LOMARC: Look ahead matchmaking for multi-resource coscheduling.

Lei Lan
University of Windsor

LOMARC – Look Ahead Matchmaking for Multi-Resource Coscheduling

By

Lan, Lei

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

Windsor, Ontario, Canada
2004

Your file Votre référence
ISBN: 0-612-92497-1
Our file Notre référence
ISBN: 0-612-92497-1

Abstract

Hyper-Threading (HT) provides a new possibility for job coscheduling without context switch and without the cost for coordinating processes of one parallel job. However, HT achieves high processor throughput at the expense of reducing the performance of the individual process. Since the hardware resources are actually shared between two coscheduled jobs, the resource contention will harm the performance of each job. Most scheduling approaches only focus on the CPU without considering the impact on other resources.

In this thesis we present LOMARC, a space-time sharing approach that takes multiple resources, including CPU, I/O, memory and network, into consideration for job coscheduling on HT processors. To improve resource utilization and reduce job response times, LOMARC matches two jobs with complementary resource requirements to coschedule. Our approach partially reorders the waiting job queue by lookahead to increase the possibility of finding a good match. LOMARC also generalizes for standard CPUs, using an adjusted matching scheme and only focusing on hiding I/O latency. In addition, LOMARC incorporates standard scheduling approaches such as priority ordering, aging and backfilling. In our simulation experiment, we use a realistic workload model to provide the convincing results. Our experimental results demonstrate that LOMARC delivers better performance than the standard space sharing approach and the other two job coscheduling approaches for HT processors. The performance gain is mainly due to an increased possibility of coscheduling two complementary jobs by looking ahead on the waiting queue.

To my mother, Shuxian Zhang

my father, Zujie Lan

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my supervisor, Dr. Angela Sodan. The work could not be achieved without her extensive guidance and constant encouragement. Also I would like to thank all the committee members, Dr. James Gauld, Dr. Scott Goodwin, Dr. Peter Tsin, for their valuable time and comments, and I must specially thank Dr. Gauld for his additional help in polishing up the English writing of this thesis.

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

I would like to thank Xuemin Huang for his help in providing the basic simulator for this thesis work and sharing his valuable experiences.

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

Table of Contents

List of Tables

List of Figures

1. Introduction

Job scheduling for parallel systems has been the subject of many research activities for decades. A scheduler for a parallel system decides not only *when* a process should run, but also *where* the process should run. The scheduling strategy of a parallel system is essential for how well the system can provide the service to the users, because it decides the resource allocation to applications and the efficiency of resource utilization. There are varieties of scheduling strategies for parallel applications in parallel systems proposed and implemented. The divergence is due to different assumptions for the cost metrics model, machine model and application model. It is believed that there is no single best solution for all different situations [Feitelson97C].

The common goals of scheduling in a parallel system can be seen according to two views: the user perspective and the system perspective. For the user perspective, how soon a submitted job can finish is important. For the system perspective, how efficient the system resources are utilized is important. Although improved system utilization may lead to improved response time, there is a trade-off between these two goals. To evaluate how well a scheduler achieves these goals, some formalized metrics are developed, the details of which are discussed in Chapter 2.

Space sharing and time sharing are two basic types of scheduling approaches. In space sharing, processors are partitioned into disjoint subsets, and each subset is dedicated to one job. Time sharing means multiple processes are allocated to one processor, and each processor switches among the processes assigned to it using time slices. In pure time sharing, the schedule of processes on one processor is independent from other processors. There are pros and cons for both space sharing and time sharing. Space sharing allows exclusive resource allocation, and therefore, gives the best execution time for each job and has little system overhead on the context switch. The main problems of space sharing are the fragmentation and reduced response time caused by inefficient packing schemes. Extensive research has been done to optimize space sharing strategies [Feitelson97A]. Time sharing provides more flexible resource sharing and better responsiveness to users. However, standard time sharing is not always suitable for parallel jobs. Usually, parallel jobs require that the process working set of one job is scheduled at the same time on different processors, which cannot be guaranteed in time sharing in that the scheduling on each processor is independent of the others. Strategies using combination of space and time sharing, i.e. space-time sharing, are developed to address the problems in pure space and time sharing, and are reported for better response time and utilization [Feitelson97A]. Chapter 2 discusses these different strategies in more detail.

Before the following discussion, we need to clarify the meaning of "coscheduling". Generally, the term of "coscheduling" in the literature can refer to two different concepts. One is the coscheduling of processes belonging to one parallel job, which means to schedule the processes on different processors at the same time to facilitate the communication or synchronization among them, e.g. in [Ousterhout82]. The other one is the coscheduling of different jobs, which means to schedule different jobs at the same time to share certain hardware resources, e.g. in [Snavely02]. To eliminate the confusion, in the context of this thesis, we use "coscheduling" for the first case and "job coscheduling" for the other case. Note that our approach focuses on job coscheduling.

The idea of coscheduling is first introduced in [Ousterhout82]. Parallel jobs consist of multiple processes that execute on different processors and coordinate with each other by communication or synchronization. It is important to keep the coordinated process working set running simultaneously to make a parallel job progress. There are two situations when a process is waiting for a message from another process that is not scheduled at the same time. First, if the process simply uses busy waiting without relinquishing the CPU, the CPU time is wasted by doing nothing. The other situation is that the process will be suspended and the CPU switches to another process. In this case, context switch cost is increased and process thrashing [Ousterhout82] can happen. In both situations, system performance will be degraded.

To guarantee the good performance of a parallel job, the processes that interact with each other should be scheduled to run at the same time. Coscheduling is developed to improve the performance of parallel applications in a time-shared system. Gang scheduling [Feitelson92], dynamic coscheduling [Sobalvarro97] and implicit coscheduling [Dusseau96] are three important strategies designed to achieve coscheduling. More details about coscheduling are presented in Chapter 2.

In a parallel system, usually there will be a mix of different applications with different resource requirements. To improve the overall system performance, a scheduler needs to consider the contention on resources other than CPUs as well. Most current scheduling research focuses on the CPU only. While some work considers the memory impact on scheduling, and some others consider I/O and network impacts, there are few scheduling strategies that take multiple resources into consideration.

Hyper-Threading (HT), developed by Intel, is a form of simultaneous multi-threading technology (SMT) where two processes of software applications can run simultaneously on one processor. However, HT achieves high processor throughput at the expense of reducing the performance of the individual process [Dorai02]. Since the hardware resources are actually shared between two processes, the resource contention will harm the performance of each process. When choosing two jobs to coschedule on HT processors, we need to consider all resource requirements of those two jobs.

Our approach, LOMARC, aims to take all resource requirements of applications into consideration for job coscheduling to fully exploit the benefit provided by the HT technology. To improve utilization and reduce response times, we match two jobs with complementary resource requirements to coschedule. The argument for this idea is as follows. First, two jobs with complementary resource requirements will have less resource contention; hence, the performance of coscheduled jobs will not be degraded. Second, coscheduling jobs with complementary resource requirements will achieve better overall resource utilization. When choosing a match for one job, we also consider the utilization gain and response time impact to other jobs resulting from the matching, and choose the best one according to a combined utilization gain and response time impact value. LOMARC can also be generalized for standard CPUs by using an adjusted matching scheme.

The rest of this report is organized as follows. Background issues are discussed in Chapter 2. Chapter 3 focuses on the discussion of resource impacts on job scheduling. Chapter 4 introduces the Hyper-Threading technology, which is one major motivation for our work. Chapter 5 describes the LOMARC algorithm in detail. We present the simulation details and experiment results in Chapter 6. Finally, the conclusion for this thesis is presented in Chapter 7.

2. Background Issues

In this chapter, we will review the existing scheduling schemes, and classify scheduling technologies into three categories, namely space sharing, time sharing and a combination of both. After this, we will briefly discuss some basic issues that have impacts on scheduling design, including metrics models and workload characteristics.

2.1. Review of scheduling strategies in parallel systems

There are three main classes of scheduling approaches according to how the resources are shared. In a parallel system, the sharing is in two dimensions: space, i.e. processors; and time. Space sharing and time sharing are two basic classes of approaches. The combination of space and time sharing is another class of scheduling strategies that aim to address the problems in pure space and time sharing.

2.3.1. Space sharing

In space sharing, processors are partitioned and each subset of processors is allocated solely to one job. This approach mainly deals with how to pack the jobs to fit into available processors to achieve better resource utilization. In the basic space sharing approach, the number of processors allocated to a job is fixed, and each process runs on the allocated processor until completion.

The simplest space sharing strategy is First Come First Serve (FCFS). In this approach, jobs are allocated to their required number of processors when available in the job submission order. The main problem for this approach is the fragmentation, which means a set of processors are left idle for some period of time. One job blocking the queue of jobs, because an insufficient number of processors are available for it, will prevent all later jobs to be scheduled. Hence, this situation also increases the job waiting time and response time.

