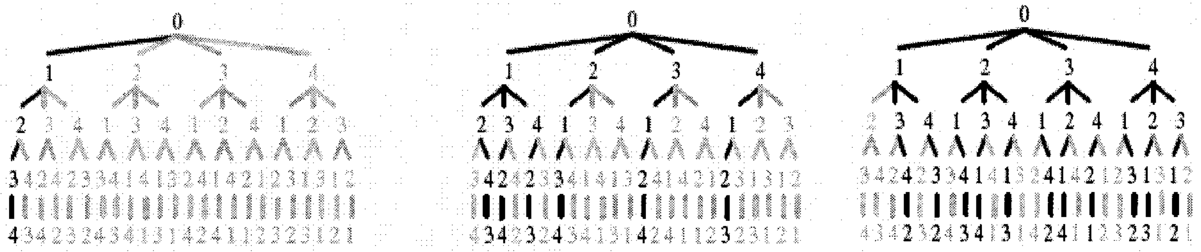
Limited Discrepancy Search (LDS) was first proposed in [10]. The scope of LDS is wide enough that its applications are not limited to scheduling problems. It explores those paths in the search tree that have the lowest discrepancies first. In other words, LDS explores all paths having *k* discrepancies in its k^{th} iteration. The heuristic always aligns the successors from left to right implying that the leftmost path from root to leaf will always completely follow the heuristic. Pseudo code for LDS [10]:

```
LDS-PROBE(node, k)
  if GOAL-P(node) return node
  s ← SUCCESSORS(node)
  if NULL-P(s) return NIL
  if k = 0 return LDS-PROBE(FIRST(s), 0)
  else
      result ← LDS-PROBE(SECOND(s), k − 1)
      if result ≠ NIL return result
      return LDS-PROBE(FIRST(s), k)

LDS(node)
  for x ← 0 to maximum depth
      result ← LDS-PROBE(node, x)
      if result ≠ NIL return result
  return NIL
```

Figure 1 - LDS Psuedocode [10]

Another variant is called Depth-Bound Discrepancy Search (DDS) [41]. Unlike LDS, the k^{th} iteration in the DDS algorithm explores those paths that have a discrepancy at depth k.



(a) (b) (c)

Figure 2 - (a) LDS and DDS: 0th iteration (b) LDS: 1st iteration (c) LDS: 2nd iteration The paths traversed by the LDS algorithm in an arbitrary search tree for 3 successive iterations are shown in dark. [40]

DDS has more built-in flexibility that it is depth-bounded and also simultaneously explores different directions through search space. LDS and DDS traversal paths are illustrated in the figure above. In 0th iteration, only the leftmost path is traversed. For the 1st iteration, all paths having exactly one discrepancy are shown, and so on. A more qualified version of LDS is proposed in [13] called "Improved Limited Discrepancy Search" (ILDS). It is found to reduce the time complexity from O((d+2)/d * 2^d) to O(2^d), where d is the depth of the uniform binary tree. This is achieved by eliminating redundant operation of generating leaf nodes in the tree. Nevertheless, ILDS is less efficient than DFS because it generates multiple interior nodes during its execution. The overhead is deduced to be d/(d-1), where d is the depth of the search tree. YIELDS [12] uses variable ordering scheme of LDS and constraint propagation techniques to solve constraint satisfaction problems. The idea reduces space complexity by minimizing the number of discrepancies to explore during the entire search.

3.4 Knapsack based optimization

Knapsack optimization can be dealt using dynamic programming approach wherein the original problem is sub-divided into sub-problems and then solving each sub-problem. Eventually the solutions are merged to obtain the solution for the original problem. Knapsack selection is largely used for dictating Quality of Service policies [38] and intelligent task selection [6][33]. For instance, [6] uses an intelligent selection of jobs offline using knapsack to form a batch (job group) using this approach. Firstly, the makespan of the current instance is calculated using a dual approximation algorithm. Secondly, jobs that are sufficiently short to be stacked one after the other are grouped into the plan. Lastly, remaining jobs are rightly placed using knapsack selection – we select that set of jobs that maximize their weights and reduce the cost while using at most *m* processors.

$$W(i,j) = max\ (W\ (i\text{-}1,j),\ W(i\text{-}1,\ j\text{-}allot_i) + w_j)$$

where *W(i,j)* recursively denotes the maximum weight of tasks *0..i* being allotted to *j* processors and $allot_i$ is the smallest allocation that fits into the batch. Shumeli et.al [33] used knapsack to select backfilling order of jobs to maximize the machine utilization. The work proposed in [23] differs from the two-phase scheduling approach proposed in [37] in two aspects: (a) allotment selection is ameliorated using a knapsack (b) generating a non-preemptive schedule of non-malleable jobs using dual approximation technique. Turek et. al [37] addresses processor allocation of non-malleable parallel jobs for non-fragmentable multi-processor systems employing shelving algorithms, while fragmented multi-processor systems are dealt separately using heuristics. Though this has been achieved, the significant effect on job scalability and machine utilization remains untouched.

3.5 Genetic algorithms

With their inherent nature to globally classify effective schedules, genetic and evolutionary algorithms are increasingly being applied to scheduling problems. It is more relevant to cite here attributing to its nature to optimize a group of elements. Generally, parallel job scheduling problems are formulated in two different ways to suit the genetic algorithm framework: (a) The jobs are encoded as chromosomes [21][26] (b) The resources are encoded as chromosomes [16]. People prefer (a) when the objective is to obtain a relatively efficient schedule of jobs in terms of response times, slowdown or utilization etc. while (b) is applied in cases where the aim is to maximize resource utilization, maximize throughput [19] or attain adaptation benefits.

Evidently, from the works of [16] and [19], genetic algorithms have also forayed into grid scheduling. For instance in [16], Li et. al presents Predictable and Grouped Genetic Algorithm (PGGA) for m/n scheduling problem (ie. scheduling m jobs on n processors) using grouping and workload prediction. Workload, basically a function of job size, is estimated using regression techniques from the job's execution history. The drawback is – the lesser the historical records of a job, the poorer would be its estimation accuracy. Unlike [19] where the encoding was (job, resource) pairs, [16] use resources to encode their chromosomes. Designing a proper fitness function is critical for the accuracy and efficiency of the GA. Using makespan as the fitness function, the evolution process of mutation and crossing over is again filtered with divisible load theory (DLT) which speeds up the convergence process by allowing populations having similar group finish times to evolve. At times additional heuristics for adaptation or load balancing can be imposed over the GA [26] to improve the solution quality. Meta-heuristics like GA's helps to explore solution spaces to be partially searched in polynomial time to obtain near optimal solutions. By applying evolutionary techniques like crossover and mutation from nature, GA's can prevent themselves from getting caught in local maximum.

4. SCOJO-PECT

SCOJO-PECT [7] is a course-grain timesharing scheduler. The resource shares allocated (done by system administrators based on the ratios of job mix) can be explicitly controlled by setting the machine share distributions per job class for different times of the day. In other words, these shares determined by the share control object gives priority to certain job classes in accordance to the workload present in the waiting queues. The time slices are associated with a particular dominant job type. The scheduling order per slice type is FIFO. Using this approach, short and medium jobs are found to receive relatively good response times. The scheduler maintains separate queues for waiting and preempted jobs for individual job classes. When the allotted time slice gets over, the jobs move to disk and jobs from next job class is scheduled on the machine. This reduces memory pressure all keeps available memory for scheduling the next set of jobs. In brief, the approach entails the following characteristics:

- controlled allocation of varying resource shares for different job classes,
- backfill preempted and waiting jobs from different slices when sufficient nodes are available,
- strictly controlled preemption to disk for jobs in select long-slices to improve response times for medium and short jobs,
- uses a share-based control without priorities to drive the scheduling of jobs

By limiting the time slices to be typically in the minute (or hour) range, the swapping overhead can be reduced. Fragmentation arising from preemption was overcome using smart backfilling techniques. Smart backfilling tries to backfill preempted and waiting jobs of other slice types into the current slice. We shall describe the backfilling with an example. Consider a set of eight distinct jobs as shown in Table 1.

Job ID	**Type**	**Submit Time**	**Runtime**	**Size**
Job 3	MEDIUM	10	65	7
Job 4	MEDIUM	35	34	3
Job 5	SHORT	40	9	7
Job 6	SHORT	41	5	2
Job 7	SHORT	50	5	3
Job 8	LONG	75	98	4
Job 9	LONG	77	80	5
Job 10	MEDIUM	90	40	6

Table 1 - Jobs in a merged waiting queue

In Figure 3, assume a smart-backfilled long job (Job 1) and a short job (Job 2) are running on a machine having a total capacity of 10 nodes (switching overhead between slices not shown). The waiting queue illustrates the job numbers, with their job types, runtimes and job sizes. For instance, Job 3 is a medium job having a runtime of 65 to be executed on 7 nodes. Job 3 is scheduled in the medium slice as neither other medium jobs nor potential backfill jobs are available at the moment. During the next long slice, the preempted long job, Job 1, is scheduled. The medium job, Job 3, is backfilled as other long jobs are currently unavailable. Upcoming short slice would start the short Jobs 5, 6 and 7 to be since there is no room to accommodate non-type slice jobs. Preempted jobs are ensured to start off again on their initial nodes (no migration).

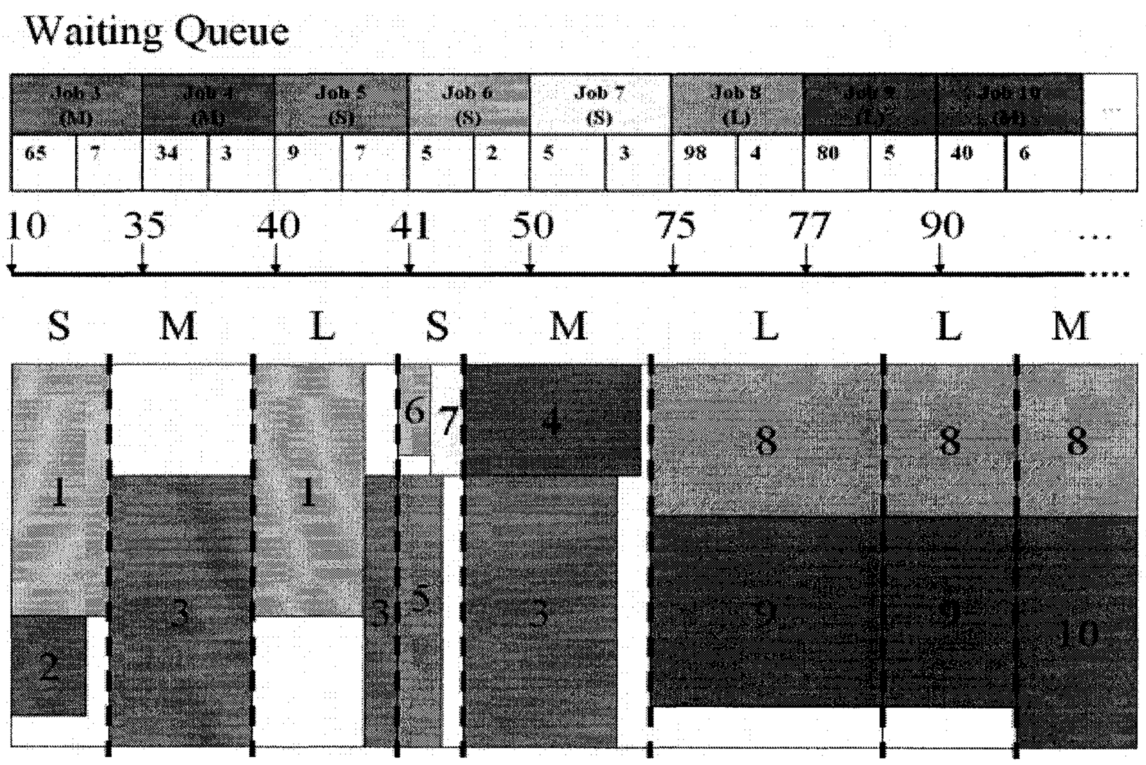


Figure 3 - SCOJO-PECT time slice scheduling example

In the next medium slice, medium preempted Job 3 is scheduled followed by medium job 4. Job 8 starts immediately after job 4 is finished. Job 9 does not have to wait as enough processors are available. From the results in [7], smart-backfilling with the synthetic trace has been able to improve the average response times of short and medium jobs by 27%.

5. Our Approach

We propose a group-based optimization approach for parallel job scheduling using heuristics. Among the different possible approaches for optimization, we have chosen to develop the search-based approach due to its suitability for domain knowledge based heuristics. Employing such optimization in a time-shared environment adds to its novelty. Other approaches mostly applied to parallel job scheduling problems demand formulating an appropriate objective or weighted function which is hard to tune properly. Our search essentially employs a hierarchical structure. We have decided to use high-quality domain knowledge-based heuristics to drive the search to promising regions and then combine the solution obtained from first level to derive a more meaningful second level solution. Integrating smart-backfilling is expected to bring about significant improvements. We make the following assumptions in our work:

- Jobs are rigid (ie. their sizes remain the same as their user submitted size)
- Actual job runtime information is available during job submission. Hence, we would not use user requested runtime estimates.
- Group sizes are limited to a threshold value due to the following reasons:
    - To address the dynamic arrival of jobs
    - Possibility of excessive fragmentation arising from wide jobs
- The size of the search tree is limited. We perform a complete search at the first level incorporating domain knowledge. Second level searching is designed to be partial towards the heuristic and hence would only parse a specific number of schedules as number of job permutations per group grows exponentially.

As workloads can have dynamic behavior, our scheduler is designed to switch between group and individual scheduling modes.

5.1 Concepts Related to Our Work

Explained below is the required conceptual framework behind the design of the scheduler. This is essential to understand the how our approach can really reach its objectives.

5.1.1 Parallel Job Groups

Jobs are classified into three different classes or types on the basis of their runtime information available at submission time: (a) Short, (b) Medium, and, (c) Long.

Groups are formed by jobs of same class. Specifically, only groups of medium and long jobs are created with distinct grouping criteria. Though small jobs are relatively larger in number, from an optimization perspective, options are more for medium and long jobs as they contribute more towards fragmentation and variation in response times. Hence results from optimization would be more evident by focusing on medium and long jobs. We also foresee that backfilling on dynamic submission and smart-backfilling would contribute to bring about significant improvements especially for short jobs.

5.1.2 Domain Knowledge Based Heuristics

Selectively optimizing the long-wide jobs in the beginning is based on the domain-knowledge that such jobs are instrumental in shifting the balance in the opposite direction. Secondly, an incomplete search should be augmented with some good quality heuristic. Goal-oriented heuristics like Largest Slowdown First (LSF) were chosen as jobs with higher slowdown would be scheduled first as such jobs would potentially have to wait for long to get resources. LSF is expected to give an overall improvement in bounded slowdown (defined later in Section 5.1.5). Backfilling techniques, in general, also incorporate domain knowledge into the optimizer and scheduler to improve the results.

5.1.3 Problem Formulation as Search Tree

Consider an offline schedule of jobs. Our aim is to find an efficient scheduling order for these jobs. With the search based approach, the possible schedule orders is visualized as a search tree where a path from the root (excluding root) to leaf corresponds to a schedule. All but root node represent jobs. Each path would have a different job ordering from root to the leaf. At each node the successor ordering is done from left to right with only the left-most branch following the heuristic. A path that comprises a right child is considered as a *discrepancy*. In other words, it is likely that the best schedule may violate the heuristic and may be a path containing multiple discrepancies. A major bottleneck with tree based search is the tree size. Given *n* jobs, the number of possible schedules is *n!* and $O(n^n)$ nodes. The table below shows how the tree grows as the number of nodes (jobs) increases.

#Jobs	#Paths	#Nodes
4	24	64
8	40K	110K
10	3,629K	9,864K
15	1,307,674M	3,554,627M

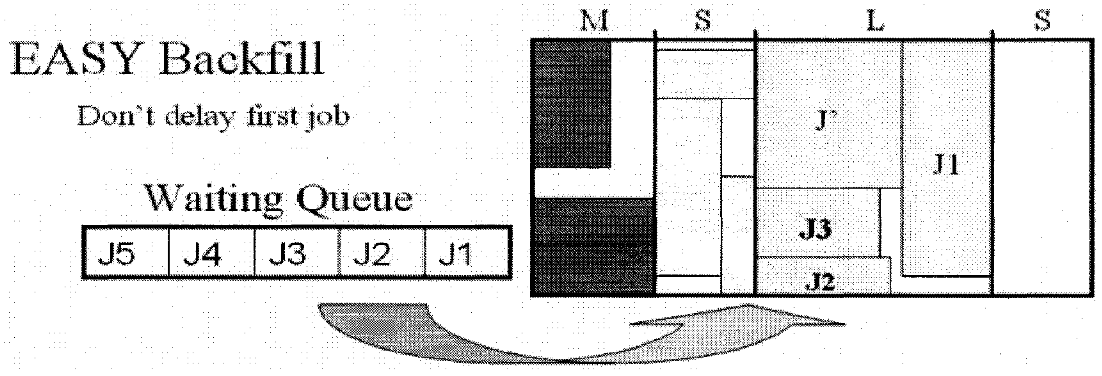
K = 1000; M = 1000,000

Table 2 – Table showing the relationship of jobs in group and nodes generated in the search tree

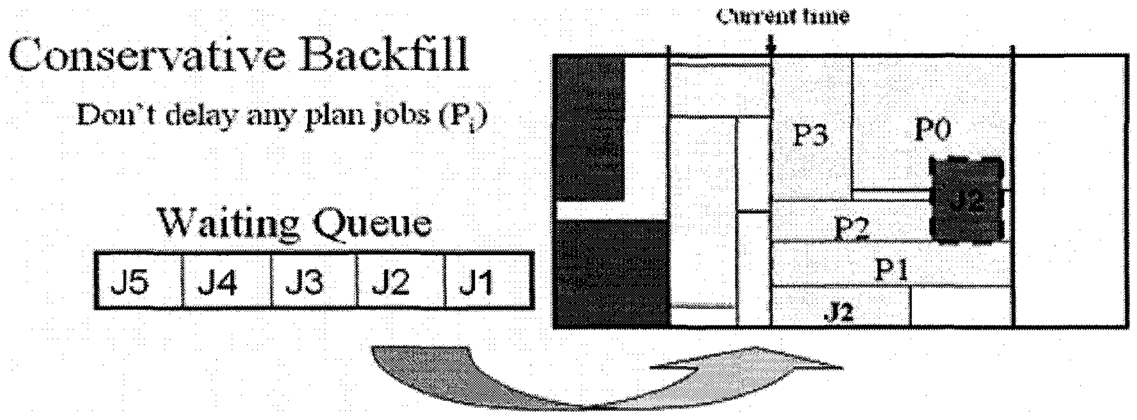
As explained in Section 3.3, the widely known complete discrepancy search algorithms are: (i) Limited Discrepancy Search (LDS), and (ii) Depth-bound Discrepancy Search (DDS).

5.1.4 Backfilling

This is a space-sharing optimization strategy used to bypass the adherence to FCFS scheduling strategy. Backfilling allows a job with lower priority to be scheduled ahead of a job with higher priority if the former does not interfere with the start time of the latter. For EASY backfilling, the shadow time of a job would be the time taken for the first job in the queue to start execution. The idle nodes of the partition in which any job runs are called extra nodes. A job is EASY backfilled if the job size is less than or equal to the currently free nodes and will terminate by the shadow time. A job is conservatively backfilled if the job can be scheduled without delaying any other job in the queue. The graphic below demonstrates the idea of EASY and conservative backfilling. In (b)
Figure 4(a), jobs J2 and J3 get EASY backfilled. In (b)
Figure 4(b), job J2 is conservatively backfilled into the prepared plan.



(a)



(b)

Figure 4 - (a) EASY backfilling (b) Conservative backfilling

Commonly employed in almost all schedulers, backfilling essentially tries to fill up an empty scheduling hole disregarding the specified scheduling order. In other words, backfilling selects jobs from behind the queue and schedules them if this action does not delay the start of other jobs. Different variants of backfilling exists like EASY or aggressive backfilling [34] (stipulates that a job can be backfilled if it does not delay the first job in the waiting queue), conservative backfilling (stipulates that a job can be backfilled if it does not delay any job in the waiting queue), slack-based backfilling (stipulates that a job can be backfilled as long as the other job's slack value remains less than its expected wait time) etc. Though both EASY and conservative approaches are found to produce the same utilization the latter removes the unbounded waiting time that may happen in case of the former.

In our context, smart backfilling would mean scheduling non-type jobs (jobs not of the same type as currently executing slice type) into the current slice if sufficient nodes are available. As the results from [7] prove, smart backfilling is expected to bring good improvements.