To address the problem of both system utilization and job response time, intensive research has been done to optimize the space sharing strategy [Feitelson97A]. Backfilling [Feitelson98B] [Zhang00] is one important approach among these efforts. Backfilling is a strategy developed to address the fragmentation problem in space sharing by allowing some small jobs from the back of the queue to move ahead to fill the empty space. Figure 2.1 [Zhang02] illustrates how backfilling can reduce fragmentation. The job numbers in the figure correspond to job positions in the waiting queue. In Figure 2.1(a), the empty space between T_1 and T_2 is called fragmentation, and in Figure 2.1(b), Job 5 is backfilled to utilize the empty space. There are basically two types of backfilling. The first one is conservative backfilling [Feitelson98B], in which a job can be chosen to backfill only if it will not delay any job in front of it in the queue. Another one is called EASY backfilling [Lifka95]; this approach relaxes the constraint for choosing backfill jobs and allows a job to backfill as long as it does not delay the first job in the queue. Both backfilling approaches have been proven to improve system utilization and reduce response time significantly relative to FCFS. The main limitation of backfilling is that it depends on the knowledge of job runtime, which is usually obtained from user estimation and is not accurate.

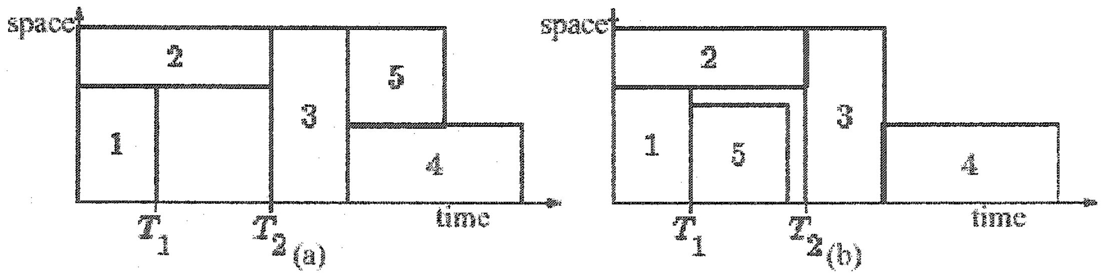


Figure 2.1. Backfilling. (from [Zhang02])

To illustrate the details of the EASY Backfilling algorithm, we can use Figure 2.1 again. At T_1, the event of Job 1 termination happens, and Job 3, 4 and 5 are already in the waiting job queue with Job 3 as the first job in the queue. The size of Job 3 is larger than the available free space at T_1, so it cannot be scheduled at this moment.

We then need to find another job in the waiting queue to do the backfilling. First, we compute the possible start time of Job 3, which is T_2, when Job 2 will terminate and free space is large enough for Job 3. Second, we look through the waiting queue to find the first job that has size no larger than the current free space and runtime no longer than T_2- T_1, which means the job will terminate before Job 3 starts, and hence won't delay Job 3. In the example in Figure 2.1, Job 5 satisfies both of these conditions, so it is chosen for backfilling

Since backfilling improves utilization and response times significantly compared to FCFS, more sophisticated strategies are proposed based on standard backfilling. Slack-based backfilling [Talby99] supports job priority, and is more aggressive when reordering the waiting job queue for backfilling. It allows a job to backfill if it does not delay any other job longer than its slack time. Results show that this approach can have a better job waiting time, and is responsive to different priority requirements. Another approach based on backfilling is presented in [Shmueli03]. In this approach, instead of considering one job at a time, it uses a certain lookahead window, and examines all jobs within the window for possible combinations of jobs for backfilling. The algorithm is implemented using dynamic programming.

In addition to backfilling, there are also many other strategies aimed to optimize the basic space sharing approach [Feitelson97A]. Most of these efforts try to reorder the job queue to improve system utilization and average response time. Research results show that sorting job queue using shortest-job-first [Perkovic00] can reduce mean response time, but has the problem of starvation when short jobs dominate the workload. Sorting job queue with job size also helps to improve utilization and response time. Research results [Perkovic00] suggest sorting job queue with LPFS (Least-Processor-First-Served) can improve the performance significantly. One advantage of sorting the queue according to job size is that it does not need the information for job runtime, thereby making the scheduling more realistic.

In spite of all the above mentioned efforts in optimizing standard space sharing approaches, the system utilization and response times for these types of approaches are still not optimal. More sophisticated strategies are required to improve the system performance. Adaptive and dynamic partitioning are two types of schemes motivated by this goal. Note that both adaptive and dynamic partitioning depend on the application types; more precisely, they are only relevant for moldable and malleable jobs.

Adaptive partitioning [Feitelson97A] is one type of approaches that can take advantage of moldable jobs to maximize the system utilization. In this kind of approach, the scheduler can decide the number of processors allocated to one job according to the current workload and available resources. There are two important choices that adaptive partitioning needs to make; first is when to do the adaptation, and second is how to adapt. There are trade-offs between maximizing system utilization and reducing overall response times when making different choices for adaptation. Dynamic partitioning [Feitelson97A] can be done by taking advantage of malleable jobs, which can change the process number during the runtime, thus improving system utilization and efficiency.

Another way to do the dynamic partitioning is through preemption and migration [Feitelson97A]. By full preemption, the scheduler can preempt all processes of one job, and give processors to other jobs with higher priority. Preemption needs additional support from the system to save the execution status of preempted jobs and to resume the job later. When a preempted job resumes, it can run on the same processors as before or run on a different set of processors, i.e. migration. Preemption can improve system utilization in that it allows CPU idle spaces to be filled with less constraint. For example, if the size of a CPU idle space is sufficient for a job to fill in, but the runtime of the job is longer than the idle time, the hole can still be filled by the job, and preemption can be applied to this job when the first job in the queue starts to run. Migration is usually used to repack the jobs to achieve better system utilization. Both preemption and migration can be expensive, because it takes time to save the execution status or to transfer execution context from one node to another (in distributed memory systems).

In summary, basic space sharing is motivated by the desire to minimize operating system overhead such as context switch costs [Feitelson97A], but the overall system utilization and mean job response time are far from optimal. Backfilling is one important approach for optimizing space sharing and can achieve significant improvement in response time and utilization comparing to FCFS. Dynamic partitioning aims to solve the problems in basic space sharing approaches. However, it causes cost increasing in the resource reallocation.

2.3.2 Time sharing

Time sharing is a general approach for parallel systems with independent processing units or nodes, such as a cluster system. In this approach, processes are scheduled independently once allocated to processors. With the use of standard time sharing, the scheduling on each node or processing unit is the same as on a uni-processor, i.e. processor switches among processes using time slicing. The main problems for this strategy are resource contention caused by sharing and the coordination for processes belonging to one parallel job.

There are some advantages of time sharing compared to space sharing approaches. At first, it gives better mean response time, especially for short interactive jobs because large long jobs will not block short jobs as in space sharing. Second, it has better resource utilization because there is no fragmentation problem. Third, it is easily available because a standard time sharing operating system can be installed on independent processing units.

The main problem of time sharing is that each processing unit is scheduled independently. As we discussed in the introduction, parallel jobs usually have coordinated processes that need to run at the same time to guarantee the progress of the whole job. Independent scheduling on each processor in standard time sharing cannot provide the coscheduling required by parallel applications.

Dynamic [Sobalvarro98] and implicit coscheduling [Dusseau98] approaches are developed to address problems of standard time sharing in coscheduling. The main idea is using communication events to guide the coscheduling decision and trying to schedule the communicating processes at the same time. This kind of approach only coschedules the processes when they need to communicate or synchronize with each other. The main difference between these two techniques is that dynamic coscheduling makes the scheduling decision based on the message arrivals, while implicit scheduling decides whether a process needs to continue to be scheduled according to the feedback of its communication or synchronization event.

The main advantages of this kind of coscheduling over gang scheduling are as follows. First, it does not need a central controller for the multi-context switch, so it makes the scheduling scalable and more flexible. Second, it makes the scheduling decision dynamically and can, therefore, adapt to the characteristics of different workload and communication patterns. Third, without using a fixed time slice for every process in a job, it can increase the utilization of the system by latency hiding, and support interactive and I/O intensive applications well.

However, the performance of dynamic and implicit coscheduling cannot compete with gang scheduling for applications with fine-grained communications. Strict coscheduling is desirable for this kind of application, so gang scheduling or space sharing are better solutions.

Another problem of time sharing is the resource contention. When using time sharing, several processes usually are loaded into memory at the same time, and multiple communication contexts need to be saved concurrently. These facts make the resource contention happen when the total resource requirement of jobs, such as memory requirement, is more than the available system resource. We will discuss resource requirement impacts and strategies considering these impacts in Chapter 3.

In summary, although there are many advantages of time sharing approaches, standard time sharing is not suitable for most parallel jobs due to the coscheduling issue. Dynamic coscheduling and implicit coscheduling can address this coscheduling problem under time sharing.

2.3.3. Combination of space and time sharing

Some scheduling strategies combine space sharing and time sharing. This kind of approach can achieve better resource utilization while maintaining the coordination of parallel processes. Gang scheduling is a typical example in this category. In the rest of this section, we will focus our discussion on gang scheduling and approaches developed based on gang scheduling.

Feitelson [Feitelson97A] presented a formal definition of gang scheduling, which includes three features. First, coordinated threads or processes are grouped into gangs. Second, all threads or processes in each gang execute at the same time on different processors, and the relation of threads or processes with processors is a one-to-one mapping. Third, all the threads or processes in a gang context switch simultaneously, using time slicing. The most important feature of gang scheduling is that it allows both space sharing and time sharing.

Extensive research has been done on gang scheduling, including different implementations and measurements of the performance in different systems. Among this research, Feitelson and Rudolph [Feitelson92] implemented gang scheduling on the Makbilan multiprocessor system based on the matrix algorithm presented in [Ousterhout82], and comprehensively examined performance implications of gang scheduling. They pointed out that gang scheduling with busy waiting will especially benefit fine-grained applications. Research results from [Feitelson97B] [Crovella91] both suggest that gang scheduling can achieve better overall system performance compared to pure space sharing scheduling or independent time-sharing scheduling.

Even though many efforts have been made to improve gang scheduling, there are still some disadvantages. First, gang scheduling does not achieve the best utilization of system resources due to the fixed time slice for both CPU and I/O bound jobs [Lee97]. Second, and again due to the fixed time slice, gang scheduling does not provide good response time for short interactive jobs.

To address the problems in traditional gang scheduling some more relaxed gang scheduling approaches are proposed. Loose gang scheduling [Zhou98] and concurrent gang scheduling [Fabricio99] both use two-level scheduling to achieve flexible coscheduling according to job characteristics. The main idea is gang scheduling is implemented at a global level, while at a local level, the local scheduler can have its own freedom in choosing another process to run when a gang scheduled process is blocked on I/O. These strategies can increase the utilization of CPU time and achieve latency hiding. The simulation results in [Fabricio99] show that concurrent gang scheduling has better performance than pure gang scheduling considering both system utilization and throughput.

Most recently, a new approach named Flexible CoScheduling (FCS) is presented in [Frachtenberg03], with the aim of improving resource utilization despite load imbalance. FCS monitors the communication granularity of each job and classifies jobs according to the monitoring results. For different classes of jobs, FCS makes different scheduling decisions. The principle is: for jobs with fine-grained communication, gang scheduling is applied, and for coarse-grained applications, their time slots are shared with other jobs to achieve latency hiding.

In summary, the combination of space and time sharing can achieve better performance than pure space or time sharing approaches.

2.2. Common goals and metrics

It is obvious that a scheduling strategy should try to use system resources efficiently and satisfy the requirements of different jobs and users. On the one hand, a scheduler needs to satisfy the users who usually want their jobs to be scheduled as soon as possible or to meet certain deadlines. Also the fairness among different users and jobs is an important factor that needs to be considered by a scheduler to satisfy the users as a whole. On the other hand, to maximize system utilization, the scheduler also needs to reduce resource idle time, e.g. CPU idle time, and system overhead, e.g. context switch overhead. To evaluate how well a scheduler achieves the goals, some formalized metrics are developed. Makespan, response time, relative response time, bounded response time and utilization efficiency are some important and well accepted metrics for evaluating scheduling strategies. The definitions of these metrics are as follows [Feitelson98A].

Makespan: the time for all jobs in the measured workload to terminate.

Response time: the time elapsed between the submission and the end of the job execution.

Relative response time: response time divided by actual running time.

Relative bounded response time: response time divided by actual running time or a lower bound runtime, whichever is larger. This metric is developed to adjust the relative response time for extremely short jobs.

Utilization Efficiency: $E = \frac{\sum_i p_i t_i}{PT}$

where p_i and t_i are the number of allocated processors and execution time (in a dedicated setting), respectively, for the *ith* job, T is the termination time for the whole workload and P is the number of all available processors. The meaning of this metric is the ratio between effective processing time for the workload and the whole available processing time.

How to choose good metrics for evaluating and comparing different scheduling approaches is still an open problem, because it depends on the real workload, system environment, and user requirements [Feitelson98A]. For different kinds of jobs, it usually is different with regards to which metrics are important. For example, for short, interactive jobs, the response time is normally most important for the users, and for some long batch jobs, maximizing the system utilization will help for the overall performance [Feitelson98A]. Some metrics will depend on real workload characteristics in a system; for example, utilization and makespan are directly related to job arrival rate [Feitelson98A], where a high job arrival rate means a heavier load.

2.3. Workload and job characteristics

Workload and job characteristics are important factors that impact the scheduling design. Usually, jobs in a parallel system can be classified into short, medium and long according to their runtimes, and small, medium and large according to their sizes, i.e. required numbers of processors. Workload reflects how different kinds of jobs are mixed, and usually it describes the percentage of different jobs in the whole load.

In [Subhlok96], the authors measured workloads in a 512 node IBM SP2 at Cornell Theory Center, a 96 node Intel Paragon at ETH Zurich, and a 512 node Cray T3D at Pittsburgh Supercomputing Center respectively. They found some common characteristics for these supercomputers. First, machine usage is dominated by medium size jobs. Second, a large amount of jobs use power-of-2 number of nodes for execution. Third, short jobs constitute a majority of the whole workload. This information is suggestive to scheduler design for these kinds of supercomputers.

Other than runtime and size of a job, there are some other characteristics of jobs that can impact the design of scheduling strategies. One important factor is the flexibility of jobs in their size configuration. There are basically four types of jobs according to their flexibility [Feitelson97C].
Rigid jobs: these jobs have fixed job size and cannot be changed by the scheduler.
Moldable jobs: the sizes of these jobs can be decided by the scheduler when jobs first start to run and cannot be changed during the execution.
Evolving jobs: the sizes of these jobs will be different in their different execution phases, and are decided by the jobs themselves rather than the scheduler.
Malleable jobs: the sizes of these jobs may change during the job's execution according to the decision of the scheduler.

More sophisticated schedulers can take advantage of moldable and malleable jobs to fully utilize the system resource. For example, adaptive partitioning uses moldable jobs to increase system utilization, while dynamic partitioning uses malleable jobs.

Besides the CPU requirement of jobs, their requirements in memory, I/O and network bandwidth also play an important role in impacting the scheduling performance. Especially in a time sharing environment, where different jobs share all resources, resource contentions can have a significant impact on both system and application performance. We will discuss these in detail in Chapter 3.

3. Resource Impacts on Job Scheduling

The scheduling strategies discussed in Chapter 2 only consider the CPU requirement of processors. In a shared system, usually there will be a mix of different jobs with different resource requirements. For example, some jobs are computation intensive, some are data intensive and require a large amount of memory space, some are I/O intensive and some consume excessive networking resources due to intensive communication. To improve the overall system performance, a scheduler needs to consider the contention on resources other than CPUs as well. The rest of the chapter will discuss the scheduling strategies considering different aspects of resource contentions.

3.1. Memory Impact on Scheduling

Basically, memory can impact the performance of parallel processing in two respects: the first is the memory access locality [Chandra94] and the second is the available physical memory size of the nodes on which a parallel process is running [Peris94].

In [Peris94], the authors developed a model to examine the performance trade-off between the optimal allocation, which reduces the processor allocation to a parallel job in a heavy workload, and the memory contention resulting from the allocation decision. Analysis results [Peris94] suggest that memory requirements should be considered in making processor allocation decisions. When the memory requirement of a process is too large to fit in the physical memory space of the node, there will be a large overhead for demand paging. In [Burger96], the author gave an evaluation for the demand paging trade-offs in parallel processing. The test results show that demand paging degrades performance of parallel applications. This is because when a process of a parallel job encounters a page fault, it will delay other coordinated processes as well, due to the synchronization requirement. If we simply switch the processors to another parallel job whenever a page fault happens, there will be very high overhead for context switching. As a result, Burger [Burger96] suggested that page faults should be avoided in parallel processing.

Realizing the memory impact for the performance of parallel application, some scheduling strategies are developed with the consideration of memory requirements of applications. In [Setia99], the authors used simulations to test the memory impact for the performance of gang scheduling and found out that a long-term scheduler strategy with memory consideration will benefit the performance of gang scheduling. Instead of using FCFS (First Come First Serve) queue, the simulation suggests using Smallest Memory First (SMF) as the long term scheduling strategy to reduce the mean response time for reasons similar to the Shortest Job First scheduling.