5.1.5 Metrics

- *Average Response time:* The time interval from the actual job submission by the user to the complete termination of the job is called the response time. Suppose there are N tasks and a task i is submitted on time t_i and terminates on time h_i. The response time for the task i is $(t_i + h_i)$. The average response time would be the average of all the tasks. Mathematically, it can be written as $1/N$ $(\Sigma(t_i + h_i))$, for $i = 1..N$. From the user's perspective, it is the response time that is crucial to be minimized. It is otherwise also called the flow time.
- *Bounded Slowdown:* Bounded slowdown of a job is defined mathematically as:

  Bounded Slowdown = Ceil(Max(Response time/Max(Runtime,BOUND)),1)

  where, Response time is the sum of partial response times of a job (in case of preemption) or the normal response time and Max(Runtime, BOUND) returns the maximum among its two arguments. The second argument of Max is a threshold that is used to limit the influence of short jobs on the slowdown. Ceil would return the nearest integer greater than equal to the argument.
- *Makespan:* In the context of groups, the finishing time of the last job or the total termination time of all jobs is called the makespan. Minimizing makespan may be an objective of scheduling depending on the domain we are working with. For instance, consider a set of N tasks. Finding a minimum makespan problem is NP-Hard.

- *Utilization:* The percentage of machine (a cluster in our case) that is involved in actual work is called utilization. Mathematically speaking, it is the ratio of work done to total work possible within a specified time interval is called utilization. Likely causes for low utilization are fragmentation and improper scheduling policy.

6. Scheduler Design

The optimization module of the simulator has three main components – group creation module, job plan module and search tree creation module. The block diagram of the optimization module is shown below. In the following sub-sections we will be describing each of them in detail.

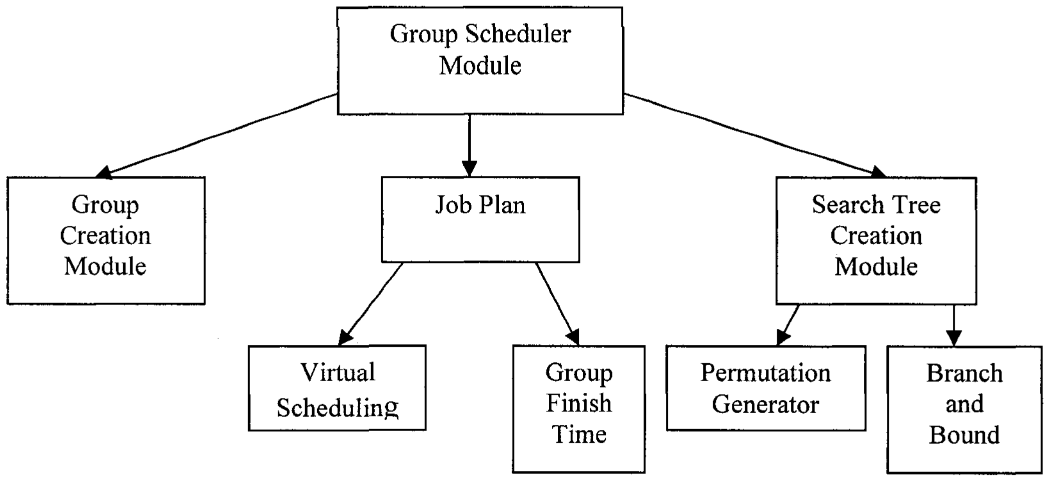


Figure 5 - Block Diagram of Group Scheduler

6.1 Job Group Creation

Limited size groups are allowed to form as the workload becomes heavy ie. when jobs begin to get accumulated in the waiting queue. The criterion for group formation is based on the work done by the jobs, their average runtime and total size. This makes sense as optimization can be more effective when we have sufficient number of jobs at hand. A group is said to have formed when the following criteria are satisfied:

- Minimum number of jobs gets accumulated in the waiting queue. The number of jobs depends upon the type of currently running slice.

- The jobs are also required to occupy more than the machine size to ensure that there is room for optimization.
- The ideal average runtime of the grouped jobs is lower than a factor times the set runtime for the type of current slice.

6.2 Scheduling Algorithm

The scheduling algorithm below (Figure 6) is straight forward and is implemented as part of the group scheduler object. The algorithm switches between groups and normal time shared SCOJO-PECT scheduling. The algorithm is framed such that no two groups of the same slice types can be formed simultaneously. Broadly the scheduling can be divided into two categories: scheduling within a group (ie. when a plan exists) and otherwise. In the former case, the algorithm does the following steps in the order specified:

- Always starts the previously pre-empted jobs on the same nodes
- Start the plan jobs when their schedule times are reached
- Try backfilling pre-empted, waiting and plan jobs from other slices into the group

When not in a group, the algorithm would follow the steps in the order specified:

- Start the previously pre-empted jobs on the same nodes
- Try backfilling jobs from plan into the current slice
- Try backfilling pre-empted, waiting and plan jobs from other slices into the group

An additional check for backfilling plan jobs has been implemented. The implementation details are specified in Section 7.

```
if(currentState == STATE.NONE)
    updateSliceTimes() //get new slice times from share controller

if(SubmitEvent && inGroup(currentState))
    try dynamicBackfill_intoPlan(job) //backfills job into current slice

if(SubmitEvent && !inGroup(Med_OR_Long_State) || (FinishEvent && currentTime ==
plan[currentState].groupFinishTime) || plan[currentState].size == 0) {

    jobgroup = createJobGroup()
    if(jobgroup!=null) {
        plan = Optimize()
        if(plan!=null) inGroup(currentState) = true
    }
}
if(inGroup(currentState)) { // when executing inside a group
    if(BeginSliceEvent) {

        // start previously preempted jobs on the same nodes in current slice
        for(job : preemptedQueue[currentState])
            scheduleJob(job)
    }

    recalculate_plan() //recalculate the schedule times for jobs in plan

    planjobs = plan[currentState].getNextJobs(currentTime)

    /* start plan jobs that start at current  time */
    for(job : planjobs) {
        if(schedulableJob(job))
            scheduleJob(job)
        else break
    }

    // try starting waiting jobs of same type
    for(job : waitingQueues[currentState]) {
        if(schedulableJob(job))
            scheduleJob(job)
    }

    // try to smart backfill preempted jobs on same nodes into the current plan slice
    for(queue : preemptionQueues) {
        if(!queue[currentState])
            if(noConflictWithPlan())
                backfillNonTypePreemptedJobs_intoPlan(jobPreemptionQueues[i]);
    }
    //tries backfill from plans of other slice
    backfillPlanjobs(plan);

    // try to smart backfill waiting jobs into the current plan slice
    for(queue : waitingQueues) {
        if(!queue[currentState])
            if(noConflictWithPlan())
                backfillNonTypeWaitingJobs_intoPlan(jobWaitingQueues[i]);
```

```
else { // normal preemptive scheduling when not within a group
    if(BeginSliceEvent) {
        for(job : preemptedQueues[currentState]) // start previously preempted jobs
            if(schedulableJob(job))
                scheduleJob(job)
    }

    for(job : waitingQueues[currentState])  // try starting jobs from waiting queue
        if(schedulableJob(job))
            scheduleJob(job)

    tryEasyBackfill(currentState)

    /* SMART backfilling */
    for(queue in preemptionQueues) {    // sorted by increasing runtime class
        for(job in queue)
            if(jobFits() && noCollisionwithCurrentSlice(job))
                scheduleJob(job)
    }

    for(job in other plan) {                    // try to backfill jobs from plan
        if(jobFits() && noCollisionwithCurrentSlice(job))
            scheduleJob(job)
    }

    for(queue in waitingQueues) {       // sorted by increasing runtime class
        for(job in queue)
            if(jobFits() && noCollisionwithCurrentSlice(job))
                scheduleJob(job)
    }
}
```

Figure 6 - Scheduling algorithm

6.3 *Job Plan Creation*

A job plan (otherwise referred as 'plan') in our definition is a scheduling frame that essentially does a virtual mapping of jobs in the current group to the processors for the duration between group start and group finish times. Figure 7 illustrates the steps until this point. The region between the two dark arrows comprises the plan. Evidently, plan can consist of jobs in the current group as well as previously executing jobs. It is important to observe that group finish times could also coincide with the finish times of previously running jobs. In order to account for this possibility, the running jobs are added to the plan and later removed after computing the group's finish time. Note that the figure shown below does not involve time sharing. Scheduling from plan in a time shared environment is performed using an additional step – recalculating the plan (see Section 6.3.2).

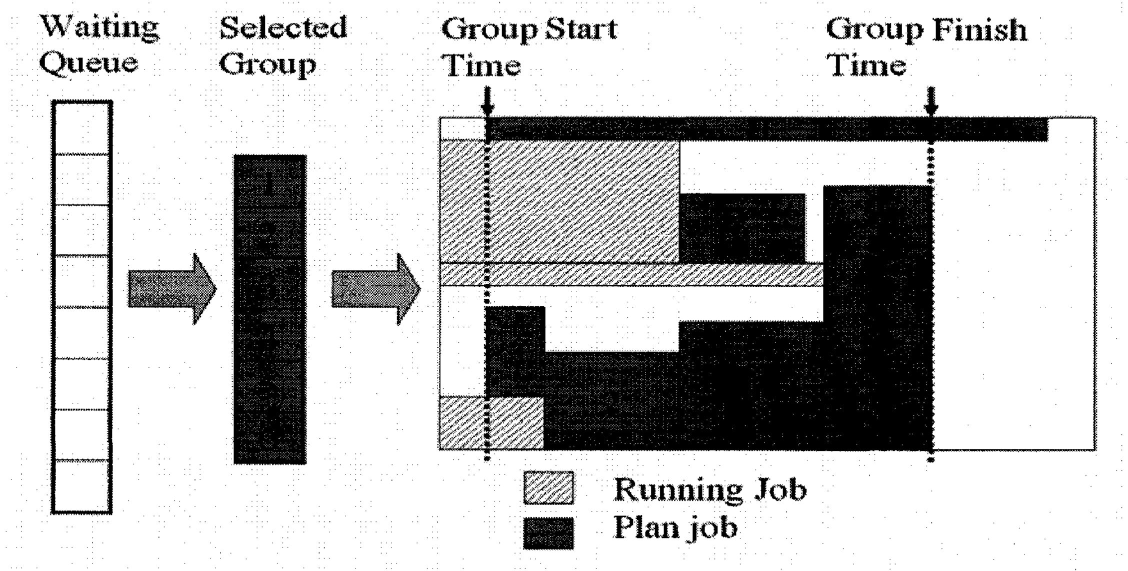


Figure 7 - Group selection and mapping the plan

6.3.1 Virtual Scheduling

A virtual scheduling part is an important module of plan creation. Virtual scheduling is nothing but an intermediary scheduling step performed with a copy of running queues and other scheduling parameters like free nodes and current time. This helps in determining the schedule time (or start time) of individual jobs for each permutation in the group. Whenever the event corresponding to the group start time is encountered, we switch from normal scheduling to plan scheduling and jobs that have their schedule time set to the current time are scheduled. In order to avoid redundant execution of the same jobs, they are then removed from the waiting queue.

This also includes storing information about the unused nodes. The nodes as well as the duration for which they remain unused are saved in a vector. This information is useful for determining jobs eligible for backfilling.

6.3.2 Plan recalculation

In order to deal with intricacies within SCOJO-PECT (time sharing and smart-backfilling), a recalculation of the plan is necessary. Recalculation is done at each point of time do update the schedule times of the jobs in the plan. These jobs are virtually scheduled again with the currently running jobs in the machine. Once the jobs are checked for successful execution until the end of slice they are started. This is essential as the smart backfilling can cause some jobs to finish early which in turn would affect the schedule times. A future option is to provide a re-optimization at this point of time.

6.3.3 Determining Group Finish Times

Setting group finish time is critical to the scheduling algorithm as certain schedules can

```
MAXUTIL = Integer.MINVALUE;
initialUtil = getCurrentGroupUtil(start, current);
for each job scheduled in plan { /* jobs in plan ordered in ascending order of finish times */
   util[i] = getCurrentGroupUtil(start, finishTime); /*machine utilization after each job gets
scheduled*/
}
totalUtil = getCurrentGroupUtil(start, finish); /* utilization from group start to end */
for all values of util {
   if ( minjobs() ) {
       if(util[i] ≥ MAXUTIL * ( 1 - delta)) { /*allowing small variations in utilization*/
           MAXUTIL = util[i];
           set group finish time as current job finish time
       }
     else if (util[i] > getminUtil())
       set group finish time as current job finish time /* ie. guarantee minimum jobs and good
       utilization*/
   }
}
```

Figure 8 - Algorithm for computing group finish times

dramatically degrade the group's system utilization. We had to come up with a strategy that would ensure a minimum number of jobs get scheduled and utilization remains reasonably high within the group. The algorithm (Figure 8) takes these into consideration. In the first place, we sort the running and plan jobs according to their finish times. This helps us in calculating the utilization from the group start time to the finish time as a contribution of each job.

In Figure 9, group finish time is set to be the termination time of job 5. This makes sense as the utilization is high enough within the group. The 'delta' value allows some degree of variation in utilization to happen as illustrated in Figure 9. The utilization plot in Figure 9(b) shows the utilization drop happening after job 5 has finished. This prompts us to decide job 5 to be group finish job.

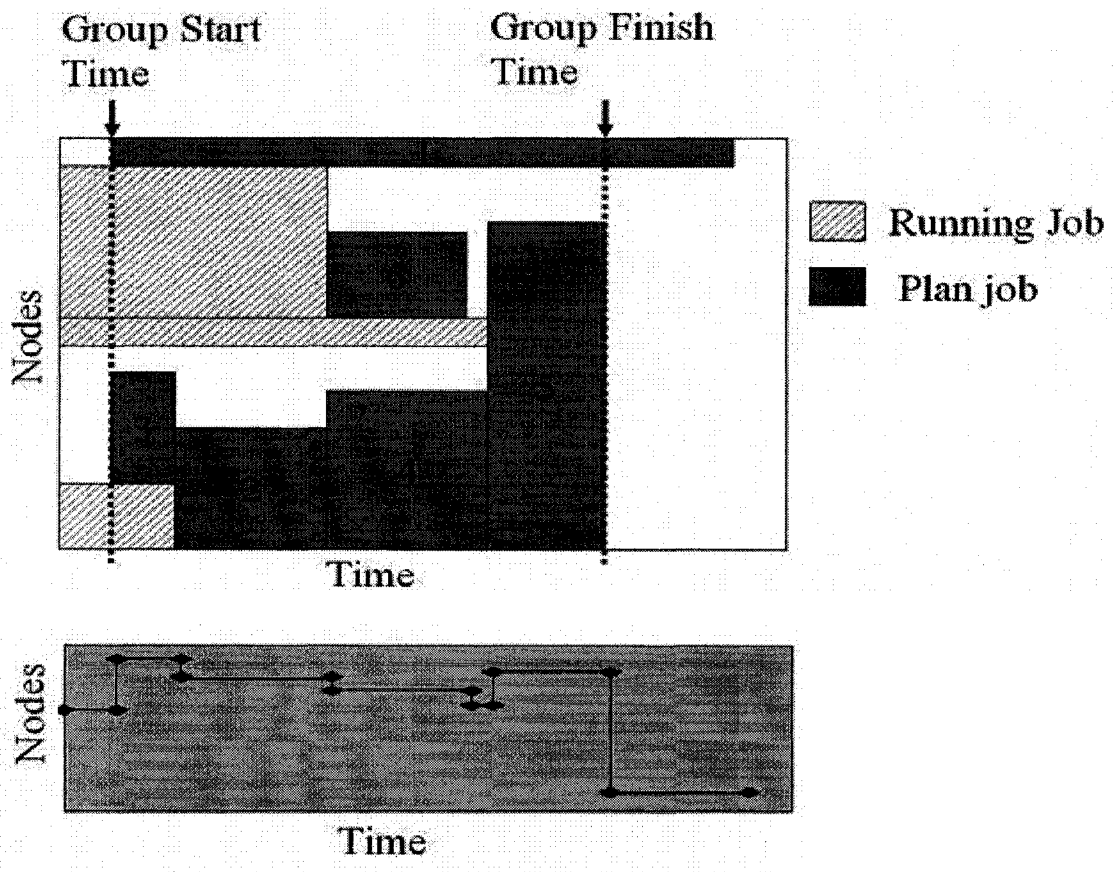


Figure 9 - (a) A sample plan after each job is virtually scheduled (b) Plot of utilization for the plan

If utilization decreases at a drastic rate, we check if the current value still remains above a certain threshold (minUtil). We eventually assign the current job termination time as the group finish time. In Figure 9, the sudden dip in the curve decreases the utilization drastically and hence we set the group finish to the termination of job 5. Though the group finish time is calculated, the actual job corresponding to the group finish may end earlier. If the group-finish job ends up completing its execution in a non-type slice, we assert that the group would be finished upon encountering the next finish event within the group slice. This was needed due to the following reasons:

- The job corresponding to the group finish time may end up finishing in other slice.
- Enable more groups to be formed as the algorithm without this condition would reduce duration of in-group execution.

6.4 Search Tree Creation

There are basically three core components in this module: (a) Group splitting (b) Scheduling order is created using a permutation generator, and (c) Branch-and-Bound design.

6.4.1 Group Splitting

Once the group is formed, long-wide jobs are optimized first. Long-wide jobs are specifically chosen as slowdown and utilization are likely worst affected by the contribution of these jobs. A group is equally split into limited size sub-groups if the original group size is equal to the maximum specified group size. These sub-groups are allowed to permute themselves and the different orderings create the search tree. As the complexity of determining good global schedules is exponential in time, a hierarchical search would enable finding locally good schedules. These schedules are later on merged and optimized separately to form a globally good schedule based on the criteria described later in Section 6.4.4.

6.4.2 Permutation Generator

This module basically generates the permutation of the jobs (or job ordering) of the jobs in the group. The discrepancy is calculated apriori on each iteration. The discrepancy values and schedules follow a many-to-one relationship. The discrepancies are generated in the depth-first search order that that can be obtained from the search tree discussed in Section 3.3. The sequence follows the order of generating the discrepancy in a search tree. For instance, the discrepancy starts off with 0, 1, 1, 2, 1, 2, etc. The reason to have such a sequence of discrepancies was to allow sufficient re-orderings among the jobs so as to bring about significant differences between consecutively generated schedules. Only schedules that have discrepancy value above a threshold are virtually scheduled and inserted into the search tree. Furthermore we allow alternate schedules to be generated from the same discrepancy as a result of force-shifting (see Section 6.5). This can be visualized as a kind of force-backfilling where jobs in the virtual plan are backfilled even if the incoming job could not be accommodated within a hole. The criteria to decide shifting is based on the ratio of the length of the hole to the remaining runtime of the job.

6.4.3 Branch-and-Bound design

Branch-and-Bound has much significance owing to its ability to prune irrelevant regions (or sub-trees) during search. Moreover, pruning can reduce the amount of memory required. In our optimization problem, discrepancy based search can be improved by devising a suitable metric to cut-off certain branches from exploring. A monotonically increasing function of the initial schedule can be used as the upper bound for branch-and-bound metric. We explore paths that have total metric less than current best path. Total slowdown of jobs in group would be appropriate criteria that would match our objective. If the metric stays within the current best value, we virtually schedule each job after inserting them into their appropriate positions in the tree. Additional pruning can be made possible using slack. Slack value could be employed to discard certain schedules from the search tree.

6.4.4 Best Schedule Selection

Once all the different permutations have been successfully scheduled, they are sorted in increasing total slowdown order. The schedules having the least three slowdown values are selected. The best amongst the three is selected based on a weighted function. Consider two schedules 'a' and 'b'. Schedule 'a' would be chosen against 'b' if the following condition is satisfied.

$(M_a > \alpha * M_b)$ && $(U_a > \beta * U_b)$ && $(AvgRT_a < \gamma * AvgRT_b)$; where,

M_a, M_b – Makespan of schedules 'a' and 'b',

U_a, U_b – Utilization of schedules 'a' and 'b',

$AvgRT_a, AvgRT_b$ – Average response times of schedules 'a' and 'b', and,

α, β, γ – constants (values are specified in Section 7.5)

6.5 Search Algorithm for Optimization

The search takes place in two levels as shown in Figure 10 – (a) permuting all combinations of the selected long-wide jobs in group (b) searching a limited number of remaining jobs for sub-groups and finally when merged together. Long-wide jobs are limited in number so as to parse through all scheduling combinations. If sufficient numbers of long-wide jobs are not available, the algorithm proceeds to check whether the group is large enough to be equally partitioned.