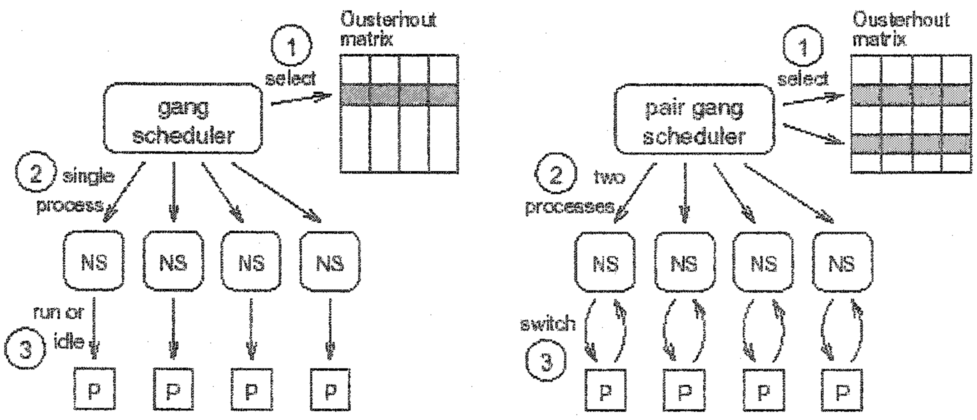
Another gang scheduling approach with memory concern is presented in [Batat00]. In order to take the memory requirements of the job into consideration, schedulers need to have the knowledge of such requirements. This can be achieved by estimating the memory requirements based on the memory usage from previous runs of a job, or using static information in the execution file of a job, if it runs at the first time. When allocating the nodes to the job, the scheduler only schedules it if there is enough memory space.

3.2. I/O Impact on Scheduling

I/O requirement of an application is another important aspect that a scheduler needs to consider for achieving better system performance. Research results [Smirni98] on characterizing the I/O behavior of parallel applications show that I/O requirements of a parallel job can have a significant impact on performance. System performance is related not only to how the processors are allocated to jobs, but also depends on the configuration of the I/O system, such as the available disk capacity and how the data is distributed on the disks [Rosti98]. When jobs need to compete for I/O, the performance of a job may decrease because the waiting time for I/O requests being served will increase. Therefore, the job execution time will increase as well. To improve the overall system performance, a scheduler should try to overlap I/O processing and computation.

In [Lee97], the authors examined I/O impact for gang scheduling, and showed that the performance of I/O demanding jobs suffers under traditional gang scheduling. The reason for this is that gang scheduling sets fixed time slots for every job, so an I/O bound job will waste a lot of CPU time when it is blocked for I/O before its time slot finishes. On the other hand, if the time slot for an I/O bound job finishes just before the I/O request is made, the I/O resource will remain idle for a whole time slot. To improve the utilization of both I/O resource and CPU, a more flexible coscheduler is needed. Such a flexible coscheduler should choose some less coscheduling demanding process to fill the CPU fragments due to the I/O intensive job blocking in its time slot. Also, I/O intensive jobs should have higher priority so they can preempt computation intensive jobs for better I/O resource utilization.

Paired gang scheduling [Wiseman03] is a strategy for solving the problem of traditional gang scheduling presented in [Lee97]. Figure 3.1. [Wiseman03] shows how paired gang scheduling differs from the traditional gang scheduling. In paired gang scheduling, the central scheduler selects one I/O bound job and one CPU bound job and packs them together into one time slot. On each node, processors switch between these two processes according to the decision of the local scheduler, which gives higher priority to the I/O bound process.



Traditional gang scheduling (left) and paired gang scheduling (right)

NS = node scheduler; P = processor.

Figure 3.1. Paired gang scheduling. (From [Wiseman03])

3.3. Communication /network Impact on Scheduling

For message-passing parallel jobs, another factor that will affect the performance is the contention of the network and the overhead of communication. Thus, a scheduler should also take this factor into consideration to achieve both good resource utilization and system performance. There is one good example [Petrini99] for the general behavior of a parallel job consuming network bandwidth, as shown in Figure 3.2 [Petrini99]. This also shows why a scheduler considering the network utilization is desired. It is readily apparent that by overlapping network request, i.e. communication, with computation, we can achieve good utilization for both CPU and network. However, how to implement this strategy while maintaining process coordination in one parallel job can be challenging.

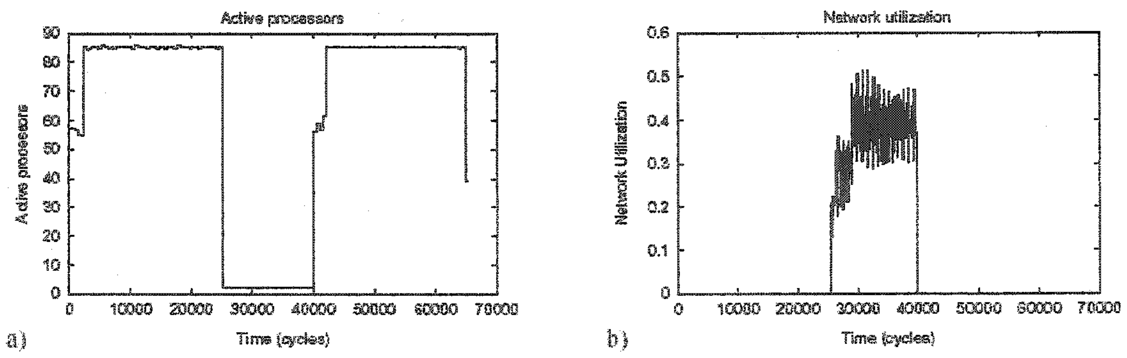
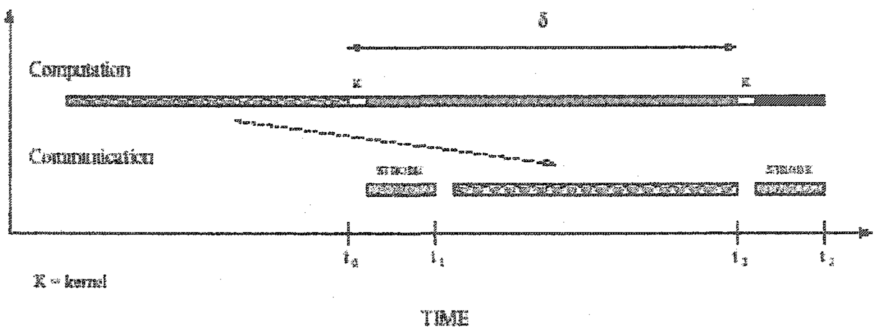


Figure 3.2. Network utilization. (From [Petrini99])

Buffered coscheduling [Petrini99][Petrini00] uses communication buffering and strobing to achieve communication and computation overlap while maintaining the coscheduling of coordinated processes. Communication buffering is intended to buffer the messages for each process and make the communication in the next time slice; thereby, reducing the overhead of system calls by generating a set of system calls for communication at one time instead of making individual system calls several times. Figure 3.3. [Petrini00] shows how the computation and communication is scheduled using buffered coscheduling.



Communication accumulated in the time-slice up to t_0 is downloaded into the network between t_1 and t_2 (after the heartbeat). $\delta \equiv$ length of a time-slice$=t_2-t_0$.

Figure 3.3. Buffered coscheduling. (From [Petrini00])

As mentioned above, one overhead caused by communication is system calls. To alleviate this overhead, user-level communication schemes are proposed. However, using user-level communication will create another problem for gang scheduling in that for every context switch the status of the network interface needs to be saved and restored [Hori98]. In [Hori98], the authors analyzed the impact of this overhead on gang scheduling and implemented a low overhead network preemption strategy.

In summary, where I/O and communication are concerned, trying to overlap I/O or communication with computation is always desirable in order to achieve better response time and system resource utilization.

4. Hyper-threading

Hyper-Threading (HT) technology, developed by Intel, is a form of simultaneous multi-threading technology (SMT) where multiple threads of software applications can be run simultaneously on one processor. This is achieved by duplicating the architectural state on each processor, and giving each logical processor its own sets of registers, while sharing one set of processor execution resources between them [Nakajima02]. Figure 4.1 [Nakajima02] shows the architecture of each HT processor package, i.e. physical processor.

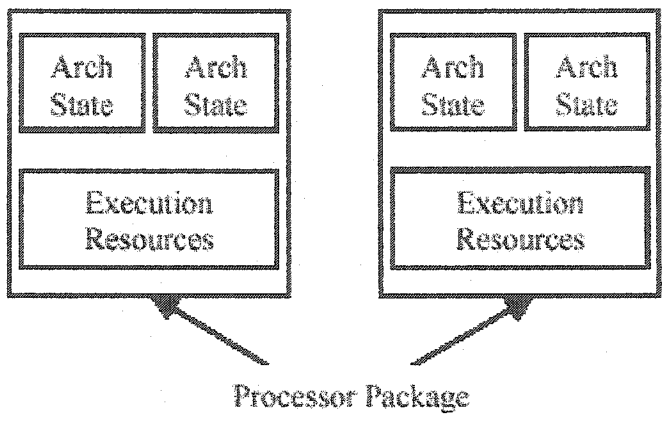


Figure 4.1. The architecture of HT processors. (from [Nakajima02])

HT can improve resource utilization by having two processes running on one physical CPU; hence, it improves the system throughput. However, HT achieves high processor throughput at the expense of performance of a single process. Since the hardware resources are actually shared between two processes, the resource contention will harm the performance of each process. So the real benefit we can get from HT will depend on the resource-consuming characteristics of processes and how two processes running on the same physical CPU compete for hardware resources such as cache and execution units.

Previous research [Magro02][Leng02] shows that the performance of HT varies across different application types. It is found that scientific applications usually get less benefit from HT compared to business applications. This is because usually, scientific applications more often compete for the same computation resources such as floating-point execution units. In [Leng02], it is suggested that the performance degradation can be up to 50% for cache friendly jobs (which have more cache locality) on HT processors due to cache conflict. Also for communication intensive jobs, HT will not provide any gain and will actually decrease the performance.

To enhance the performance of HT, a sophisticated micro-architecture scheduler is needed [Nakajima02]. Symbiotic scheduling [Tullsen00][Snavely02] is developed to support SMT processors and is aimed at enhancing job performance on SMT while improving the processing resource utilization. It monitors the execution resource conflict from different job coscheduling, and coschedules the jobs that have the least resource contention. MASA presented in [Nakajima02] has the same goal as symbiotic scheduling while using a different approach which does not consider the job coscheduling in one physical CPU. When it detects resource contention, MASA tries to balance the loads among different physical CPUs. It is worth noting that MASA is not targeted for uni-processor systems.

HT, by its nature, provides a new possibility for job coscheduling without a context switch and cost for the coordination among processes of one job. However, as discussed above, two jobs coscheduled on HT should be chosen carefully. Not just any random two jobs coscheduled can benefit from HT. In addition to the execution resources shared in CPU, other resources like memory, I/O and network are also shared between two coscheduled jobs. A scheduler should take all resource requirements of a job into consideration for job coscheduling decisions, and this motivates our LOMARC approach.

5. LOMARC – Lookahead Matchmaking for Multi-Resource Coscheduling

LOMARC is a space-time sharing approach that exploits HT (Hyper-Threading) technology to coschedule two jobs to reduce response time and maximize resource utilization. On a HT processor, two jobs can be coscheduled without context switch overhead. Also, since both jobs are actually running at the same time, there is no cost for coscheduling the process working set of each job. When making job pairs to be coscheduled, LOMARC takes multiple resource requirements, including CPU, memory, I/O and network, of a job into consideration.

LOMARC can also be generalized for applications on normal CPUs, i.e. without HT, by changing the matching scheme. When two jobs are coscheduled on a CPU without HT, the processor is actually switched between these two jobs using time slices according to the policy of the local scheduler. In LOMARC, we assume that the local scheduler for each node is a standard time sharing scheduler as in Unix/Linux.

5.1. Goals and Solutions

The design of LOMARC aims to achieve the following goals:

- Considering multiple resources

  As we discussed in Chapter 3, resource contentions can have a significant impact on the performance of the whole system and individual jobs. To maximize the advantage of HT, two coscheduled jobs should have little interference with each other, which means little resource contention between two coscheduled jobs. Usually, parallel jobs can be classified into three different types according to their resource requirement characteristics, namely CPU intensive, I/O intensive and communication intensive. In addition, the memory requirement of a job also has a notable impact on how well it can be coscheduled with other jobs. Our approach will make a scheduling decision according to different resource consuming characteristics of different applications.

- Exploiting coscheduling on HT and supporting time-sharing on standard CPUs
  HT technology provides the possibility of coscheduling two jobs without context switch and the cost for coordinating processes in one job. However, as discussed in Chapter 4, not just any random two jobs coscheduled together can get benefit from this new technology due to fact that the contentions may be encountered on other resources such as memory and I/O. The goal of exploiting HT is actually how to coschedule two jobs to maximize the benefit from it. LOMARC can also be generalized to support job coscheduling using time sharing in standard CPUs. There is no latency hiding in pure space sharing unless the application itself handles this issue, because each processor is dedicated to one process. The goal of LOMARC for standard CPUs is to coschedule two jobs on the same subset of processors to achieve latency hiding while not harming the performance of each job.

- Increasing utilization while improving response time
  Reducing response time and increasing utilization are two major goals for a job scheduler, but sometimes there are trade-offs between these two goals in that maximized utilization does not always lead to minimized response time for each job. LOMARC aims to achieve both goals when it makes scheduling decisions. More precisely, LOMARC estimates the impact on average response time of waiting jobs when attempting to coschedule two jobs that can increase system utilization, and will choose the schedule that can have the best benefit considering both response time and utilization.

To achieve the above goals, we can provide the following solutions:

- Matching two applications that complement each other in all resource requirements to coschedule for improved utilization.
- Estimating both response time impact and utilization gain while reordering the waiting queue for matching jobs to coschedule.

- Providing a clear matching scheme based on application characteristics in resource requirement and a cost model to estimate slowdowns from job coscheduling.
- Exploiting other standard job scheduling techniques such as priority, aging system and backfilling.
- Classifying applications in different resource consuming types such as CPU intensive, I/O intensive and communication intensive.

5.2. LOMARC Scheduling Algorithm

LOMARC uses a priori knowledge about application characteristics, including runtime, size and resource consuming type, to guide the scheduling decision. Usually such types of information can be obtained in two ways. The first approach is that the user collects related measurement data via several execution experiments and submits it together with the application. Another approach is to use compile time analysis to generate such information and then provide it to the scheduler as an a priori input. In LOMARC, we assume such information is provided by applications. Specifically, we assume the following information is given by applications:

- Resource Type – CPU intensive, I/O intensive or communication intensive
- Runtime – estimated execution time
- CPU Time – the ratio of CPU time to whole execution time
- I/O Time – the ratio of I/O time to whole execution time
- Communication Time – the ratio of communication time to whole execution time
- Memory Usage – the ratio of memory requirement per process to total available physical memory in one node.
- Cache Locality – the degree of cache locality, i.e. high, medium or low.
  Cache locality means that one process accesses the same set of data regularly when it does computation. If the cache locality of one application is high, keeping more data in cache will increase its performance significantly. In HT, two processors share the cache in one physical CPU, so two coscheduled jobs

compete for cache. If two coscheduled jobs both have high cache locality, their performance will be degraded notably due to less cache space they can actually get.

- Size – number of processors needed

It is important to note that although LOMARC requires all this information and assumes correct estimations, it can tolerate the inaccuracy of such information. For example, CPU time, I/O time, communication time and memory usage estimation accuracy is not critical to LOMARC, as long as the Resource Type information is provided correctly. Also, for Runtime, since we do not provide reservation for any job, the accuracy of estimation only matters for backfilling, and the maximum slowdown factor used in our backfilling implementation can tolerate certain inaccuracies of the estimation. For the applications that run much longer than their estimated runtime, we can still kill the applications and add them at the end of the waiting queue. This policy gives penalty to the users that report shorter estimated runtime, and hence forces them to supply more accurate execution time estimation. This is a standard policy in most schedulers for dealing with the wrong estimation of job runtime.

Before further explanation of the LOMARC algorithm, it is necessary to clarify our definition of the term ***slowdown***. We use the following formula to define ***slowdown*** in our approach.

$$SL_A = T_{A,B}/T_A$$

Slowdown of job A, SL_A, is actually the ratio of the execution time of job A when it is coscheduled with job B, i.e. $T_{A,B}$, to the execution time of job A, i.e. T_A, when it runs on its own.

5.2.1. Algorithm Abstract

LOMARC is an online scheduler that is driven by new job submission and job termination events. For every such event, LOMARC re-computes the schedule and updates machine node status. Figure 5.1. illustrates the abstract steps of LOMARC algorithm.

```
//Step1:  sum up to the current event, the utilization and effective utilization.
//Step2:  update waiting queue according to aging priority policy
          update_priority();
//Step3:  for job termination event, update corresponding processor status
          loop for the number of processors assigned to this departure job
          {  reduce job_numbers allocated to this processor;
             if (job_numbers==0)
                add corresponding processor ID into the empty_processor_list;  }
//Step4:  schedule new jobs from the job waiting queue according to queuing order
//        until reach a job that cannot be scheduled
          schedule_first_job()
//Step5:  reduce fragmentation by EASY backfilling remaining jobs from the waiting job
queue,
          easy_backfilling();
//Step6:  update job execution time.
          execution_time_change();
```

Figure 5.1. LOMARC abstract algorithm

Details of these six steps are explained as follows.