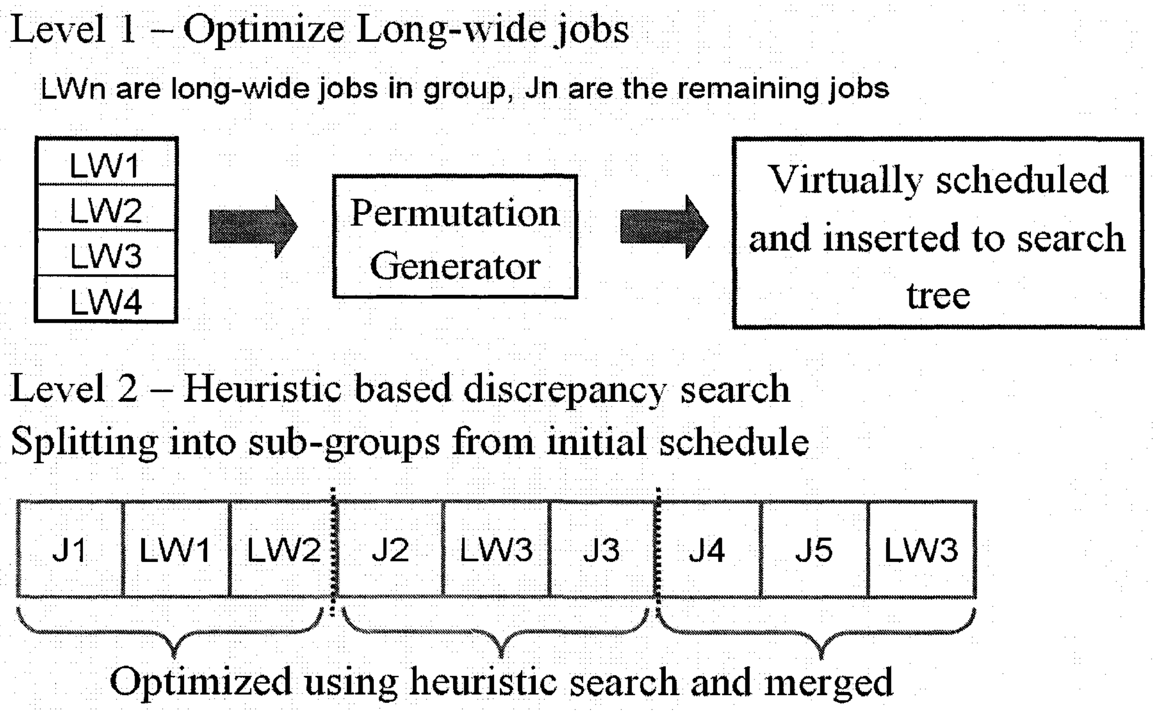

Level 1 – Optimize Long-wide jobs
LWn are long-wide jobs in group, Jn are the remaining jobs
LW1
LW2
LW3
LW4
Permutation Generator
Virtually scheduled and inserted to search tree
Level 2 – Heuristic based discrepancy search
Splitting into sub-groups from initial schedule
J1
LW1
LW2
J2
LW3
J3
J4
J5
LW3
Optimized using heuristic search and merged

Figure 10 - Hierarchical search design

The search algorithm starts off with the heuristic generating the initial schedule. The subsequent job orderings are created by the permutation generator. Splitting into sub-groups would experience the heuristic based search. If all the jobs in a schedule are scheduled without branching, the schedule is added to a separate vector and is potential candidate for the best schedule. The search algorithm is shown below in Figure 11.

```
Sort the limited number of long-wide jobs
Permute all combinations to find the best schedule of long-wide jobs

for each job in the current group {
    for each job in the current group {
        Virtually schedule without updating freenodes and current time
    }
    Virtually schedule the next long-wide job from the best schedule without updating
    freenodes and current time
    nextJ[] = heuristic.getNextJob(); /*heuristic would return the next job*/
    Virtually schedule the job
}


/*At this point, you have the initial job order from the heuristic saved in nextJ[]*/
if(nextJ.length == Constants.MAXGROUPSIZE) {
   Split the group equally to form sub-groups
}
else {
   The group itself is the sub-group
   Sort limited jobs according to secondary heuristic
}
for each sub-group {
    sTree = Create new Search Tree per sub-group
    Initialize the permutation generator with the sub-group job order
    while permutations does not exceed the maximum specified schedule limit {
        Compute the discrepancy of the next permutation
        Create new plan for new job ordering /*starts reordering from the first job*/
        Generate the job permutation
        for job j in permutation {
            branched = Create tree nodes for j and insert them in the search tree
            /*the above step comprise of virtually scheduling the job into a temporary
            Plan and computing the schedule*/
        }
        // try shifting jobs to form alternate schedule
        for job j in permutation {
            if( tryForceShifting(j) ) {
                newJobOrder = createNewJobOrder()
                restarts loop with the new shifted order
            }
            else {
                branched = Create nodes for each job and insert them in the search tree
                /*the above step comprise of virtually scheduling the job into a
                temporary plan, and computing the schedule*/
            }
        }
        Count the total number of nodes pruned with BnB
    }
}
```

```
Bsched = sTree.getBestSchedule(); //selects the best schedule amongst the candidates

for each job j in Bsched {
      Virtually schedule j into the plan
}
plan.computeGroupFinishTime(); // computes the group finish time of the current group
return;
```

Figure 11 - Search algorithm used for optimization

Force-shifting plays an important role in generating alternate schedules. It works as depicted in Figure 12. Job J3, with a runtime 'x', is inserted into the hole, with a width 'y', if the ratio of 'x' to 'y' is less than a threshold value. Job J3 would push other jobs scheduled after it to produce a new schedule.

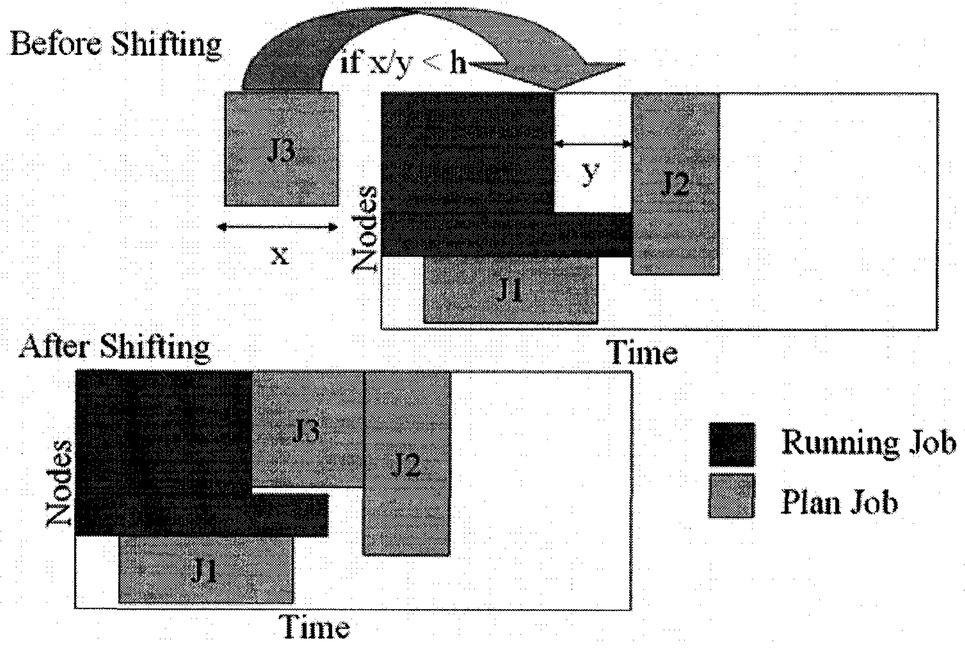


Figure 12 - Force-shifting plan jobs

6.6 Backfilling

We have used conservative backfilling within the group. Apart from these, smart backfilling is also used. Conservative backfilling has been implemented in different ways as described in the Sections 6.6.2 and 6.6.3.

6.6.1 Smart Backfilling within groups

This feature has been integrated with the SCOJO-PECT scheduler. Its implementation deviates from the original when trying to smart backfill jobs within a group. Conflict resolution with the plan has to be checked before starting the jobs so that jobs to be scheduled from the plan remains undisturbed. The jobs are scheduled right away if sufficient nodes are free. While pre-empted jobs runs on the same nodes, waiting jobs start on excluded resources to make sure that they do not conflict with the resources of the preempted jobs of the same slice type. The latter condition is essential to maintain continuous execution of the jobs in pre-emption queue.

Our search strategy combined with backfilling (whose implementation is discussed below) is designed to deliver good system utilization. Consider the set of eight distinct jobs in Table 3.

Job ID/ Attributes	**Type**	**Submit Time**	**Runtime**	**Size**
Job 8	MEDIUM	10	65	5
Job 9	MEDIUM	35	40	3
Job 10	MEDIUM	40	30	6
Job 11	MEDIUM	41	70	7
Job 12	MEDIUM	50	50	5
Job 13	SHORT	75	15	4
Job 14	LONG	77	80	3
Job 15	MEDIUM	90	15	6

Table 3 – A typical example of smart backfilling within group

Figure 13 demonstrates the scheduling happening within a group. As jobs can be backfilled into other slices, the possibility of jobs getting finished earlier is addressed by recalculating the plan on each event. For simplicity, let us assume that groups are formed when more than five jobs of same type are waiting in the queue. The events happen chronologically from left to right as shown by the time advancement below waiting queue. The letters 'L', 'S', 'M' denotes the long, short and medium slice respectively. The dotted line signifies the end of slice. The jobs of same job classes have similar shades. Normal SCOJO scheduling happens until the end of the second small slice. Though we have three medium jobs (Jobs 8,9 and 10) at time 41, Jobs 8 and 9 cannot be backfilled into the second long slice (even when sufficient nodes are available) as those nodes are occupied by Job 1. Newly scheduled job always starts in nodes not occupied by the pre-empted jobs of the same slice type. When the time advances to 41, we have five medium jobs (Jobs 8-12) which can be grouped together for optimization. Once the optimized scheduling order is generated, the jobs would try to stick to the plan unless backfilled. Say the group ends when job 15 gets finished. The case of jobs 1 and 14 are important to observe as they are getting backfilled into the group slice. In such a scenario, we have to ensure that the job starts on the same nodes and the optimized plan is not disturbed. Short job 13 is an example of dynamic arrival inside a group. As sufficient processors are available, it is backfilled and started immediately.

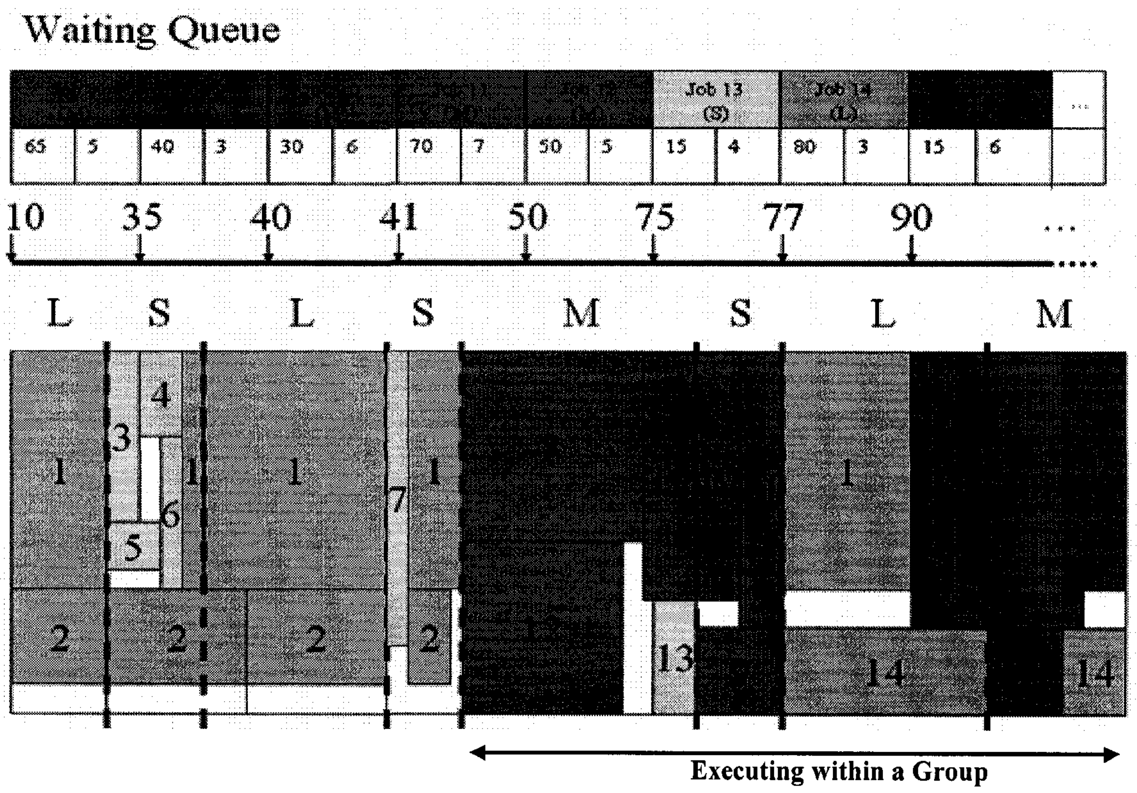


Figure 13 - Time slices and backfilling / scheduling within a group

Job 11 despite being a medium job continues executing (smart backfilled) in the next long slice as nodes remain free after Job 1 terminates. The group finishes once Job 15 gets terminated, and we begin normal SCOJO-PECT execution.

From the above example, we see that backfilling is performed at the following instances:

- Within a group until the end of current slice
    - Backfill preempted jobs of other slice types when no conflict with plan
    - Backfill plan jobs from other plans to the current slice
    - Backfill waiting jobs of other slice types when no conflict with plan
- Outside a group
    - Backfill preempted jobs of other slice types
    - Backfill jobs from plan
    - Backfill waiting jobs of other slice types

6.6.2 Backfilling upon Dynamic Submission

Irrespective of the currently running slice, individual dynamic job submission events within the group scheduling frame may have potential chances to be backfilled into the plan. If such jobs returns positive upon checking for conflict resolution with plan within the current slice, they are scheduled at the time when available resources are free. Dynamic jobs are guaranteed to run within the group only until the end of slice or until it finishes (whichever happens first). In the former case, the job would finish earlier and gets preempted to its respective preemption queue. This feature is particularly expected to enhance the response times of medium and short jobs as they are potential candidates to be plugged into a medium or long group. The backfilled job runs in excluded resources so that it does not conflict with the pre-empted jobs of its own slice type.

6.6.3 Plan Backfilling

Plan backfilling follows a conservative approach as discussed in Section 5.1.4.

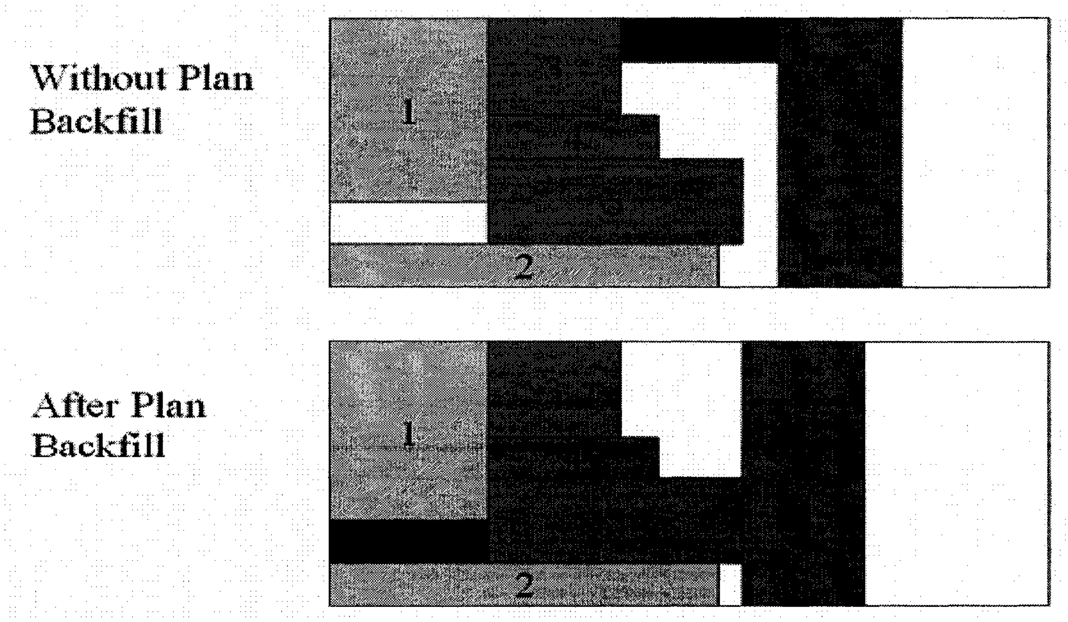


Figure 14 - Plan backfilling example

The backfilled job may alter the created plan which reflects the optimal schedule of jobs. This has been implemented to compensate the incomplete discrepancy tree search which may miss out good schedules. Plan backfilling intentionally deviates from the actual schedule order. An example is depicted in Figure 14 where darkened job is been backfilled into the plan without altering other jobs except the ones scheduled after them. As you can observe, the resulting schedule is more compact. Plan backfilling is also employed during recalculation. In this case, the earlier plan would comprise of pre-empted and plan jobs. Pre-empted jobs are scheduled on the same nodes and new jobs try to occupy excluded nodes for the same reason explained for smart backfilling.

7. Implementation Details

This section would comprise a detailed breakdown on the implementation aspects of the scheduler. Basically, the jobs are classified into three types based on runtime:

- Short – those having runtime less than ten minutes
- Medium – those having runtime greater than short and less than three hours
- Long – those having runtime greater than medium but less than eight hours

Slices are scheduled on the machine by the share control object (part of SCOJO-PECT). Slices are of three different types – short, medium and long. Ideally, jobs of one type are scheduled on identical slices. For instance, short jobs are only scheduled in short slice and so on. The switching overhead between slices is set to sixty seconds.

During optimization, the response times of the virtually scheduled plan jobs were calculated as follows:

For a long job `j`:

```
j.responseTime = actual current time - j.getSubmitTime() + (virtual
current time + j.getRemainingRuntime() - actual current time)/0.73
```

For a medium job `j`:

```
j.responseTime = actual current time - j.getSubmitTime() + (virtual
current time + j.getRemainingRuntime() - actual current time)/0.265
```

This is done to simulate the real time sharing environment. The time shares are absent during optimization. In order to address this issue, the time within the optimization frame is expanded by those factors. Share controller allocates 73% of the machine for long slices and 26.5% are allotted for medium slice. The same factor is used to expand the time.

7.1 Data Structures

Listed below are the data structures used to develop the different modules of optimization. The code was written in Java (SDK 1.5.3) using NetBeans IDE. One external library (colt.jar) comprising the Lublin-Feitelson workload model [17] was used. The heuristic based search tree uses the following data structures:

- Search Tree: The branch-and-bound tree was implemented using the tree data structure using arrays of nodes.
- Priority Queue: Priority Queue for sorting best schedules based on branch-and-bound criteria (total slowdown) is implemented.

The optimizer uses the following data structures:

- Unused nodes information: A vector of unused object stores information about the duration for which the nodes are available. This is implemented as part of the optimizer object and is later used for resolving conflict with the plan.
- Plan object stores the jobs in a vector. Jobs are added into the vector upon virtual scheduling and removed once scheduled. Array Lists are also used.

7.2 Group Creation

The criteria for group formation comprise of minimum group size, minimum machine utilization and work done by the jobs.

- The minimum number of jobs in a group is set to five for long jobs and ten for medium job class. The maximum number of jobs in a long group is set to 30 and in a medium group is set to 20. These values are based on the results provided in the Section 9.1.

- If the current slice is long, then the total job size of the waiting jobs should be greater than twice the machine size for long slice. On the other hand, for a medium slice, the total job size of the waiting jobs should be greater than 1.5 times that of the machine size.
- For a long slice, the ratio of total accumulated work to the machine size should be greater than the product of lambda (constant) and average runtime of the waiting jobs. Medium slice would have a different value for lambda. The average runtime for long jobs is set to 15000 and lambda is set to 1. For medium jobs, the average runtime is set to 800 and lambda is assigned the value 2.

7.3 Search Module

The search module encompasses the implementation of the search algorithm. Notable are the values of the parameters. The permutation generator is allowed to loop for a maximum of 10000 iterations, thus generating the same number of schedules. Assuming an average group size to be 20 in a workload of 10000 jobs and an average number of groups to be 50, approximately 1,000,000 nodes would be generated per group. Nodes are identified with four attributes: job number, count of current branches emanating from the node, maximum branch limit, and pointers to other nodes.

We proceed through the first level of search only when the minimum number of long-wide jobs is greater than 3. The criteria for selecting long-wide jobs differ according to the slice we are in. For a long slice, long-wide jobs are differentiated when the job has a minimum of one-fourth the size of the machine and having a runtime of 15000. Alternatively, for a medium slice, any job having a minimum size of one-tenth the machine size and runtime of 800. The maximum number of long-wide jobs permitted in a group is 6 because we do a complete search for the best combination of jobs. Force-shifting is performed when the remaining runtime of the plan job is at most one and a half times the length of the hole. If shifting has taken place, the jobs scheduled after the shifted job is removed and a new shifted order is generated and later scheduled. The best schedule is determined by the weighted function discussed in Section 6.4.4. The constants were set to the following values: $\alpha = 1.1$, $\beta = 0.95$, $\gamma = 1.1$.

7.4 Branch-and-Bound

While LSF heuristic would determine the schedule ordering, total bounded slowdown of the jobs in the schedule would determine its efficiency. After each job is virtually scheduled in the permuted order, its slowdown is calculated based on estimated response times achieved (explained above). The number of pruned nodes was counted at the end of the iteration. Section 5.1.5 describes how to compute bounded slowdown. The value of BOUND was set to 600 seconds.

7.5 Summary of Values Used

The constants used for my implementation, initialized values of variables are shown in Table 4.

Constants	**Values**
Minimum group size for long slice	5
Minimum group size for medium slice	10
Minimum long-wide jobs in group	3

Maximum long-wide jobs in group	6
Relative response time bound	600
Iterations in search tree	10000
Threshold ratio for force-shifting job	1.50
Avg. response time factor (α)	1.1
Utilization factor (β)	0.95
Makespan factor (γ)	1.1
Avg. runtime for long jobs	15000
Avg. runtime for medium jobs	800
Limit on group size	30
Slices switching overhead	60 sec
Number of kind of job classes	3 classes: short, medium, long
Default machine size	128
Number of jobs in workload	10000
Classification of short jobs	runtime < 10 min
Classification of medium jobs	60 sec $\leq$ runtime < 3 hours
Classification of long jobs	runtime $\geq$ 3 hours
Classification of narrow	size $\leq$ 10% of machine size
Classification of medium size	10% of machine size < size $\leq$ 50% of machine size
Classification of wide size	Size > 50% of machine size
Workload share allocated for long jobs	73%

Workload share allocated for medium jobs	26.5%
Seeds tried for workload	71, 31, 35, 73, 7
Real traces used for experiments	SDSC, LANL

Table 4 - Implementation values

8. Experimental Setup

In this section I would like to elaborate on my input dataset and evaluation plan. The results are obtained after executing the simulator on our Horus cluster, a 16 node 2-way machine with 512MB RAM. The Horus cluster runs Debian Linux with kernel version 2.6.6. The only external library used is the Lublin-Feitelson model [17]. The cluster had JDK version 1.5.0_07 installed.

8.1 Input Workload Specifications

Input data is chosen to be of relatively heavy loaded (less inter-arrival times and good system utilization) as potential for group formation are higher and hence creating higher chances for optimization. The input workloads were generated from Lublin-Feitelson model. All workloads comprise of 10000 jobs. The following four traces (generated by using different seed values) were selected. N_{type} refers to the percentage of jobs that belongs to that type whereas, W_{type} denotes the amount of work associated with the type. Table 5 describes the number of short, medium and long jobs in the workload, the amount of work done by them, average job sizes and average inter-arrival time of jobs. The model has generated workloads for default machine size of 128 processors.

Work Load	% of Jobs			% of Work			Avg. Job Size			Avg. Inter-Arrival Time (sec)
	N_{short}	N_{med}	N_{long}	W_{short}	W_{med}	W_{long}	S_{short}	S_{med}	S_{long}	
Seed 71	64	19	17	0.5	26.5	73	9	17	19	808
Seed 31	63	20	17	0.5	25	74.5	9	16	20	859
Seed 73	64	20	16	0.4	26.5	73.5	9	17	21	797
Seed 35	65	19	16	0.5	27	72.5	9	18	20	800
Seed 7	64	20	16	0.5	26	73.5	8	17	21	839

Table 5 - Characteristics of synthetic workloads

Table 6 describes the features of real workloads under consideration. Jobs having runtime greater than 8 hours and occupying at most 10% of the machine were considered long-narrow jobs. The inter-arrival times for real traces were much higher than those used from the model.

Work Load	% of Jobs			% of Work			Avg. Job Size			Ratio of long-narrow to long jobs
	N_{short}	N_{med}	N_{long}	W_{short}	W_{med}	W_{long}	S_{short}	S_{med}	S_{long}	
SDSC-BLUE	73.7	17.7	8.6	1	15	84	34	114	112	0.79
LANL	61.5	34	4.5	2	41	57	74	92	244	0.52

Table 6 - Characteristics of real workload traces

The workload distribution is quite different and it is clear that long narrow jobs decrease the benefits from groups as they cannot be packed well to deliver good utilization. SDSC-BLUE trace runs on 1152 processors while LANL runs on 1024 processors. All the above traces also had an equally good utilization (> 75%). We neglect those jobs which have negative runtimes or sizes as they represent either jobs that were terminated abnormally or dummy jobs inserted by system administrators. Observe that percentage of long narrow jobs in traces is 3-4 times higher than workloads generated by the model. This would mean that 70-75% of the long and medium groups are formed are narrow jobs. This would have implications as described later for worst cases in Section 9.4.

8.2 Evaluation Plan

In order to prove the efficiency of optimization, the following tests were used:

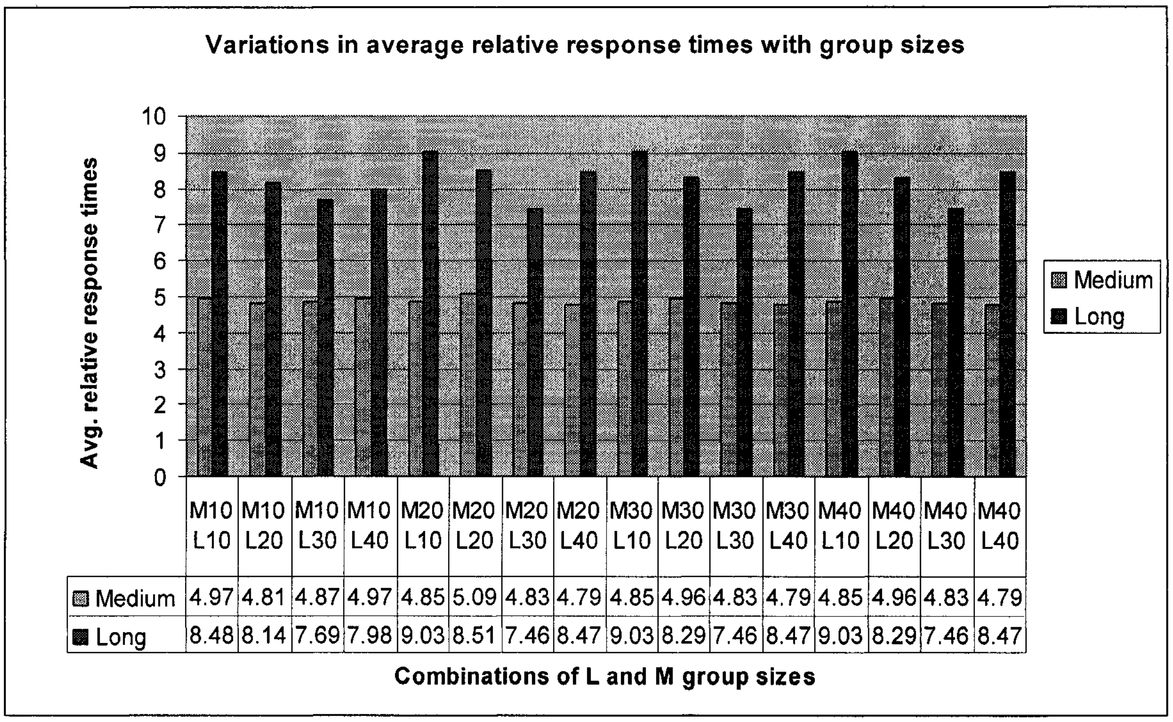
1. Firstly, we determine the optimal group sizes for long and medium jobs that gives the best results for average relative response times.

2. As the number of groups formed would reflect the level of optimization, a table of groups of job types created would be included for each workload.
3. For each workload, comparison curves for average relative response times and average response times, average wait times and machine utilization with and without optimization for conservative SCOJO-PECT would be plotted.
4. The effect of dynamic backfilling upon submission on the wait times of jobs would be analyzed and interpreted.
5. The efficiency of pruning achieved through branch-and-bound for the discrepancy search tree would to be ascertained by keeping track of the percentage of explored nodes.
6. The gain achieved from the second level of hierarchical search can be plotted against the actual optimized results to understand the contribution of the second level search and branch-and-bound.
7. Smart-backfilling is part of SCOJO-PECT. Its influence on the optimization would be studied.
8. Finally to conclude, we have shown the results of our optimization relative to the bottom line approach.

9. Experimental Results

9.1 Best group sizes

The chart below indicates the variations in average relative response times for different group sizes for medium and long jobs.



	M10 L10	M10 L20	M10 L30	M10 L40	M20 L10	M20 L20	M20 L30	M20 L40	M30 L10	M30 L20	M30 L30	M30 L40	M40 L10	M40 L20	M40 L30	M40 L40
▨ Medium	4.97	4.81	4.87	4.97	4.85	5.09	4.83	4.79	4.85	4.96	4.83	4.79	4.85	4.96	4.83	4.79
■ Long	8.48	8.14	7.69	7.98	9.03	8.51	7.46	8.47	9.03	8.29	7.46	8.47	9.03	8.29	7.46	8.47

Figure 15 - Relationship of avg. relative response times and group sizes

The chart in Figure 15 shows the group sizes of 10, 20, 30 and 40 jobs. M10 L20 would mean a group of 10 jobs for medium and 20 for long jobs. As seen from Figure 15, the group size giving the best results was found to be L30 M20. This group size has been used for all the following tests.

9.2 Groups formed

The number of groups formed is critical to measure the effectiveness of optimization. More importantly, the percentage of jobs being optimized and the groups formed per job class has to be tabulated. The groups formed for different workloads are displayed in Table 7.

WORKLOAD	#M GROUPS	#L GROUPS	# GROUPS	GROUPING%
SEED 71	24	31	55	17%
SEED 35	23	38	61	19%
SEED 31	20	39	59	18%
SEED 73	30	29	59	18%
SEED 7	37	32	69	23%
SEED 13	29	27	56	16.5%
SDSC	9	0	9	3%
LANL	50	11	61	19%

Table 7 - Number of groups formed per workload

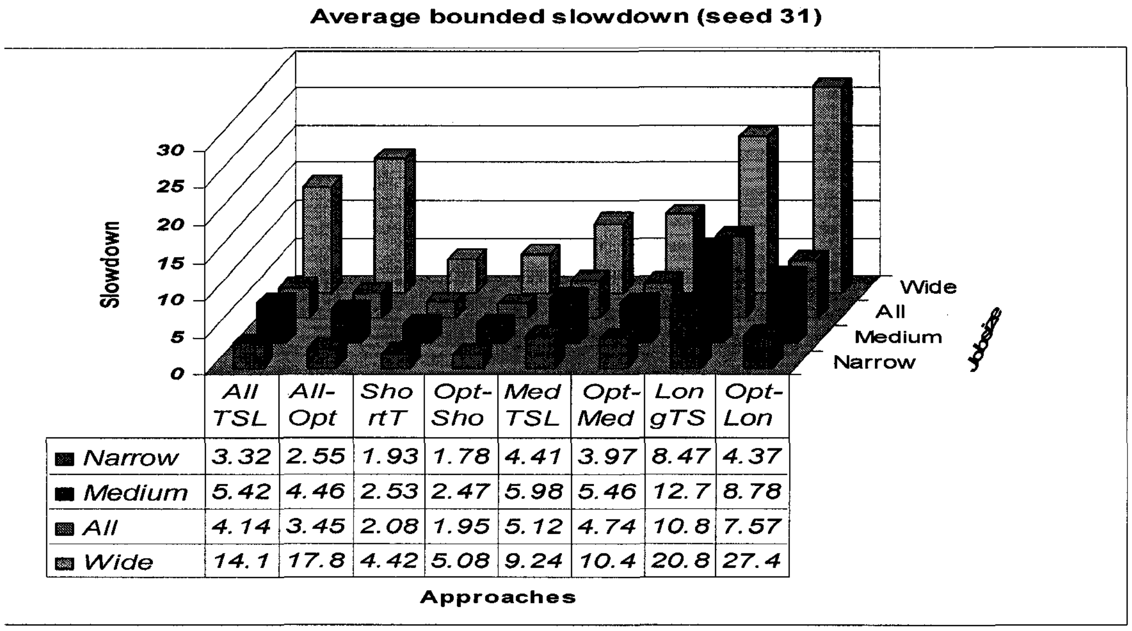
Notice that SDSC trace restricts groups to be formed due to its workload distribution which are also reflected in the number of days of workload creation.

9.3 Efficiency of optimization

In this section, we will compare the average slowdown, average response times, average wait times, utilization and gains from branch-and-bound and bi-level search. We shall discuss the results and implications of the synthetic workloads first followed by the real workloads.

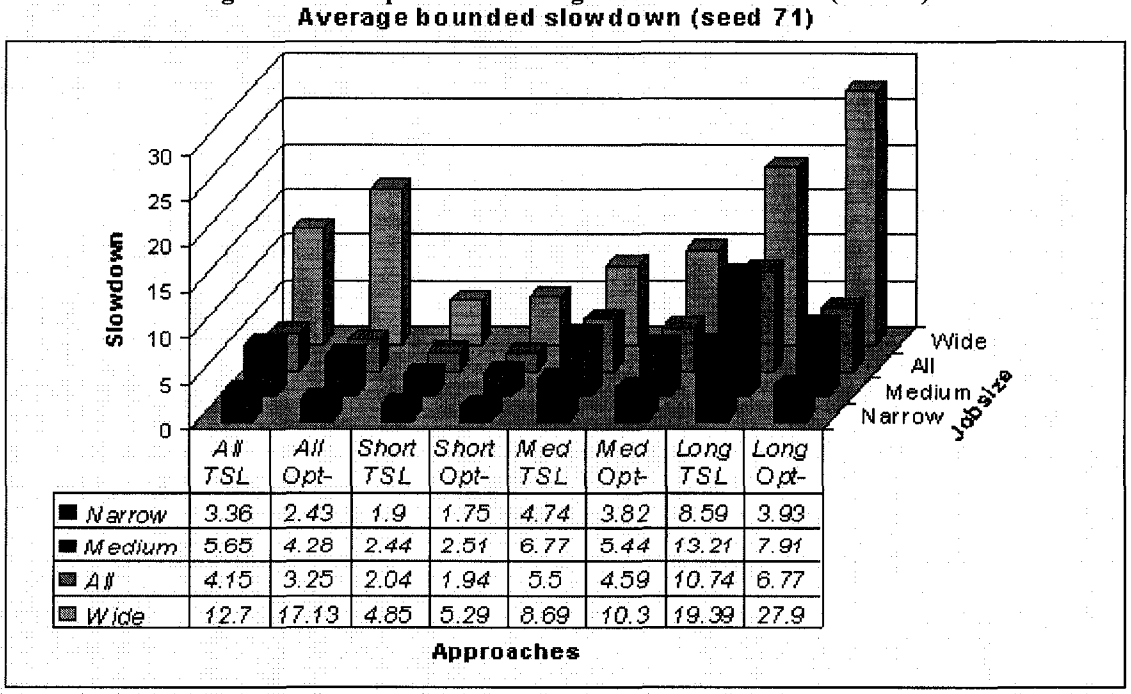
9.3.1 Average Bounded Slowdown

Bounded slowdowns charts are displayed in the figures below. Clearly, the optimized version of the SCOJO-PECT scheduler has brought about an average of 9% (for medium) and 15% (for long) improvements in average bounded slowdown. Notable are the improvements achieved using seed 31 (Figure 16) for individual classes are 49% for long narrow and medium and 26% for long job classes.



	All TSL	All-Opt	ShortT	Opt-Sho	Med TSL	Opt-Med	LongTS	Opt-Lon
■ Narrow	3.32	2.55	1.93	1.78	4.41	3.97	8.47	4.37
■ Medium	5.42	4.46	2.53	2.47	5.98	5.46	12.7	8.78
▨ All	4.14	3.45	2.08	1.95	5.12	4.74	10.8	7.57
▨ Wide	14.1	17.8	4.42	5.08	9.24	10.4	20.8	27.4

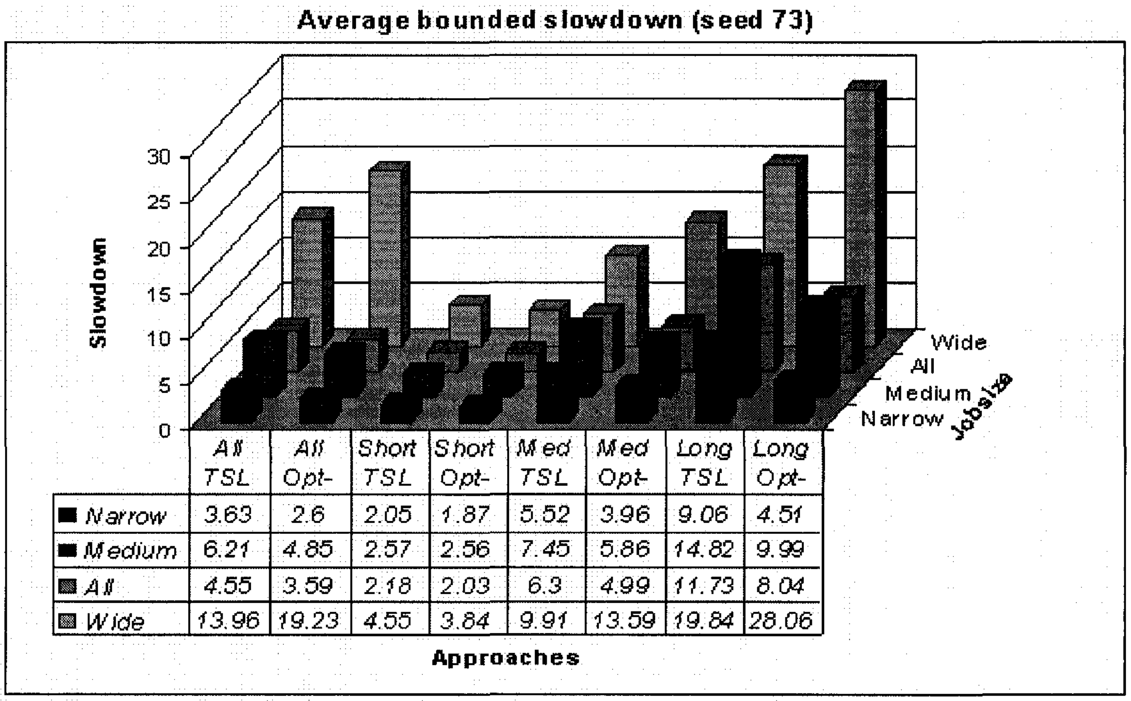
Figure 16 - Comparison of average bounded slowdown (seed 31)



	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	3.36	2.43	1.9	1.75	4.74	3.82	8.59	3.93
■ Medium	5.65	4.28	2.44	2.51	6.77	5.44	13.21	7.91
▨ All	4.15	3.25	2.04	1.94	5.5	4.59	10.74	6.77
▨ Wide	12.7	17.13	4.85	5.29	8.69	10.3	19.39	27.9

Figure 17 - Comparison of average bounded slowdown (seed 71)

In Figure 17, SCOJO-PECT had been outperformed by a maximum of 10% for short, 21% for medium and 55% for long job classes. Though the algorithm on the whole gives a fair advantage over SCOJO-PECT, it fails to optimize the wide jobs. In general, wide jobs cannot be packed well into groups. Another reason for this effect pertaining to our approach would be that optimizing once per group would disregard the updated partial runtimes of the preempted jobs. As jobs continue to be scheduled in the SCOJO-PECT fashion upon preemption, wide jobs are badly affected. This effect is also visible in all the subsequent graphics below.