- Step 1: sum up the utilization and effective utilization before the current event.

  We calculate the utilization and effective utilization according to the current node status information, such as whether a node is occupied or not, which job is running on it and how many jobs are assigned to it.

- Step 2: update waiting queue according to aging priority policy.

  The waiting job queue is maintained by priority. First, we classify jobs into three categories according to their runtime estimation, namely short jobs, medium jobs and long jobs. Second, we assign a priority to a job based on its runtime classification, and give short jobs the highest priority and long jobs the lowest priority, i.e. initially, short jobs are assigned a priority of 2, medium jobs, assigned a priority of 1 and long jobs assigned a priority of 0. Jobs with the same priority are queued in their submission order. To give shorter jobs higher priority, we can expect better overall response time [Perkovic00].

  In order to avoid potential starvation for medium and long jobs, we include an aging policy into our algorithm. Aging means that after a job has waited for a certain amount of time, T_{age}, in the waiting queue, its priority will be promoted to a higher level. Choosing T_{age} is critical for how well an aging scheme works, and there is a trade-off between fairness and average response time [Talby99].

In our implementation of aging, we use average waiting time as T_{age}. When a job has been waiting in the waiting queue longer than this T_{age}, its priority will be boosted to one higher level, i.e. priority increased by five. After another T_{age}, if the job is still waiting, its priority will be promoted again. So for a long job, it will take twice of the average waiting time for it to have the same priority as a short job.

- Step 3: for job termination events: update corresponding processor status

  The status of a processor contains two kinds of information. One is the IDs of jobs assigned to it, and the other is how many jobs are assigned to it. In LOMARC, the maximum number of jobs coscheduled is two, which means the highest number of jobs assigned to a processor is two. When a job terminates, we first reduce the number of jobs on all processors assigned to it; then we check if any processor has zero number of jobs; and then we add those processors to the empty processor list. This implementation makes it possible to coschedule two jobs having different runtimes and sizes. The job with longer runtime in a job pair can still be coscheduled with another job later.

- Step 4: schedule new jobs from the job waiting queue according to queuing order until a job is reached that cannot be scheduled

  This step is the main part of the algorithm, and the pseudo code is presented in Figure 5.2. In this step, we use different strategies for short jobs and medium or long jobs.

  We take the first job from the waiting queue and try to assign available processors to it. If the job is a short job, only pure space sharing is applied, i.e. we need to check only whether there are enough free processors for it. There are two reasons that we do not consider matching for short jobs. First, usually for short jobs, we expect it to finish as soon as possible, so allocating resource to it exclusively is a better choice. Second, due to the short execution time, we do not expect much resource utilization benefit from the job coscheduling.

```
while (! waiting_queue.is_empty ()) {                     // loop over the waiting queue as long as
                                                           the
   current_job = waiting_queue.first;                     // first job can be scheduled
   while (current_job.size <= freenodes.size) {           // enough free nodes for job
   if (current_job.is_medium_or_long_job () )             // try find a match for the job among
     match = find_match (current_job);                    // remaining jobs in waiting queue
   allocate_nodes (current_job);                          // allocate nodes to the current job.
    if (match != null)
      coallocate_nodes (current_job, match);              // coallocate match on same nodes
   current_job = waiting_queue.first;
   }
   if (current_job.is_medium_or_long_job) {              // current job won't fit on free nodes
     match = find_match_among_running (current_job)      // find best match among running jobs
     if (match != null)
       coallocate_nodes (match, current_job);            // allocate current job on same nodes

     else                                                // current job does not match any job
       break;                                            // current job cannot be scheduled now
   }
   else                                                  //short jobs, current job cannot be
                                                          scheduled
     break;
}
```

Figure 5.2. Pseudo code for scheduling jobs

If the job is a medium or long job, there will be two cases for allocating it to available processors. The first case is that there are enough free processors for it. In this case, we search the waiting queue to find the best match job for this first job, and co-allocate the match job and the first job. The other case is that there is not enough free space for the first job. In this case, we search the working job queue and try to match the first job to one current running job, and allocate the first job to the set of processors on which the match job is running.

If the first job can be scheduled, then we remove this job from the waiting queue (the previous second job becomes the first job in current waiting queue) and add it to the job working-queue. We then loop over the above procedures until we cannot schedule the current first job of the waiting queue.

Finding a match is the core of the LOMARC algorithm. The issues addressed by LOMARC include which two jobs can be coscheduled and how to choose the best one among potential matching candidates. We will discuss these two issues in more detail in the following sections.

- Step 5. reduce fragmentation by EASY backfilling remaining jobs from the waiting job queue.

  We choose the EASY backfilling, as introduced in Chapter 2, because it has a better time complexity than the conservative backfilling. If a job is suitable for backfilling, we also try to find a match job from the jobs behind this backfilling job in the waiting queue.

- Step 6: update job execution times

  After allocating processors to new jobs, we update the execution times of scheduled jobs. The actual execution time of a job depends on whether it is coscheduled with any other job and with which job it is coscheduled. For calculating execution time, we examine the slowdown resulting from job coscheduling. We use these actual execution times for determining job termination events.

5.2.2. Matching Scheme

How to choose two jobs to coschedule is essential for exploiting the benefits provided by HT CPUs and achieving latency hiding in standard CPUs. LOMARC considers all job resource requirements, including CPU, memory, I/O and network, when it matches two jobs to coschedule. There are three steps in making a matching decision.

- Step 1. Checking memory usages of two jobs.

  LOMARC uses memory usages of two jobs as a constraint when it decides whether two jobs are matchable, i.e. can be coscheduled. We only coschedule two jobs if the sum of their memory consumption is no more than the total available physical memory. This means that the data of two coscheduled jobs can be loaded in the main memory at the same time and hence can prevent paging, which will seriously degrade the performance of parallel jobs as discussed in Chapter 3.

- Step 2. If two jobs can meet the memory constraint in LOMARC, we further consider their resource types.

  With the idea of coscheduling two jobs that complement each other in resource consumption to reduce resource contention, it is intuitive to coschedule two jobs with different resource consumption types, for example, to coschedule a CPU intensive job and an I/O intensive job.

  In HT CPUs, coscheduling two CPU intensive jobs can still be beneficial as reported in [Magro02]. In LOMARC, we consider that two CPU intensive jobs are matchable, but we also consider the possibility of cache conflict in the slowdown calculation. The final decision of whether to coschedule two jobs together will still depend on the resulting slowdown.

  For a standard CPU, we only coschedule CPU intensive jobs with I/O intensive jobs. Unlike in HT CPUs, job coscheduling on standard CPUs is based on time sharing, which means each processor switches between two processes independently of other processors. Although coscheduling CPU intensive and communication intensive jobs can also achieve latency hiding, communication intensive jobs require process coscheduling within their own process working sets and hence, cannot tolerate the frequent independent context switch on each processor.

  In the standard CPU environment, LOMARC does not control the local scheduling between two coscheduled jobs. We assume that the local scheduler is a time-sharing scheduler that gives higher priority to I/O bound process. This strategy is actually used by most time-sharing operating systems such as Unix, Linux and Windows NT. By having higher priority, an I/O bound process can preempt a CPU bound process when it is ready to run, that is, its I/O request has been served. Since the I/O bound process will block for I/O again soon, it will relinquish the CPU to the CPU bound process. Hence, when the I/O bound process is doing I/O, the CPU bound process can do computation. So this can achieve I/O latency hiding, and keeps both CPU and I/O devices busy to increase resource utilization.

To summarize, regarding job resource types, LOMARC coschedules:

- CPU and CPU, CPU and I/O, CPU and communication intensive jobs on HT CPUs,
- CPU and I/O intensive jobs on standard CPUs.

- Step 3. If two jobs match in resource type, we further calculate the slowdown from job coscheduling.

  We check whether the slowdown from coscheduling two jobs is less or equal to a certain maximum slowdown limit —MAX_SLOWDOWN. Only two jobs with a coscheduling slowdown no more than this limit will be coscheduled.

  The detailed slowdown estimation will be discussed in Chapter 6, because it is independent from the algorithm, yet relevant to our comparison experiments.

5.2.3. Utilization and Response Time Impact

In LOMARC, before scheduling a medium or long job, we search the waiting job queue to find the best match job for it to coschedule. For each job in the waiting queue, we first check whether it can be matched to the current job to be scheduled under the LOMARC matching scheme; if the answer is yes, this job becomes a match candidate for the current job to be scheduled. When choosing the best match among all match candidates, we have two questions for each match candidate: 1. How much utilization gain can we get from matching this candidate to the current job? 2. What is the response time impact on the other waiting jobs?