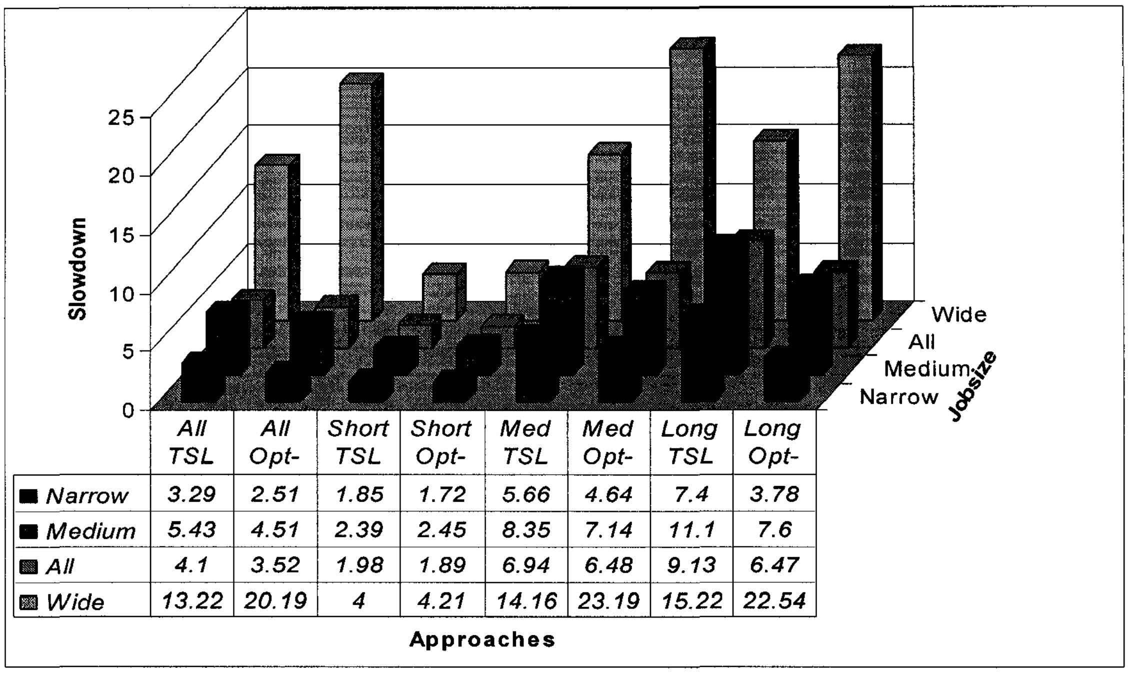


	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	3.63	2.6	2.05	1.87	5.52	3.96	9.06	4.51
■ Medium	6.21	4.85	2.57	2.56	7.45	5.86	14.82	9.99
▨ All	4.55	3.59	2.18	2.03	6.3	4.99	11.73	8.04
▨ Wide	13.96	19.23	4.55	3.84	9.91	13.59	19.84	28.06

Figure 18 - Comparison of average bounded slowdown (seed 73)

For seed 73 (Figure 18), long jobs display a maximum improvement of 40% while the medium jobs account for only 22% improvement.

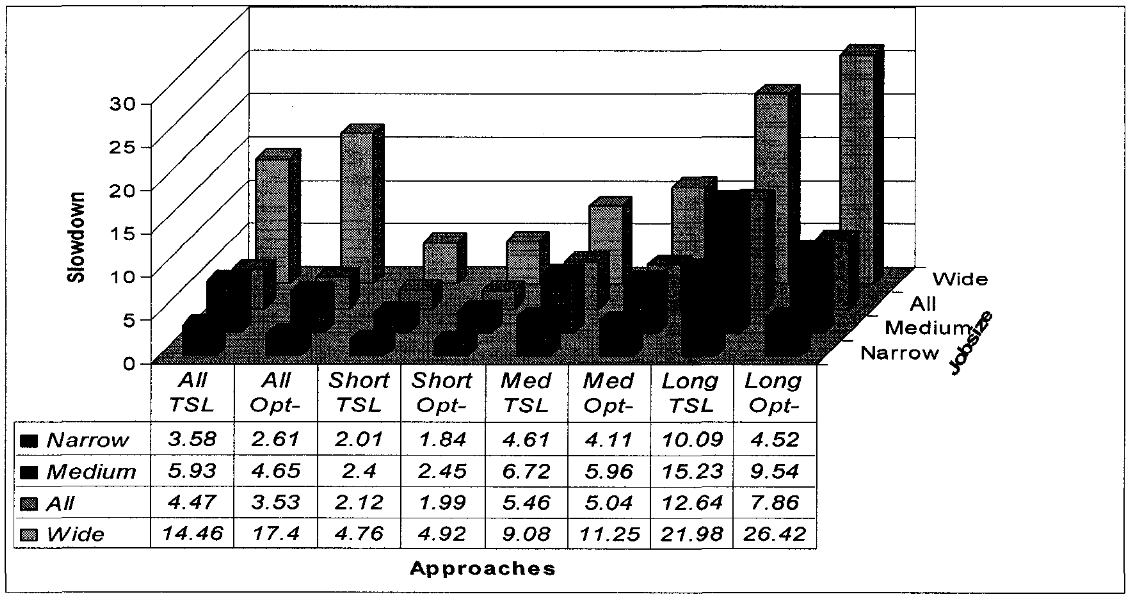
Average bounded slowdown (seed 35)



	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	3.29	2.51	1.85	1.72	5.66	4.64	7.4	3.78
■ Medium	5.43	4.51	2.39	2.45	8.35	7.14	11.1	7.6
▨ All	4.1	3.52	1.98	1.89	6.94	6.48	9.13	6.47
▨ Wide	13.22	20.19	4	4.21	14.16	23.19	15.22	22.54

Figure 19 - Comparison of average bounded slowdown (seed 35)

Average bounded slowdown (seed 7)

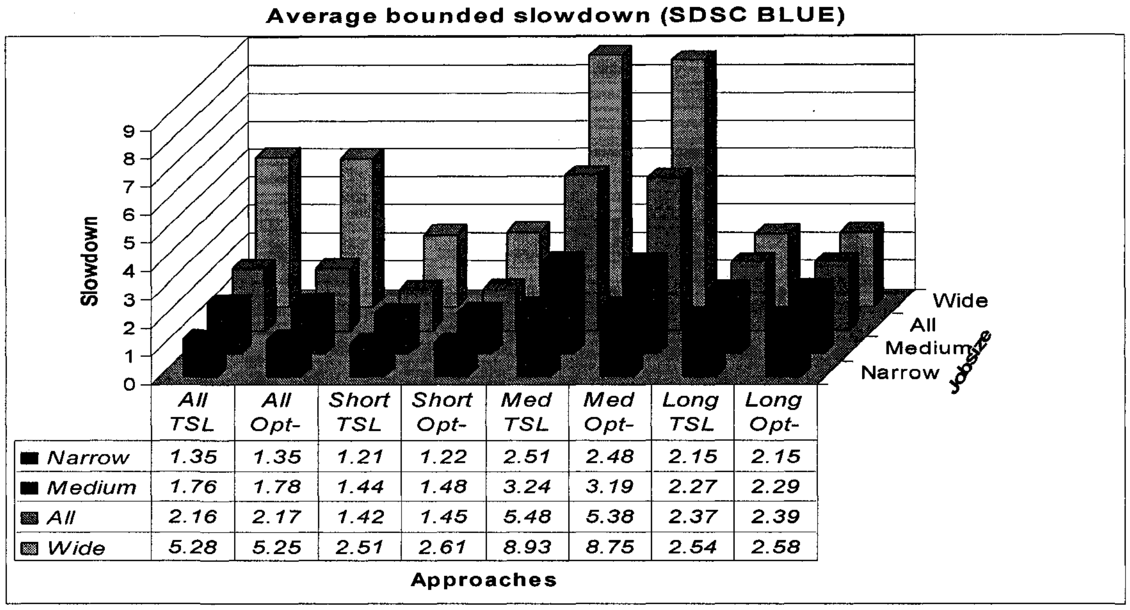


	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	3.58	2.61	2.01	1.84	4.61	4.11	10.09	4.52
■ Medium	5.93	4.65	2.4	2.45	6.72	5.96	15.23	9.54
▨ All	4.47	3.53	2.12	1.99	5.46	5.04	12.64	7.86
▨ Wide	14.46	17.4	4.76	4.92	9.08	11.25	21.98	26.42

Figure 20 - Comparison of Average bounded slowdown (seed 7)

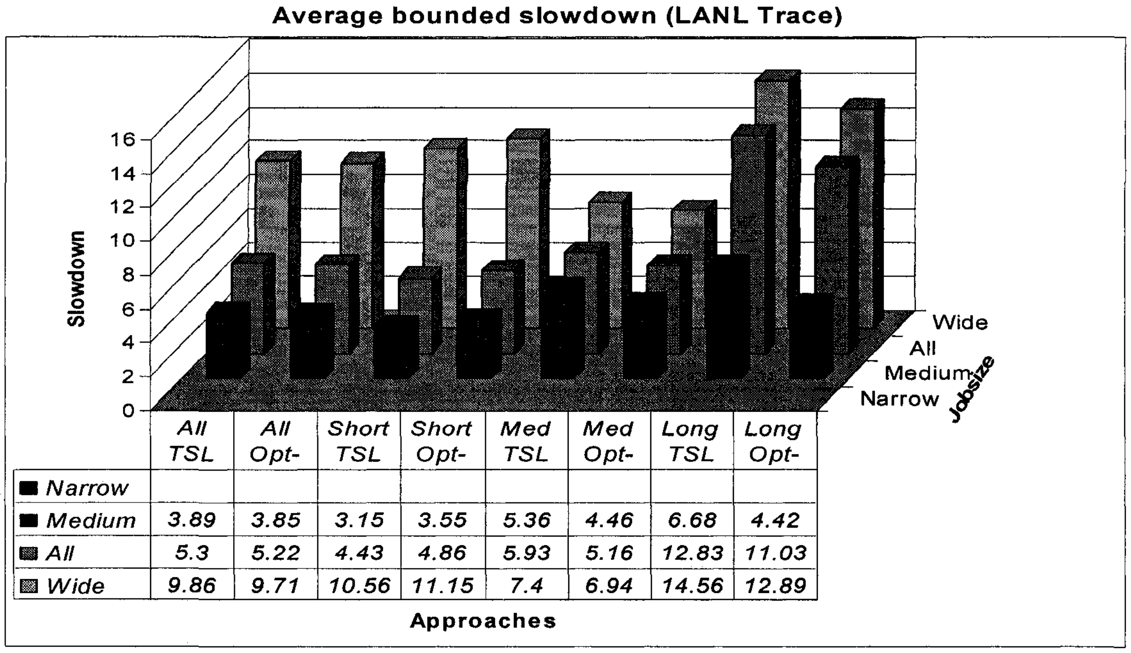
Figure 19 (using seed 35) gives 30% optimized results for relative response times in the case of long jobs. On the other hand, short and medium jobs are addressed less than 10%.

Figure 20 illustrates a similar comparison chart for seed 7. If wide jobs remain scattered over the trace, optimization would not provide much benefit as seen for long and medium jobs.



	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	1.35	1.35	1.21	1.22	2.51	2.48	2.15	2.15
■ Medium	1.76	1.78	1.44	1.48	3.24	3.19	2.27	2.29
▨ All	2.16	2.17	1.42	1.45	5.48	5.38	2.37	2.39
▨ Wide	5.28	5.25	2.51	2.61	8.93	8.75	2.54	2.58

Figure 21 - Comparison of average bounded slowdown for SDSC-BLUE trace



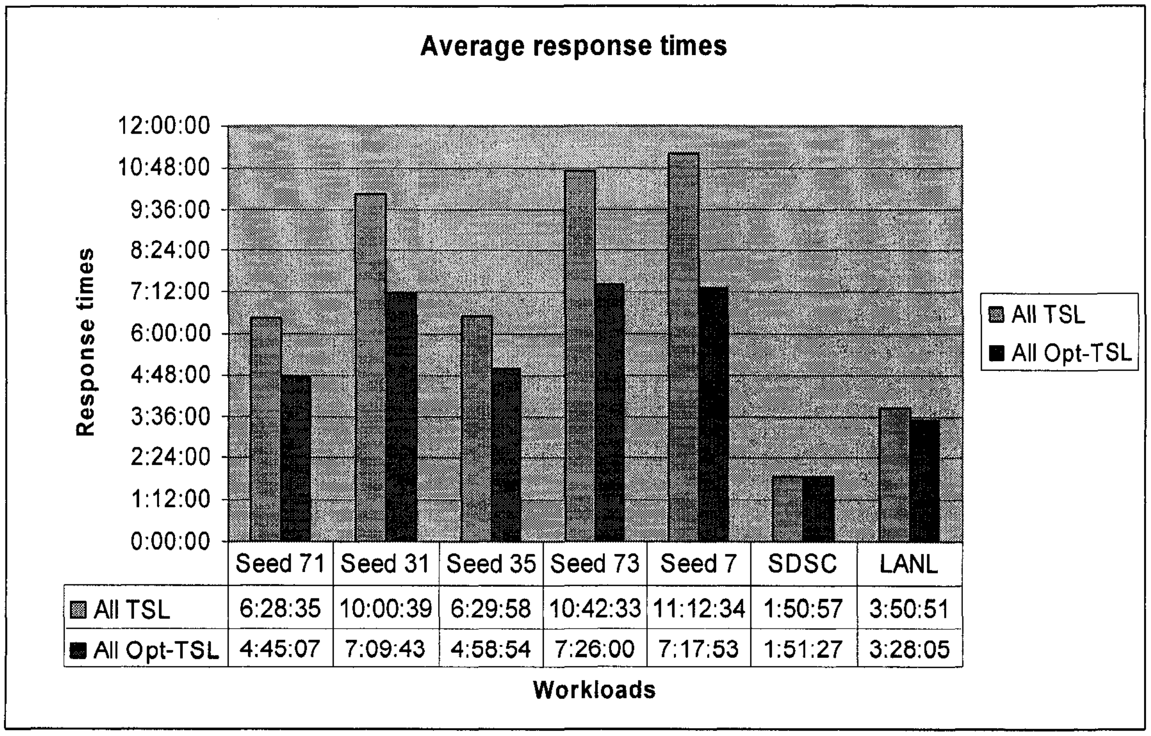
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow								
■ Medium	3.89	3.85	3.15	3.55	5.36	4.46	6.68	4.42
▨ All	5.3	5.22	4.43	4.86	5.93	5.16	12.83	11.03
▨ Wide	9.86	9.71	10.56	11.15	7.4	6.94	14.56	12.89

Figure 22 - Comparison of average bounded slowdown for LANL trace

The graphic shown in Figure 21 and Figure 22 illustrates the comparison of SCOJO-PECT and optimized SCOJO-PECT for real workload traces. The traces have a different workload distribution as shown in Table 6. The results got from traces are less promising due to higher inter-arrival times of jobs leading to a poor utilization.

9.3.2 Average Response Times

The response times reflect the amount of time spent for a particular job in the system. Response times are affected by the percentage of long and wide jobs as they consume a large portion of the system leading to greater wait times for shorter jobs. However, our optimal scheduling scheme in the average has shown to provide improvements of over 12% in average response times.



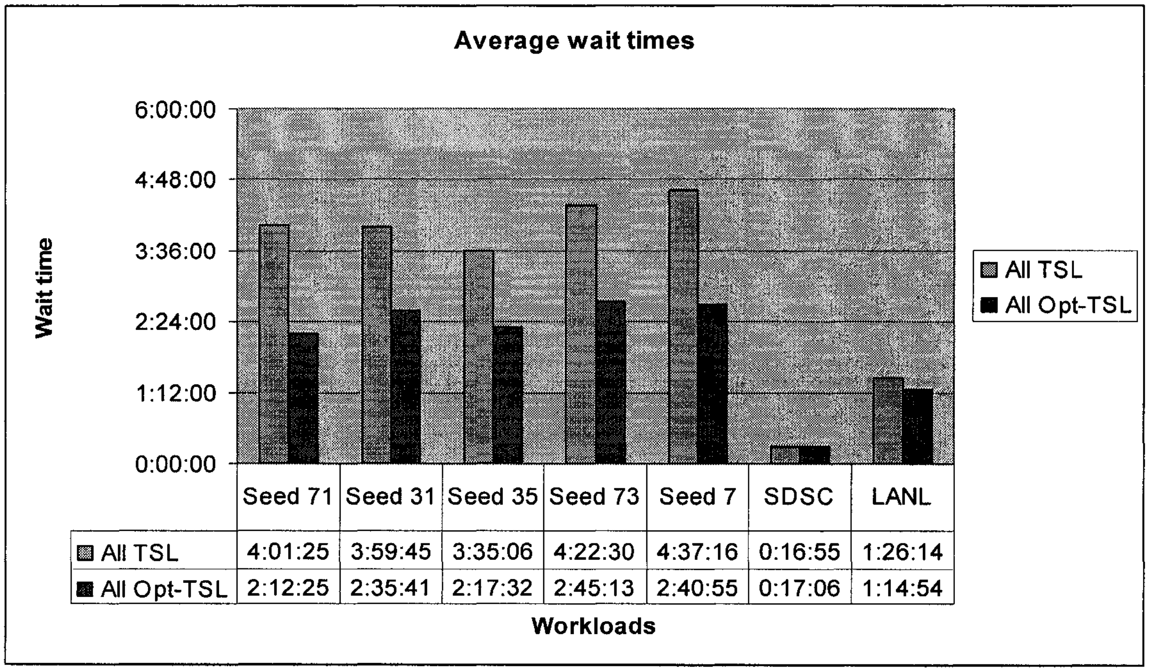
	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7	SDSC	LANL
▨ All TSL	6:28:35	10:00:39	6:29:58	10:42:33	11:12:34	1:50:57	3:50:51
■ All Opt-TSL	4:45:07	7:09:43	4:58:54	7:26:00	7:17:53	1:51:27	3:28:05

Figure 23 - Comparison of average response times

It is observed that traces experience larger number of narrow jobs. Consequently, these jobs are pushed behind by the heuristic resulting in worsening the response times of these jobs as visible from Figure 23.

9.3.3 Average Wait Times

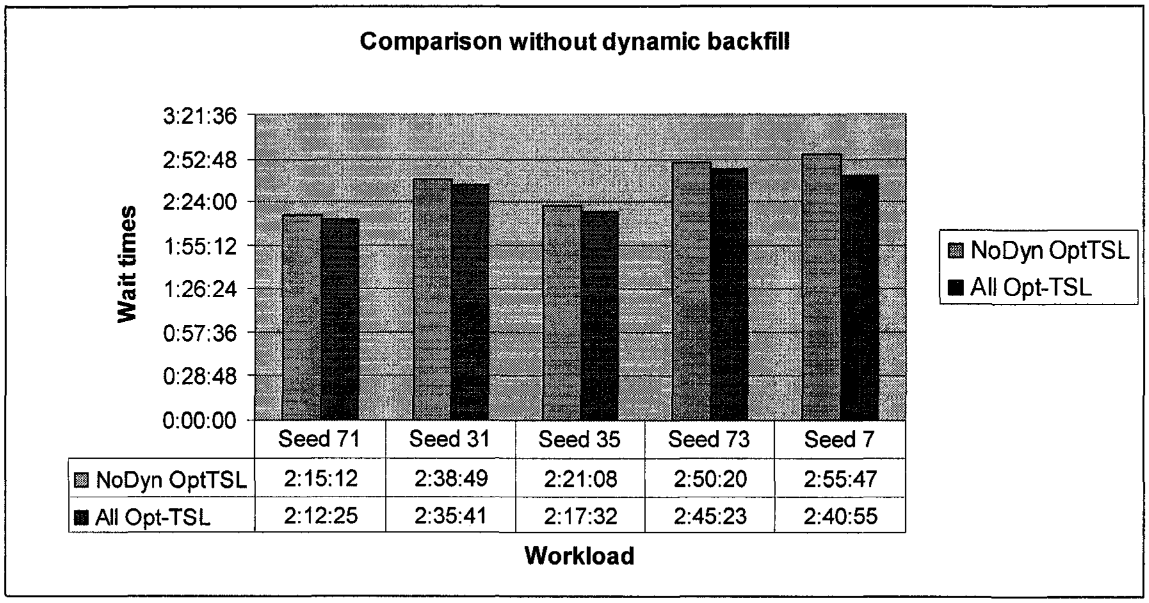
Wait times are straight forward to understand, and logically wide jobs incur large wait times especially during heavy workload times when other jobs occupy the resources most of the time. The wait times have got better by an average of 10%. The comparison of the average wait times with and without optimization is shown in Figure 24.



	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7	SDSC	LANL
▨ All TSL	4:01:25	3:59:45	3:35:06	4:22:30	4:37:16	0:16:55	1:26:14
■ All Opt-TSL	2:12:25	2:35:41	2:17:32	2:45:13	2:40:55	0:17:06	1:14:54

Figure 24 - Comparison of average wait times

The influence of dynamic backfilling upon submission on wait times is illustrated in Figure 25. Evidently, dynamic backfill does account for an average of 5% improvements in average wait times. Medium and short jobs are best served using dynamic backfilling as they can be readily started within a plan until the end of slice if sufficient nodes are free. This prevents such jobs from waiting even though the jobs are of a different slice type.

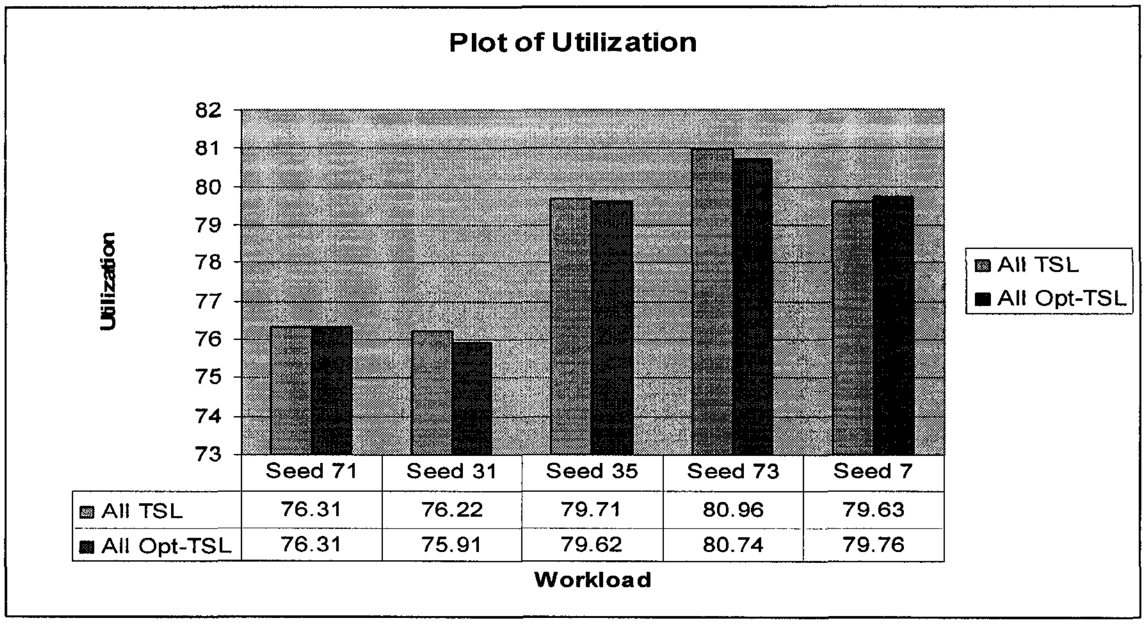


	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7
▨ NoDyn OptTSL	2:15:12	2:38:49	2:21:08	2:50:20	2:55:47
■ All Opt-TSL	2:12:25	2:35:41	2:17:32	2:45:23	2:40:55

Figure 25 - Effect of dynamic backfilling upon submission on average wait times

9.3.4 Utilization

In our approach we have tried to contain the utilization by restricting the utilization within a group to fall below a minimum bound. As discussed in 6.3.3, group finish time calculation permits minor variations in utilization to occur. The comparison of the overall machine utilization achieved is shown in Figure 26. On the whole, the optimization has produced a utilization drops by 1-2% which is acceptable.



	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7
▨ All TSL	76.31	76.22	79.71	80.96	79.63
■ All Opt-TSL	76.31	75.91	79.62	80.74	79.76

Figure 26- Utilization comparison (SCOJO-PECT against Optimized SCOJO-PECT)

9.3.5 Gain from Branch-and-Bound pruning

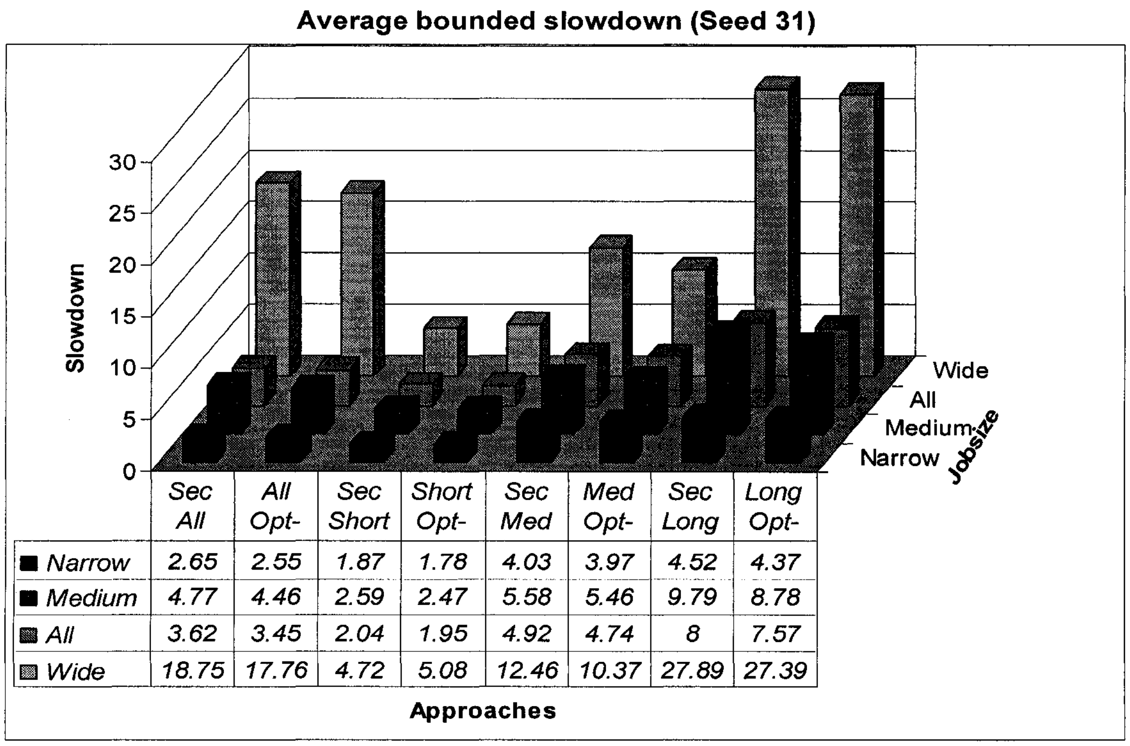
Though pruning occurs in two distinct levels of the search tree, we are interested in the second level search where more permutations are tried. In summary, it was observed that pruning is not very beneficial (see Table 8). This may be due to the fact that largest slowdown heuristic based discrepancy threshold discards majority of the search tree thereby only allowing branch-and-bound to only focus on a smaller search space. Branch and bound is expected to have done better in the absence of our hierarchical search.

Workload	#Total nodes	#Nodes pruned	Pruning %
Seed 71	1709500	69916	4%
Seed 31	2308200	92806	4%
Seed 35	2530702	211552	9%
Seed 73	2115736	113981	5%
Seed 7	2340981	112341	5%

Table 8 - Contribution of branch-and-bound in pruning

9.3.6 Gain from second level search

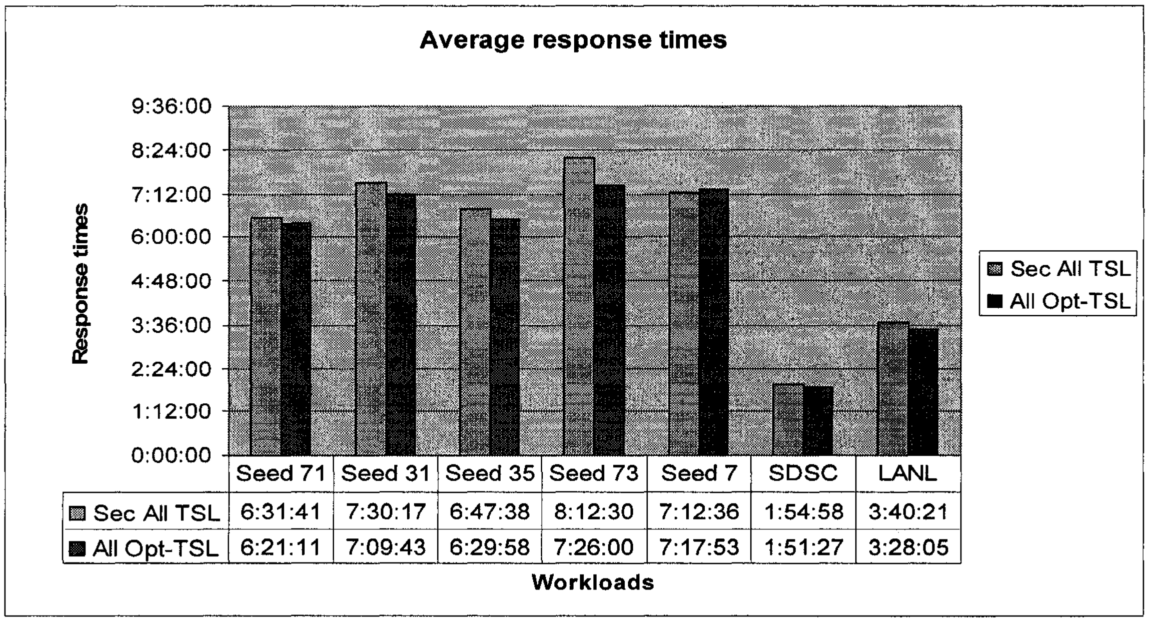
The following graphs would determine the degree of optimization contributed by the second level of our hierarchical search.



	Sec All	All Opt-	Sec Short	Short Opt-	Sec Med	Med Opt-	Sec Long	Long Opt-
■ Narrow	2.65	2.55	1.87	1.78	4.03	3.97	4.52	4.37
■ Medium	4.77	4.46	2.59	2.47	5.58	5.46	9.79	8.78
▨ All	3.62	3.45	2.04	1.95	4.92	4.74	8	7.57
▨ Wide	18.75	17.76	4.72	5.08	12.46	10.37	27.89	27.39

Figure 27 - Influence of second level search on average slowdown (seed 31)

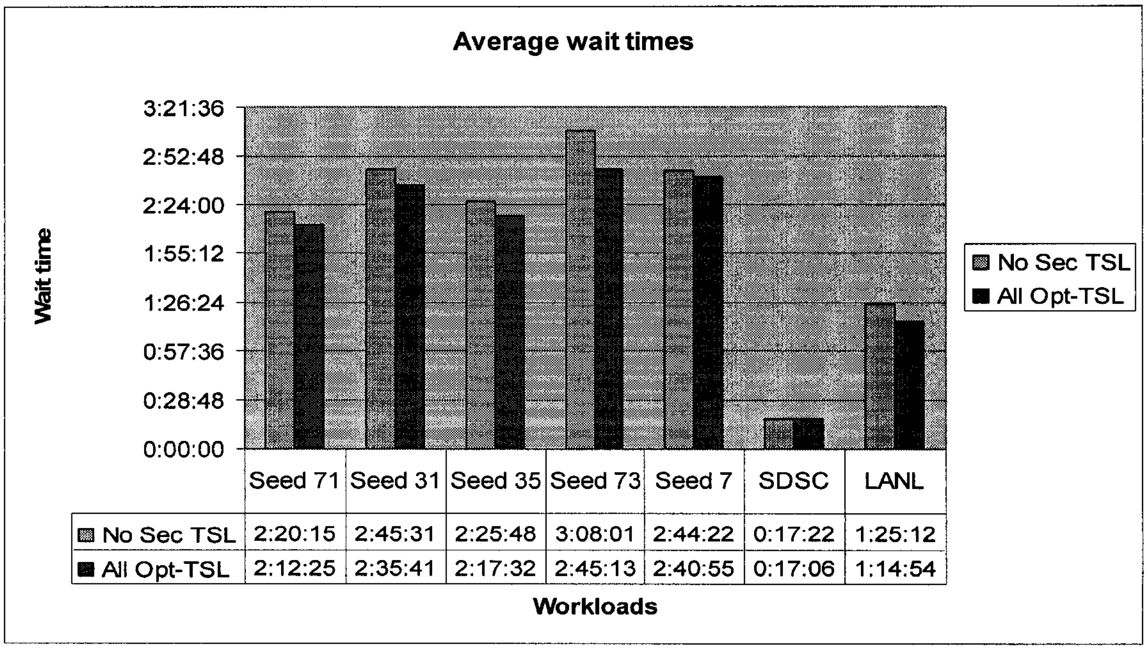
This search is performed after merging the fully optimized long-wide schedule to the remaining non-optimized one. Largest slowdown first is used as the heuristic in this level. The number of groups formed with and without second level search are the same. Figure 27 (using seed 31) clearly projects that second level search contributes little to the optimization framework. To be stressed medium and long jobs are optimized fairly in the range of 2-6%. The wide jobs without second level are getting equally good improvements as these jobs are optimized thoroughly. However, in summary, the influence of the second level search is lower than expected. The graphs pertaining to the comparisons of average response times and average wait times would be able to give a more conclusive picture.



	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7	SDSC	LANL
▨ Sec All TSL	6:31:41	7:30:17	6:47:38	8:12:30	7:12:36	1:54:58	3:40:21
■ All Opt-TSL	6:21:11	7:09:43	6:29:58	7:26:00	7:17:53	1:51:27	3:28:05

Figure 28 - Comparison of average response times (with and without second level search)

From the comparison above (Figure 28), the search was able to deliver only a maximum of 10% improvement in average response times. The average wait time comparison (Figure 29) also supports the graph of average response times.



	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7	SDSC	LANL
▨ No Sec TSL	2:20:15	2:45:31	2:25:48	3:08:01	2:44:22	0:17:22	1:25:12
■ All Opt-TSL	2:12:25	2:35:41	2:17:32	2:45:13	2:40:55	0:17:06	1:14:54

Figure 29 - Comparison of average wait times (with and without second level search)

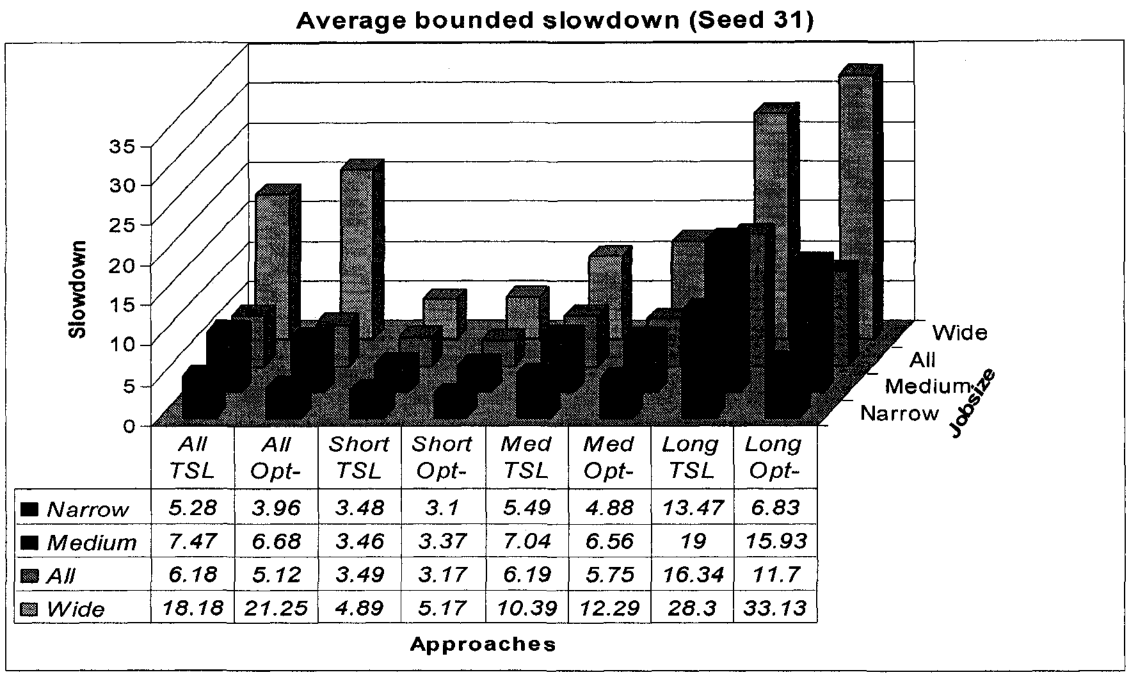
9.4 Effect of Smart-Backfilling

Smart-backfilling is an important strategy that is dynamically invoked to schedule jobs from other slices into the currently running slice. This assumes a conservative approach when backfilling jobs within a group. In this section, we compare SCOJO-PECT and optimized SCOJO-PECT; both devoid of smart backfilling. Naturally, switching off smart backfilling has been able to increase to increase the number of groups formed due to more queue-up of jobs. This has been shown in Table 9.

Workload	#Medium Groups	#Long Groups	#Total Groups	% of jobs grouped
Seed 31	35	46	81	30%
Seed 35	34	39	73	26%
Seed 71	39	33	72	25.5%
Seed 73	40	32	72	25.5%
Seed 7	41	32	73	26%
SDSC	18	9	27	8%
LANL	81	14	95	36.5%

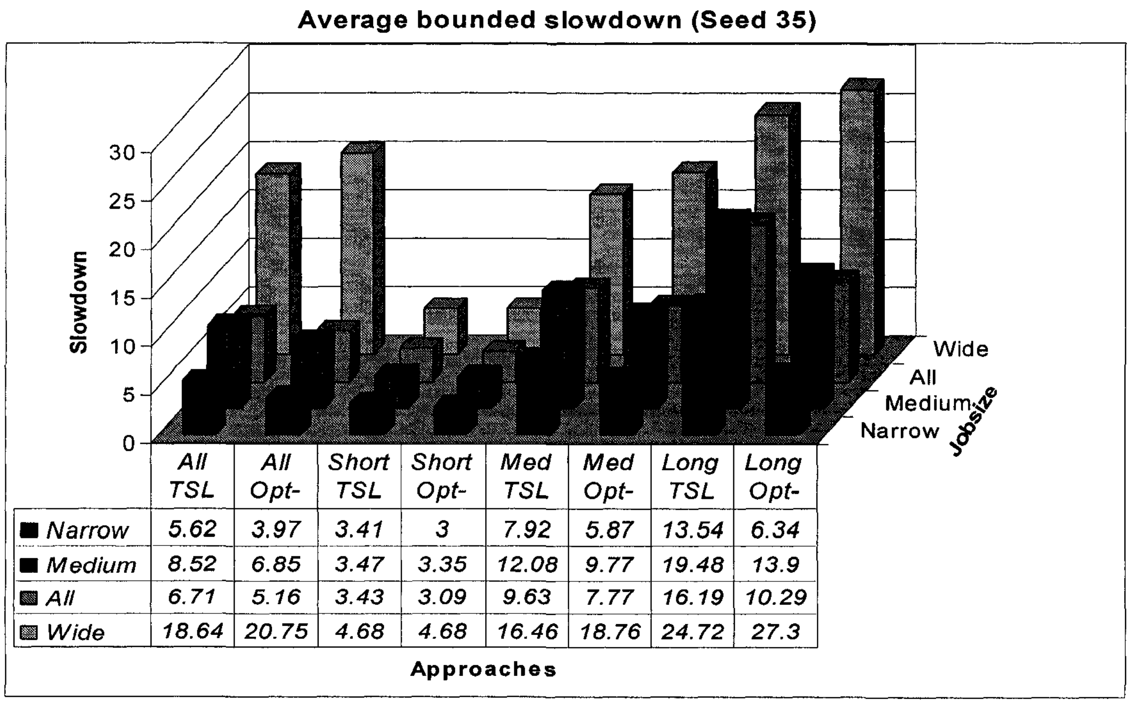
Table 9 - Groups formed without non-type backfilling

The results for average bounded slowdown values without non-type backfilling is illustrated in the following figures. The results emphasize the effectiveness of our optimization approach. Firstly, we have plotted the results running the synthetic workloads (Figure 30, Figure 31, Figure 32, Figure 33, and Figure 34), followed by the real traces (Figure 35 and Figure 36).