To answer the first question, LOMARC calculates the utilization gain for each job pair based on their sizes and the slowdown factors from job coscheduling. Figure 5.3 can help to visualize our utilization gain calculation. In this figure, Job J1 and J2 are coscheduled, while J1 has a larger size and J2 has a longer runtime. The areas circled by dashed lines are occupied by processes from one job, i.e. either J1 or J2, while the other areas are allocated to both J1 and J2.

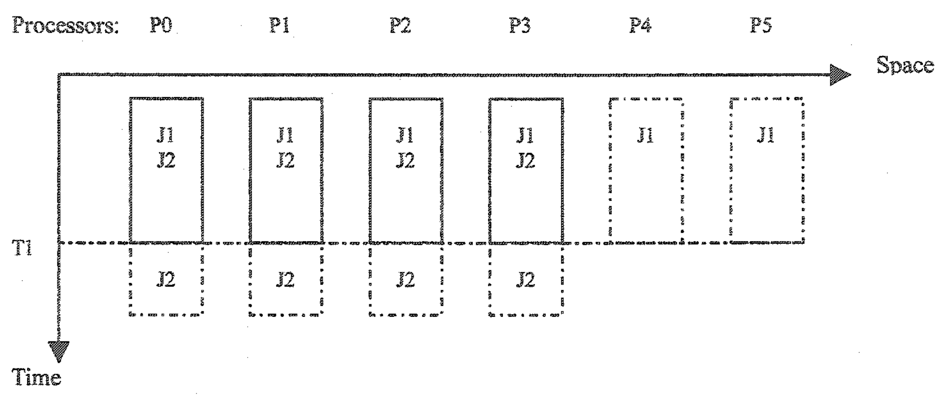


Figure 5.3. An example of job coscheduling

When calculating the utilization gain, we only consider the space-time area before T1, when J1 will terminate, because after T1, it is possible to have another job coscheduled with J2 for the rest of its execution time. We use the following formula to calculate the utilization gain from coscheduling jobs J1 and J2.

$$U_{gain}= (\min(S_{J1}, S_{J2})^* (2/SL_{J1,J2}-1) - |S_{J1}-S_{J2}| ^*(1-1/SL_{J1,J2}))/\max(S_{J1}, S_{J2})$$

In this formula, S_{J1} and S_{J2} refer to the sizes of J1 and J2, respectively. $SL_{J1,J2}$ is the slowdown factor of coscheduling jobs J1 and J2. When a job is running on its own, the utilization of processors allocated to it is 1 (or 100%). The utilization efficiency of the processor that has two jobs running on it is calculated as 2/ $SL_{J1,J2}$. With the slowdown factor being larger, the utilization efficiency is decreased. For example, in an ideal case, if the slowdown factor from coscheduling two jobs is 1, which means their runtime will not increase from job coscheduling, the effective utilization of a processor allocated to both of these two jobs would be 2, which means the same as two processors. As we can see, when the slowdown factor is less than 2, the effective utilization will be greater than 1, which means a utilization gain from job coscheduling. (2/ $SL_{J1,J2}$ -1) represents this increase in utilization. $\min(S_{J1}, S_{J2})^*$ $(2/SL_{J1,J2}-1)$ is the total utilization increase for the processors allocated to both of the two jobs, e.g. for processors P0 to P3 in Figure 5.3. For the processes of J1 running on processors P4 and P5 in figure 5.3, the same slowdown factor applies, because they will have the same runtime as the other processes of J1. There is actually a decrease in utilization for P4 and P5, because only one process is running on each of them, and the runtime of the process is longer than it would be by running on its own. $|S_{J1}-S_{J2}| *(1-1/SL_{J1,J2})$ calculates this utilization decrease. The total gain is the utilization increase minus the utilization decrease. Finally, we divide this total gain by the size of the larger job, and this gives an average utilization gain for each processor allocated to this coscheduled job pair. This calculation implies that we will have better utilization gain from coscheduling two jobs if the two jobs have less difference in sizes and lower slowdown factors.

The second question for choosing a match is how the pairing will impact the response time of other jobs in the waiting queue. Figure 5.4 shows response time impacts for all jobs in the waiting queue. There will be two different impacts respectively for the jobs before the matching job in the queue and the jobs behind the job in the queue.

- Jobs in front of the matching job: ***push-down*** jobs

  Delays are caused for these jobs due to the slowdown from job coscheduling and the runtime of the matching job, if it is longer than the current job to be scheduled. For them, we calculate an estimate of the impact by the sum of all relative delays.

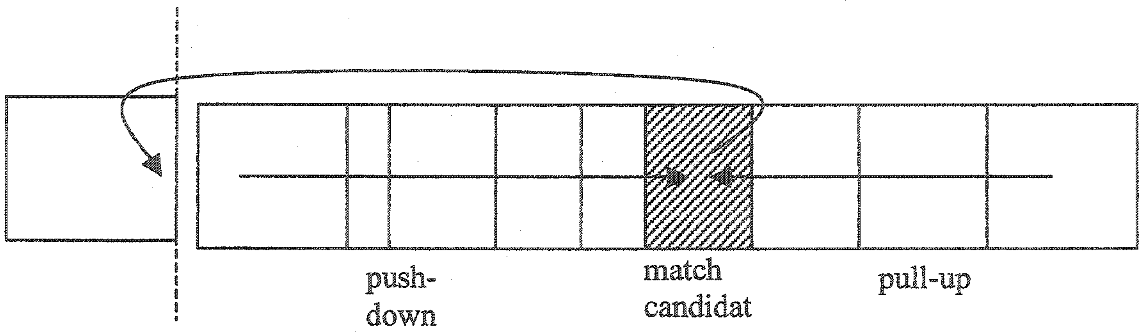


Figure 5.4. Response time impacts

- Jobs behind the matching job: ***pull-up*** jobs

  Response time improvements are expected for these jobs, because the joint runtime of two jobs that are matchable under the LOMARC matching scheme is assumed to be shorter than the sum of runtimes of the two jobs. For these jobs, we calculate an estimate of the impact by the sum of all relative improvements.

In addition, we also predict the impact on future arrival jobs within this job pairing runtime period. To predict the future job arrivals, we use the parameters in the workload modeling to calculate the average number of arrival jobs in this time duration and the average job work size, which is the product of job runtime and job size. When calculating the response time for future arrival jobs, different from the jobs that are already in the waiting queue, we only consider their own execution times, because future arrival jobs have not been waiting for other jobs at the time of calculation. The response time improvement is also expected for future arrival jobs with the same reason as for the pull up jobs.

When calculating relative response delay or improvement, we do not make any specific future schedule plans for waiting jobs. We base our heuristic calculation on job runtime and size, with the idea that in a perfect packing situation, only job runtime and size will have an impact on the response time, i.e. we sum up the product of job runtime and size, then divide it by the total number of nodes and use this value as the estimate of response time. This calculation is reasonable because the exact packing of all jobs will change dynamically when there are new jobs submitted with different priorities. After calculating the total average delays and total average improvement, we use the improvement value minus the delay value for the final response impact value. If this value is positive, it means that we have an overall response time improvement by matching this job. If the value is negative, the overall impact on response is a delay. Figure 5.5 shows the pseudo code of response time impact calculation.

```
// calculates overall average relative  response-time impact, in increase/decrease relative to
normal //response time
calculate_response_time () {

        //sum up estimated relative delay for push-down jobs
        for (all push_down_jobs (jobn)) {
              response_time += jobn.runtime * jobn.size / n_nodes;
              response_increase += delay / response_time;
             }
        response_increase/=number_of_push_down_jobs;
        // sum up estimated improvement for pull-up jobs
        for (all_pull_up_jobs(jobn)){
              response_time += jobn.runtime * jobn.size / n_nodes;
              response_decrease += improvement / response_time;
             }

        //estimate response time impact on future arrival jobs within the current runtime
        duration
        for (future_arrival_ jobs(jobn)){
             response_time  = jobn.runtime * jobn.size / n_nodes;
             response_decrease+= improvement / response_time;
        }
        response_decrease/=(number_of_pull_up_jobs + number_of_future_arrival_jobs);
        total_response_impact= response_decrease – response_increase;
}
```

Figure 5.5. Pseudo code for calculating response impact.

In the calculation, the delay is the runtime increase from job coscheduling plus the runtime difference between two jobs if the match job has longer runtime than the current job to be scheduled. The improvement is the difference between the runtime of the match job and the delay.

Having the knowledge of utilization gain and response time impact, we use a weight value to combine these two factors to calculate the overall benefit shown as the following formula.

benefit = (1-WEIGHT)*response_impact + WEIGHT*utilization_gain

According to our calculation, the values of response_impact and utilization_gain will fall into a similar range, which is (-1,1), and this makes it possible to combine these two values together using a weight value. The value of WEIGHT can be varied in the range of [0,1]. As shown in the above formula, when WEIGHT is 1, only utilization gain counts for the benefit. With the WEIGHT value of 0, we focus on the response time impact from the matching. Changing WEIGHT value can tune the algorithm for the trade-off between different goals, i.e. best response time vs. maximum utilization.