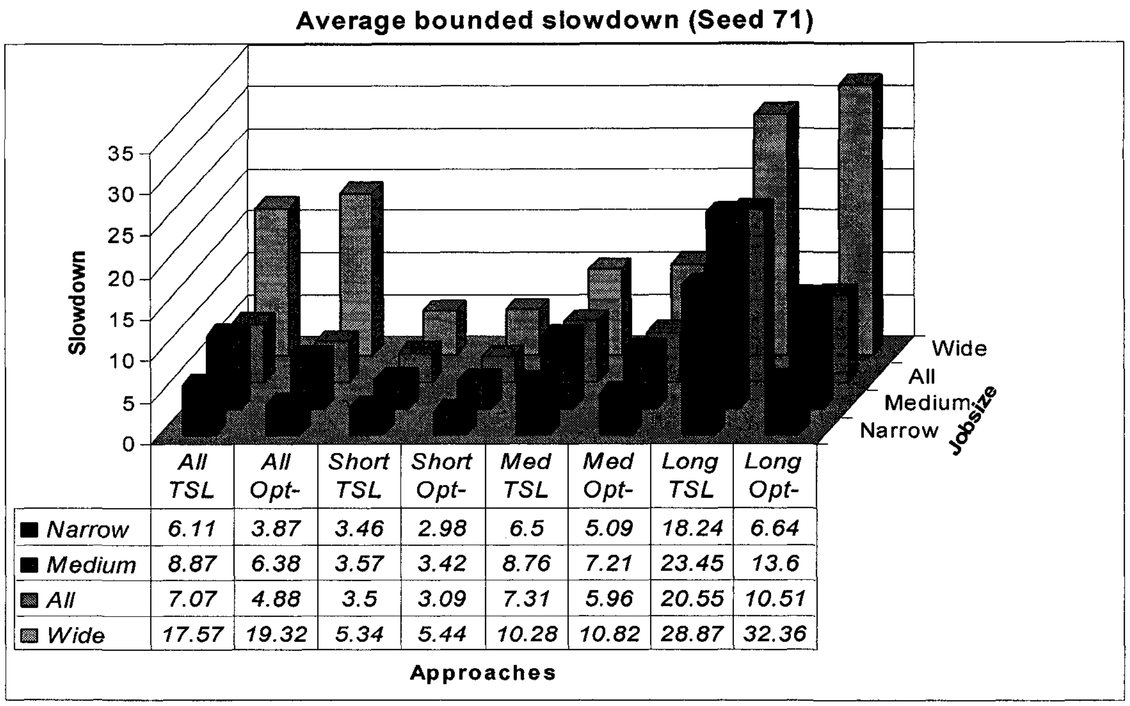
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ *Narrow*	*5.28*	*3.96*	*3.48*	*3.1*	*5.49*	*4.88*	*13.47*	*6.83*
■ *Medium*	*7.47*	*6.68*	*3.46*	*3.37*	*7.04*	*6.56*	*19*	*15.93*
▨ *All*	*6.18*	*5.12*	*3.49*	*3.17*	*6.19*	*5.75*	*16.34*	*11.7*
▨ *Wide*	*18.18*	*21.25*	*4.89*	*5.17*	*10.39*	*12.29*	*28.3*	*33.13*

Figure 30 – Comparison of average bounded slowdown without smart backfilling (Seed 31)



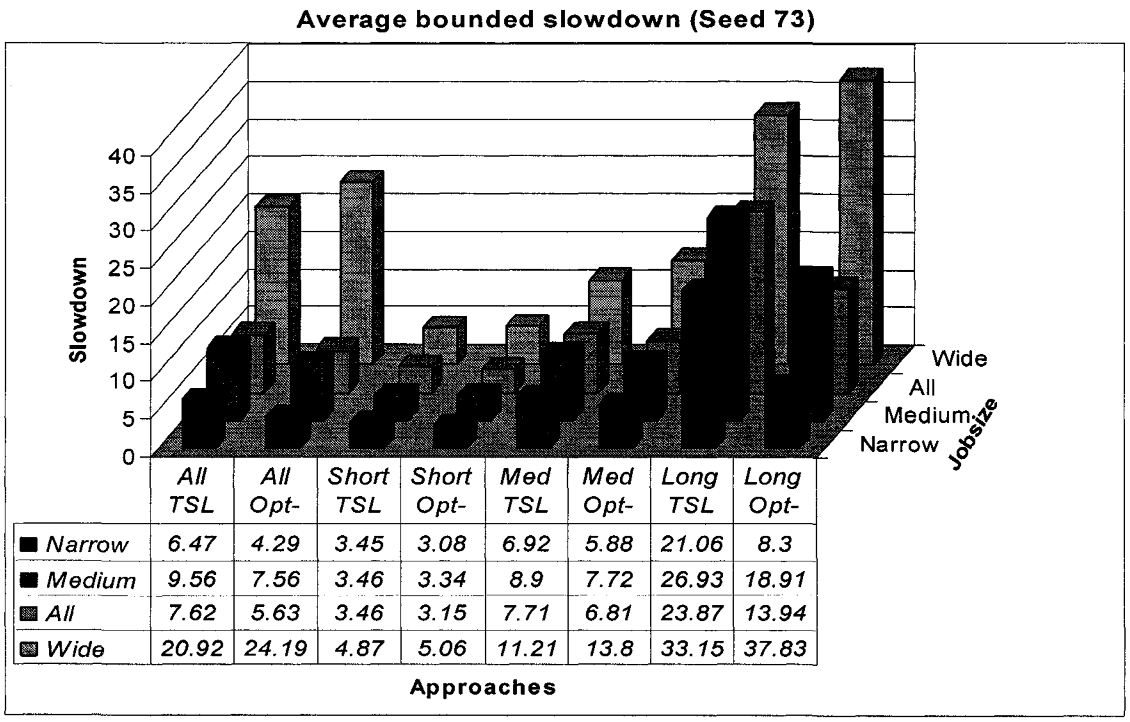
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ *Narrow*	*5.62*	*3.97*	*3.41*	*3*	*7.92*	*5.87*	*13.54*	*6.34*
■ *Medium*	*8.52*	*6.85*	*3.47*	*3.35*	*12.08*	*9.77*	*19.48*	*13.9*
▨ *All*	*6.71*	*5.16*	*3.43*	*3.09*	*9.63*	*7.77*	*16.19*	*10.29*
▨ *Wide*	*18.64*	*20.75*	*4.68*	*4.68*	*16.46*	*18.76*	*24.72*	*27.3*

Figure 31 - Comparison of average slowdown without smart backfilling (Seed 35)



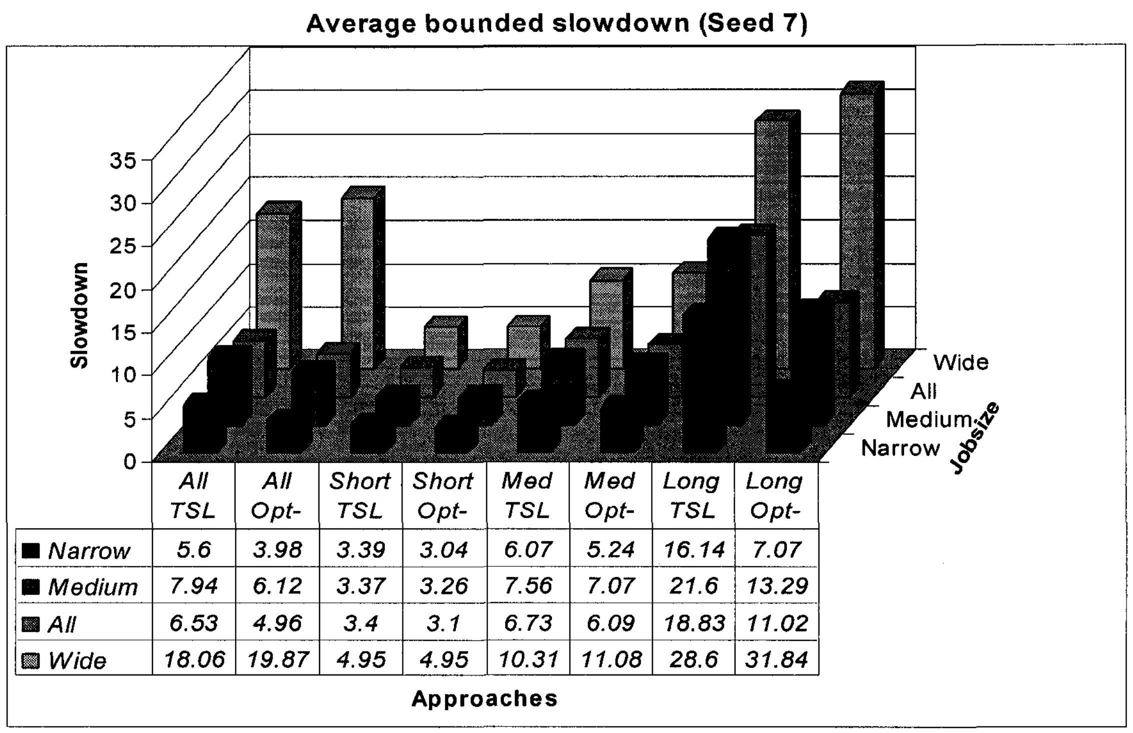
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	6.11	3.87	3.46	2.98	6.5	5.09	18.24	6.64
■ Medium	8.87	6.38	3.57	3.42	8.76	7.21	23.45	13.6
▨ All	7.07	4.88	3.5	3.09	7.31	5.96	20.55	10.51
▨ Wide	17.57	19.32	5.34	5.44	10.28	10.82	28.87	32.36

Figure 32 - Comparison of average slowdown without smart backfilling (Seed 71)



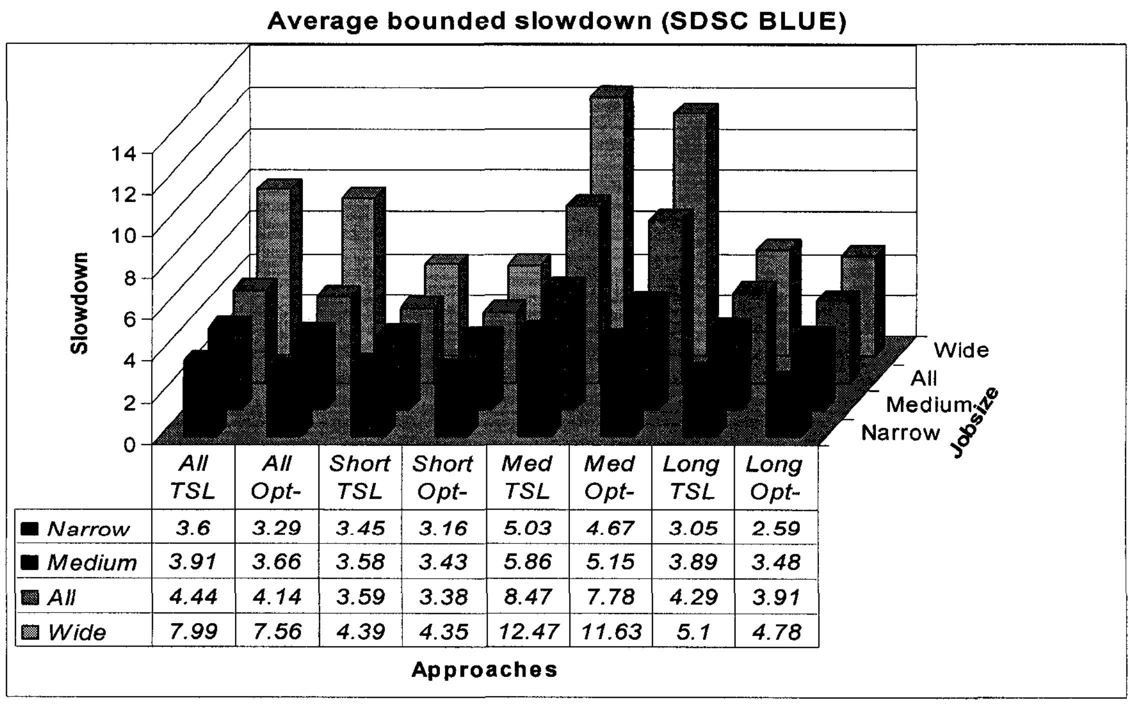
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	6.47	4.29	3.45	3.08	6.92	5.88	21.06	8.3
■ Medium	9.56	7.56	3.46	3.34	8.9	7.72	26.93	18.91
▨ All	7.62	5.63	3.46	3.15	7.71	6.81	23.87	13.94
▨ Wide	20.92	24.19	4.87	5.06	11.21	13.8	33.15	37.83

Figure 33 - Comparison of average slowdown without smart backfilling (Seed 73)



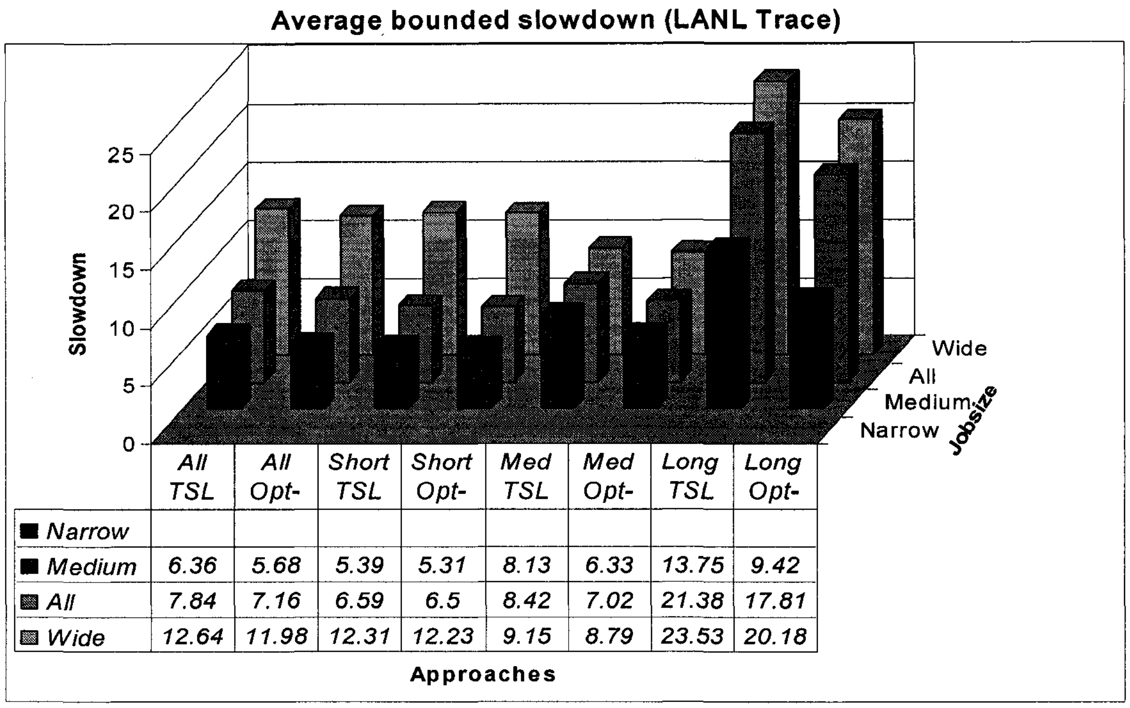
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ *Narrow*	*5.6*	*3.98*	*3.39*	*3.04*	*6.07*	*5.24*	*16.14*	*7.07*
■ *Medium*	*7.94*	*6.12*	*3.37*	*3.26*	*7.56*	*7.07*	*21.6*	*13.29*
▨ *All*	*6.53*	*4.96*	*3.4*	*3.1*	*6.73*	*6.09*	*18.83*	*11.02*
▨ *Wide*	*18.06*	*19.87*	*4.95*	*4.95*	*10.31*	*11.08*	*28.6*	*31.84*

Figure 34 - Comparison of average slowdown without smart backfilling (Seed 7)



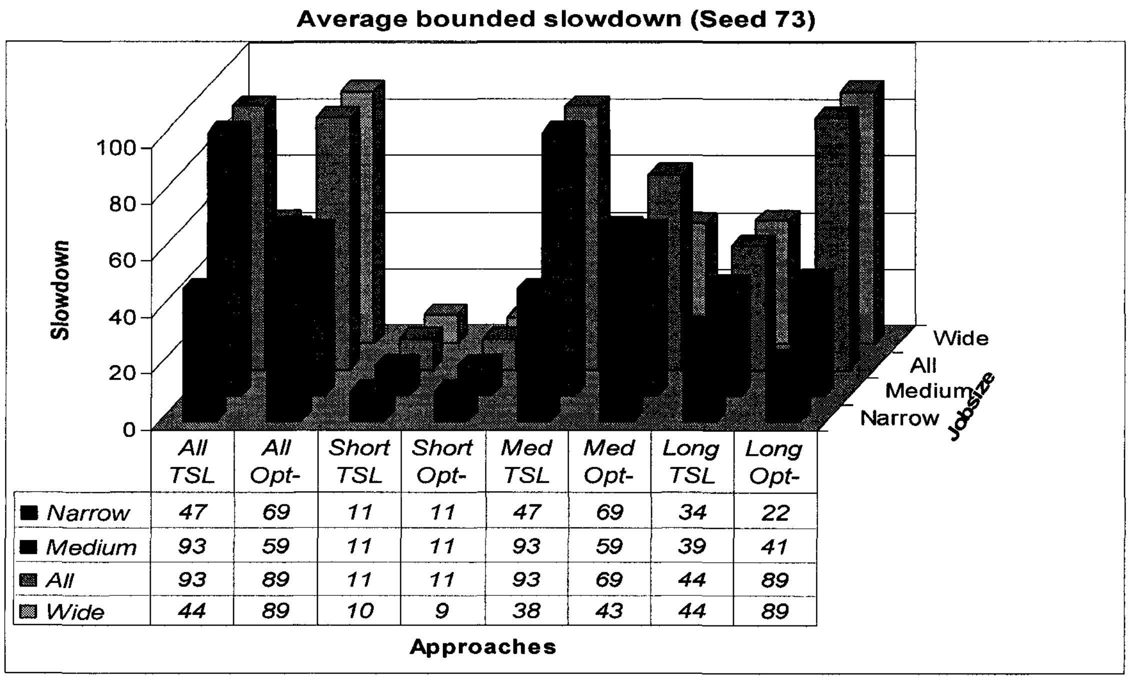
	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ *Narrow*	*3.6*	*3.29*	*3.45*	*3.16*	*5.03*	*4.67*	*3.05*	*2.59*
■ *Medium*	*3.91*	*3.66*	*3.58*	*3.43*	*5.86*	*5.15*	*3.89*	*3.48*
▨ *All*	*4.44*	*4.14*	*3.59*	*3.38*	*8.47*	*7.78*	*4.29*	*3.91*
▨ *Wide*	*7.99*	*7.56*	*4.39*	*4.35*	*12.47*	*11.63*	*5.1*	*4.78*

Figure 35 - Comparison of average slowdown without smart backfilling (SDSC BLUE)



	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow								
■ Medium	6.36	5.68	5.39	5.31	8.13	6.33	13.75	9.42
▨ All	7.84	7.16	6.59	6.5	8.42	7.02	21.38	17.81
▨ Wide	12.64	11.98	12.31	12.23	9.15	8.79	23.53	20.18

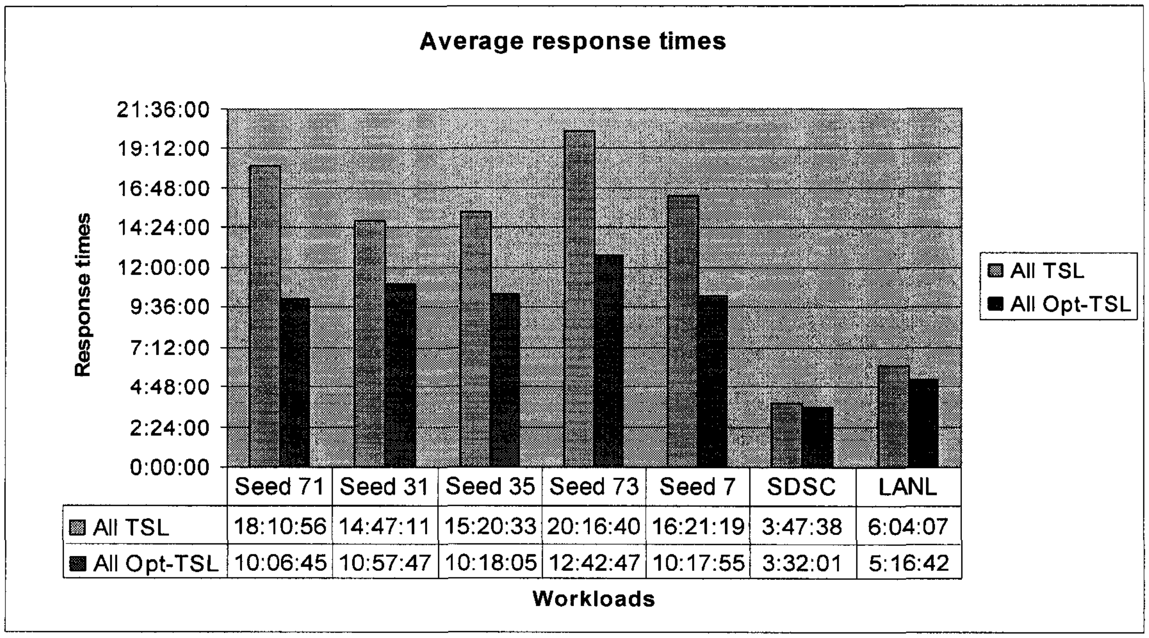
Figure 36 - Comparison of average slowdown without smart backfilling (LANL Trace)



	All TSL	All Opt-	Short TSL	Short Opt-	Med TSL	Med Opt-	Long TSL	Long Opt-
■ Narrow	47	69	11	11	47	69	34	22
■ Medium	93	59	11	11	93	59	39	41
▨ All	93	89	11	11	93	69	44	89
▨ Wide	44	89	10	9	38	43	44	89

Figure 37 - Comparison of average bounded slowdown without smart backfilling (worst case)

Though optimized scheduling has been able to deliver on the average 25% improvements for long jobs and 15% for medium jobs, the short jobs experience only little improvements. The worst case behavior for average bounded slowdown is shown in Figure 37. Medium jobs were found to be served relatively better than long jobs for worst cases. Long jobs on the other hand was seen considerably worse.

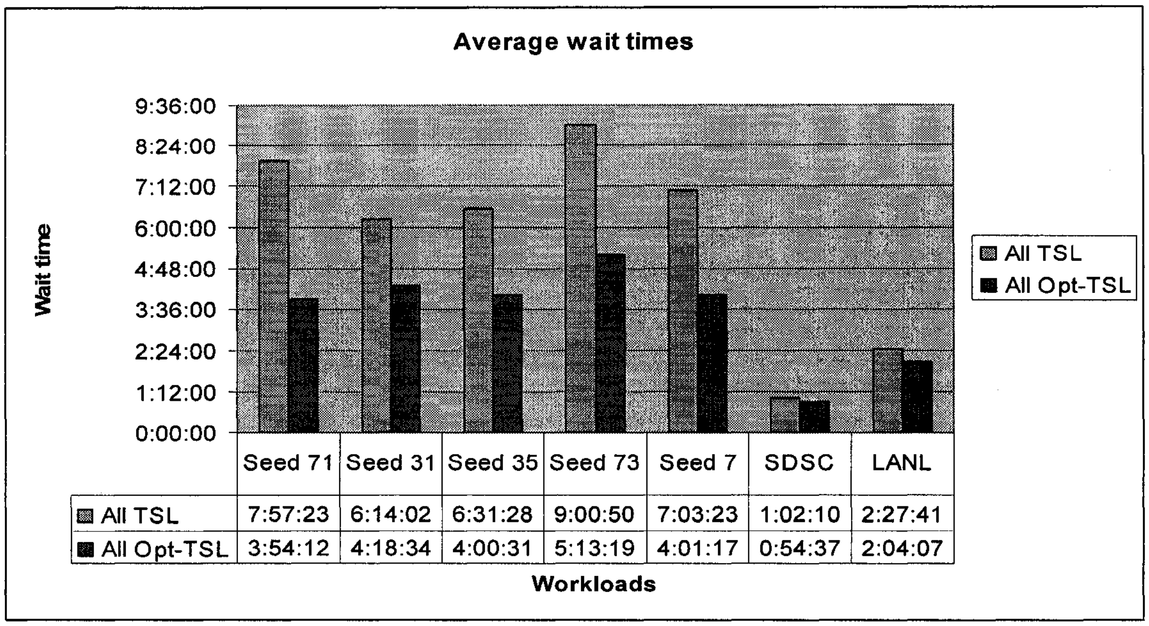


	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7	SDSC	LANL
▨ All TSL	18:10:56	14:47:11	15:20:33	20:16:40	16:21:19	3:47:38	6:04:07
■ All Opt-TSL	10:06:45	10:57:47	10:18:05	12:42:47	10:17:55	3:32:01	5:16:42

Figure 38 - Comparison of average response time without smart backfilling

The average response time and wait times without smart backfilling have been improved by more than 12%. The comparison graphs are shown in Figure 38 and

Figure 39.

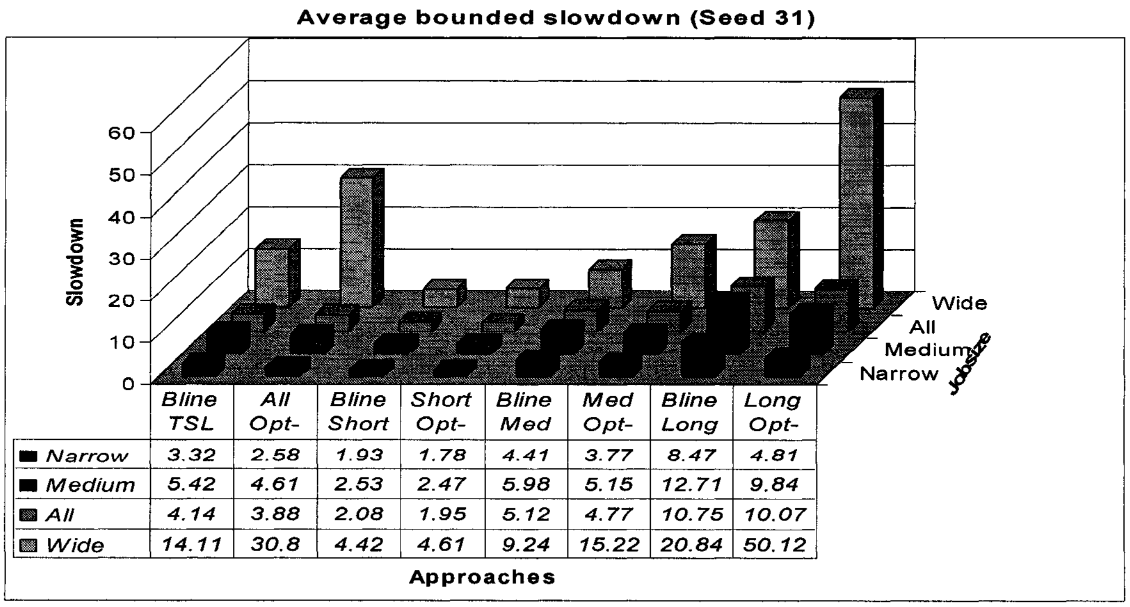


	Seed 71	Seed 31	Seed 35	Seed 73	Seed 7	SDSC	LANL
▨ All TSL	7:57:23	6:14:02	6:31:28	9:00:50	7:03:23	1:02:10	2:27:41
■ All Opt-TSL	3:54:12	4:18:34	4:00:31	5:13:19	4:01:17	0:54:37	2:04:07

Figure 39 - Comparison of wait times without smart backfilling

9.5 Comparison against bottom line approach

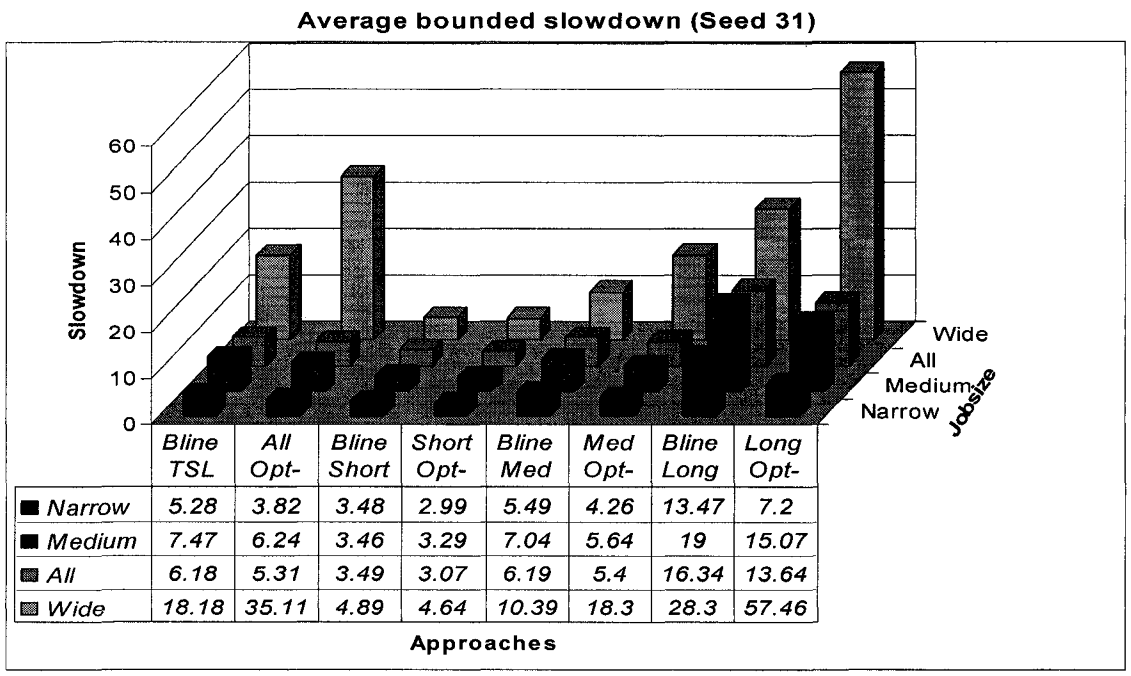
To further emphasize the efficiency of our approach, we are showing a comparison of our results against an approximation of the approach formulated in [40].



	Bline TSL	All Opt-	Bline Short	Short Opt-	Bline Med	Med Opt-	Bline Long	Long Opt-
■ Narrow	3.32	2.58	1.93	1.78	4.41	3.77	8.47	4.81
■ Medium	5.42	4.61	2.53	2.47	5.98	5.15	12.71	9.84
▨ All	4.14	3.88	2.08	1.95	5.12	4.77	10.75	10.07
▨ Wide	14.11	30.8	4.42	4.61	9.24	15.22	20.84	50.12

Figure 40 - Comparison of our optimized approach with bottom line approach

This makes sense as we have similar objectives of reducing the average relative response times using discrepancy search based approach. Our results have stood fairly well for narrow and medium jobs as illustrated in the comparison charts below (Figure 40 and Figure 41). As seen before, the wide jobs are suffering due to the lesser optimization options available with limited group sizes.



	Bline TSL	All Opt-	Bline Short	Short Opt-	Bline Med	Med Opt-	Bline Long	Long Opt-
■ Narrow	5.28	3.82	3.48	2.99	5.49	4.26	13.47	7.2
■ Medium	7.47	6.24	3.46	3.29	7.04	5.64	19	15.07
▨ All	6.18	5.31	3.49	3.07	6.19	5.4	16.34	13.64
▨ Wide	18.18	35.11	4.89	4.64	10.39	18.3	28.3	57.46

Figure 41 - Comparison of our optimized approach with bottom line approach without smart backfilling

Figure 40 shows the comparison including smart backfilling. On the other hand, smart backfilling is switched off in Figure 41. The results point out that we were able to outperform their approximate approach using our optimization methods. We were able to produce improvements resulting in an average 10% for all sizes (except wide) of long and medium jobs.

10 . Conclusion and Future work

In our approach, we have presented a group-based optimization algorithm designed for parallel job scheduling systems for a pre-emptive time-shared environment. The focus of our framework was to achieve consistent improvement in average bounded slowdowns while also not compromising machine utilization.

As expected, the search-based optimizer was able to churn out good schedules and branch-and-bound search. Searching hierarchically using heuristics and domain knowledge paved the way for efficient movement through search space. The idea of creating a plan of virtually scheduled jobs and its integration to a time sharing scheduler required recalculating the plan. Dynamic backfilling upon submission has proved to benefit the average wait times. Conservative backfilling has been used throughout the optimization framework so as not to alter the positions of the optimized plan jobs. Comparison against the bottom line approach further emphasized the efficiency of our approach.

The experimental results are quite promising after integrating our approach to SCOJO-PECT, a course-grain time-sharing scheduler. We were able to achieve an average of 16% better results for slowdown while only compromising a little over 1% in utilization.

Due to various backfilling policies, re-optimizing at a point where the schedule might go critical could be a future addition. We foresee situations where user submitted runtime over estimates can affect the optimized job ordering. Evaluating optimization benefits in such scenarios are also planned. Having an intelligent selection of certain jobs from the group and fixing their schedule times while permuting other jobs also seems promising.

References

[1] A. Anjum, R. McClatchey, H. Stockinger, A. Ali, et.al. "Bulk Scheduling with DIANA Scheduler", IEEE Transactions on Nuclear Science, Vol. 53, No. 6, Dec 2006, pp. 3818-3829

[2] L. Barsanti, A.C. Sodan, "Adaptive Job Scheduling via Predictive Job Resource Allocation", Twelfth Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'06), Saint Malo, France, Jun 2006

[3] C. Blum, A. Roli, "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison", ACM Computing Surveys (CSUR), Vol. 35, No. 3, Sep 2003

[4] T.D. Braun, H.J. Siegel, N. Beck, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems", Journal of Parallel and Distributed Computing, Vol. 61, No. 6, pp. 810-837, 2001

[5] W. Cirne, F. Berman, "When the Herd is Smart : Aggregate Behavior in the Selection of Job Request", IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 2, Feb 2003

[6] P. Dutot, G. Mounie, L. Eyraud, D. Trystram, "Bi-Criteria Algorithm for Scheduling Jobs on Cluster Platforms", ACM Symposium on Parallel Algorithms and Architectures (SPAA), Spain, Jun 2004, pp. 125-132

[7] B. Esbaugh, A.C. Sodan, "Preemption and Share Control in Parallel Grid Job Scheduling", CoreGrid Workshop on Grid Middleware (in conjunction with ICS), Dresden, Jun 2007, to appear in Springer

[8] A. V. Fishkin, K. Jansen and M. Mastrolilli, "Grouping techniques for scheduling problems: Simpler and Faster", Proceedings 9th Annual European Symposium on Algorithms, Lecture Notes in Computer Science, Vol. 2161, Springer-Verlag, 2001, pp. 206-217

[9] D.G. Feitelson, "Parallel Workloads Archive", As retrieved on Dec 10th, 2007, http://www.cs.huji.ac.il/labs/parallel/workload/

[10] W.D. Harvey, M.L. Ginsberg, "Limited Discrepancy Search", Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Vol. 1, Aug 1995, pp. 607-615

[11] L. He, S.A. Jarvis, D.P. Spooner, X. Chen, G.R. Nudd, "Dynamic Scheduling of Parallel Jobs with QoS Demands in Multiclusters and Grids", Proceedings of the

Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04), Nov 2004, Pittsburgh, USA

[12] W. Karoui, M.J. Huguet, P. Lopez, W. Naanaa, "YIELDS: A Yet Another Improved Discrepancy Search for CSP's", Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Brussels, May 2007, pp. 99-111

[13] R.E. Korf, "Improved Limited Discrepancy Search", Thirteenth National Conference on Artificial Intelligence (AAAI), Volume 1, Portland, USA, Aug 1996, pp. 286-291

[14] B.G. Lawson, E. Smrini, "Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems", Lecture Notes in Computer Science, Vol. 2537, 2002

[15] W. Leinberger, G. Karypis, V. Kumar, "Job Scheduling in the presence of Multiple Resource Requirements", Conference on High Performance Networking and Computing, Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, Oregon, USA, 1999, pp. 47

[16] M. Li, B. Yu, M. Qi, "PGGA : A Predictable and Grouped Genetic Algorithm for Job Scheduling", Future Generation Computer Systems, Vol. 22, No. 5, Apr 2006, pp. 588-599

[17] U. Lublin, D.G. Feitelson, "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs", Journal of Parallel and Distributed Computing, Vol. 63, No. 11, Nov 2003, pp.1105-1122

[18] M. Maheswaran, H. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto heterogeneous Computing Systems", Journal of Parallel and Distributed Computing, Vol. 59, No. 2, Nov 1999, pp. 107-131

[19] V.D. Martino, M. Mililotti, "Scheduling in a Grid computing environment using Genetic Algorithms", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02), Florida, Apr 2002

[20] M. Mezmaz, N. Melab, E-G. Talbi, "Using the Multi-Start and Island Models for Parallel Multi-Objective Optimization on the Computational Grid", Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06), Amsterdam, Dec 2006

[21] M. Moore, "An Accurate and Efficient Parallel Genetic Algorithm to Schedule Tasks on Cluster", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, Apr 2003

[22] J.E. Moreira, W. Chan, L.L. Fong, H. Franke, M.A. Jette, "An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments", Proceedings of ACM/IEEE Supercomputing (SC), Nov 1998

[23] G. Mounie, C. Rapine, and D. Trystram, "Efficient approximation algorithms for scheduling malleable tasks", Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, 1999, pp. 23-32

[24] A.W Mu'alem, D.G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling", IEEE Transactions on Parallel and Distributed Systems, Vol. 12, No. 6, 2001, pp. 529-543

[25] N. Muthuvelu, J. Liu, N.L. Soe, S. Venugopal, A. Sulistio, R. Buyya,"A Dynamic Job Grouping-Based Scheduling for Deploying Applications with Fine-Grained Tasks on Global Grids", Australasian Workshop on Grid Computing and e-Research (AusGrid 2005), New Castle, Jan 2005, pp. 41-48

[26] A.J. Page, T.J. Naughton, "Dynamic Task Scheduling using Genetic Algorithms for Heterogeneous Distributed Computing", Proceedings of the Nineteenth ACM/IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Denver, Apr 2005

[27] C.L. Pape, P. Baptiste, "Heuristic Control of Constraint-Based Algorithm for the Preemptive Job-Shop Scheduling Problem", Journal of Heuristics, Vol. 5, 1999, pp. 305-325

[28] E.W. Parsons, K.C. Sevcik, "Implementing Multiprocessor Scheduling Disciplines", Proceedings of IPPS Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), April 1997, Lecture Notes in Computer Science, Vol.1291, Springer-Verlag

[29] B.G. Patrick , M. Jack, "Equi-partitioning versus Marginal Analysis for Parallel Scheduling", Proceedings of the Fourth International Conference on Parallel And Distributed Computing, Applications and Technologies (PDCAT'03), Chengdu, China, Aug 2003

[30] G. Sabin, M. Lang, P. Sadayappan, "Moldable Parallel Job Scheduling using Job Efficiency : An Iterative Approach", Twelfth Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'06), Saint Malo, France, Jun 2006

[31] S. Setia, M. Squillante, V.K. Naik, "The Impact of job memory requirements on gang-scheduling performance", Performance Evaluation Review, Vol. 26, No. 4, 1999, pp. 30-39

[32] C.Y. Shen, Y.H. Pao, P.P.C. Yip, "Scheduling Multiple Job Problems with Guided Evolutionary Simulated Annealing Approach", International Conference on Evolutionary Computation, Orlando, USA, Jun 1994, pp. 628-633

[33] E. Shumeli, D.G. Feitelson, “Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling”, Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, Vol. 2862, 2003, pp. 228-251

[34] J. Skovira, W. Chan, H. Zhou, D.A. Lifka, “The EASY – Loadleveler API Project”, Lecture Notes in Computer Science; Vol.1162, pp. 41-47

[35] D. Talby, D.G. Feitelson, “Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling”, Thirteenth International Parallel Processing Symposium and Tenth Symposium on Parallel and Distributed Processing (IPPS/SPDP‘99), 1999, Puerto Rico

[36] X. Tang, S.T. Chanson, “Optimizing static job scheduling in a network of Heterogeneous Computers”, Proceedings of 2000 International Conference on Parallel Processing, Aug 2000, pp. 373 – 382

[37] J. Turek, J.L. Wolf, P.S. Yu, “Approximation Algorithms for Scheduling Parallelizable Tasks”, Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, 1992, pp. 323-332

[38] D.C. Vanderster, N.J. Dimopoulos, R.Parra-Hernandez, “Evaluation of Knapsack-based Scheduling using the NCAPI JOBLOG”, Proceedings of 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS’06), St.John’s, May 2006

[39] S. Vasupongayya, “Improving Search-based Parallel Job Scheduler", Parallel and Distributed Processing Techniques and Applications (PDPTA'06), Las Vegas, Jun 2006

[40] S. Vasupongayya, S.-H. Chiang, B.Massey, "Search-based Job Scheduling for Parallel Computer Workloads", IEEE International Conference on Cluster Computing (Cluster 2005), Boston, Sep 2005

[41] T. Walsh, “Depth-Bound Discrepancy Search”, International Joint Conference on Artificial Intelligence (IJCAI-97), Nagoya, Japan, Aug 1997, pp. 1388-1395

[42] A.T. Wong, L. Oliker, W.T.C. Kramer, T.L. Kaltz, D.H. Bailey, “ESP: A System Utilization Benchmark”, Proceedings of ACM/IEEE Supercomputing Conference (SC), Dallas, Texas, Nov 2000

Vita Auctoris

Name:		Arun Kumar Kanavallil
Place of Birth:		Ernakulam, India
Education:		
	2005-2007	M.Sc., Computer Science University of Windsor Windsor, Ontario, Canada
	2000-2004	Bachelor of Technology in Computer Science and Engineering (B.Tech) University of Kerala Trivandrum, India
Work Experience:		
	2004-2005	Project Staff Center for Development of Advanced Computing (C-DAC) Pune, India