Finally, we choose the job with the highest benefit value as the match for the current job.

5.2.4. Overhead Analysis

To evaluate the overhead of the LOMARC approach, we will analyze the time complexity of the LOMARC algorithm and space needed by the algorithm.

The problem size of our approach includes two values: n as the total number of jobs and k as the machine size, i.e. the total number of processors, where $n>>k$. The LOMARC algorithm consists of six steps as described in Section 5.2.1. Table 5.1 shows the time complexity of each step in the worst case.

Steps	Time Complexity
Step 1	$O(k)$
Step 2	$O(nlgn)$
Step 3	$O(k)$
Step 4	$O(n^2)$
Step 5	$O(n)$
Step 6	$O(n)$

Table 5.1. Time complexity analysis.

The explanation of the time complexity for each step is as follows.

- In step1, we calculate utilization and effective utilization according to the status of each processor. So it takes $O(k)$ time to compute the total utilization for k processors.
- In step 2, for each job in the waiting queue, it takes constant time to update the priority and $O(lgn)$ time to insert the job in the proper position in the waiting

queue according to its updated priority. So this gives an overall O(*nlgn*) time for updating the priority waiting queue.

- In step 3, it takes constant time to update the status for each processor, and hence gives an overall O(*k*) time to update the statuses of all processors.

- In step 4, for each job it takes O(*n*) time to traverse the waiting queue to find the best match and O(*k*) time to allocate processors to the job. So this gives an $O(n^2+kn)$ time for allocating jobs from the waiting queue in the worst case. Since $n>>k$, we claim the time complexity of this step is $O(n^2)$.

- In step 5, we use EASY backfilling which takes O(*n*) time to traverse the waiting queue to find the jobs that can be used for backfilling.

- In step 6, we re-compute the execution time of each job in the working queue, and this gives O(*n*) time complexity for this step.

The overall time complexity of the LOMARC algorithm is the sum of the time complexities of these six steps, and it gives us a result as $O(n^2)$.

The space used by LOMARC is $O(n+k)$. In LOMARC, we maintain one working job queue and one waiting job queue, the total length of which are *n* in the worst case. An array of node status and one empty node list are used to keep node information and process allocation respectively.

6. Simulation and Experiment

We implement LOMARC using event-based simulation, as the scheduler is driven by job arrival and termination events. In our simulation, we model a system with 128 single-CPU nodes with 512MB memory per node.

6.1. Slowdown Estimation

To evaluate the performance of our approach, we model the slowdown factor for two coscheduled jobs. For the comparison test purposes, we simulate slowdown for all possible job coscheduling situations, i.e. including the cases in which LOMARC will not coschedule two jobs together.

In slowdown calculations, we first check whether two jobs have memory contention. If memory contention exists, i.e. $(f_{mem,A} + f_{mem,B}) > 1$, we use the following formula to calculate the memory slowdown resulting from coscheduling job A and job B,

$$((f_{mem,A} + f_{mem,B}) - 1) * 2$$

where f_{mem} is the fraction of the memory size needed for each job. $((f_{mem,A} + f_{mem,B}) - 1)$ is the portion of job data sets that cannot fit into physical memory. The factor of 2 represents the slowdown from demand paging according to the experimental results presented in [Burger96], and this value is optimal.

If two coscheduled jobs do not satisfy the requirement of Step 2 in the LOMARC matching scheme, i.e. their resource consuming types are not complementary, we assume a slowdown of 2, which is optimum for this case. Otherwise, we use the following calculations which consider cache conflict possibilities on HT processors.

- If two jobs have cache conflict:

  $SL_{AB} = 1+(3-1)*min(f_{cpu,A}, f_{cpu,B}) + (2-1)*min(f_{io,A}, f_{io,B}) + (2-1) *min(f_{comm,A}, f_{comm,B})$

- If two jobs have no cache conflict:

  $SL_{AB}= 1+(1.4-1)*min(f_{cpu,A}, f_{cpu,B})+(2-1)*min(f_{io,A}, f_{io,B})+(2-1) *min(f_{comm,A}, f_{comm,B})$

For the two jobs coscheduled on standard CPUs, we use following formula to calculate the slowdown.

$SL_{AB}= 1+(2-1)*min(f_{cpu,A}, f_{cpu,B}) + (2-1)*min(f_{io,A}, f_{io,B}) + (2-1) *min(f_{comm,A}, f_{comm,B})$

In the above calculations, SL_{AB} is the slowdown from coscheduling job A and B, while f_{cpu}, f_{io} and f_{comm}, are the fraction of CPU, I/O and communication times respectively for each job. We assume the worst case in our calculation, i.e. two jobs perform computing, I/O and communication at the same time. As reported in [Leng2002], the slowdown from coscheduling two jobs on HT processors can be up to 3 due to cache conflict. So we use factor of 3 as the slowdown impact from CPU sharing when there is cache conflict between two jobs. If there is no cache conflict between two jobs, the slowdown from computation is much less. Based on experimental results reported in [Magro2002], we assume the slowdown from CPU sharing is 1.4 on HT processors, when there is no cache conflict. For I/O and communication sharing, we assume a slowdown factor of 2 for sequential execution. For job coscheduling on standard CPUs, we use a slowdown factor of 2 to represent the sequential execution in each execution components. We use the slowdown factor of each execution component minus 1 in the calculation to represent the execution time increase from the slowdown. Finally, we calculate the overall slowdown by summing up all the execution increases and adding them to 1, which presents the original runtime.

At last, we add the memory slowdown, if applicable, to the above slowdown factor to get the final slowdown from coscheduling.

Note that a slowdown of 2 corresponds to time-sharing on a standard CPU and that any slowdown larger than 2 means a decrease in utilization efficiency. In LOMARC, we set a slowdown limit – MAX_SLOWDOWN for the final coscheduling decision, and the maximum value of this limit can be 2.

6.2. Workload Modeling

To evaluate a scheduling strategy design, a realistic workload model is important for providing convincing experimental results. In our simulation we use a workload model presented in [Lublin01] to model job sizes, job runtimes and job arrival times. This model is based on the analysis of workload logs from three different locations: the 416-node Intel Paragon machine installed at San-Diego Supercomputer Center(SDSC), the 1024-node connection Machine CM-5 in Los-Alamos National Lab (LANL) and the 100-node IBM SP2 machine in the Swedish Royal Institute of Technology in Stockholm (KTH). The model is created to represent the common characteristics of these real workloads. For other characteristics of a job such as resource consuming types, we model them based on our assumption, because there is no statistic model available regarding this kind of information for real workloads.

6.2.1. Job size modeling

For job size modeling, we only consider rigid jobs, the sizes of which do not change during their execution, because our approach does not consider job size adaptation. The model classifies jobs into three categories according to their size: serial jobs with the size of one; power-of-two jobs where the sizes are numbers that are the power of two; and the rest. This classification reflects the notable fraction of serial jobs and power-of-two jobs in real workloads.

The model applies a logarithmic transformation to the data, because job sizes span a large range. A two-phase-uniform distribution is used to generate the logarithmic sizes, which are the logarithms of job sizes, with the base of two. Two-phase-distribution is a generalization of the uniform distribution, and it consists of two uniform distributions in two different ranges. The parameter *l* (low) and *m* (medium) define the first range, while *m* (medium) and *h* (high) define the second range. The parameter p (proportion) defines the probability of a number falling into the first range. Figure 6.1 shows the cumulative distribution function (CDF) of a two-phase-uniform distribution.

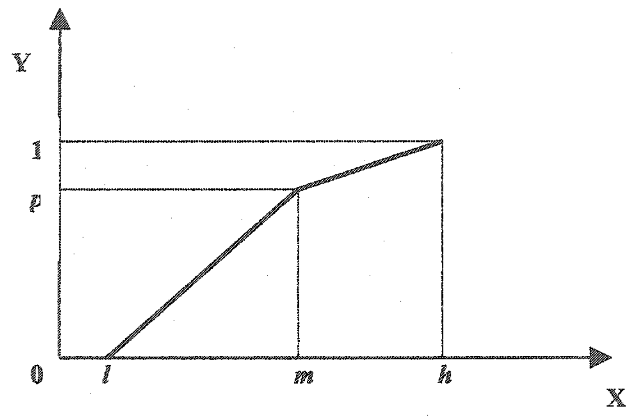


Figure 6.1. CDF of two-phase-uniform distribution.

Figure 6.2 [Lublin01] shows the algorithm for modeling the size of a job. To decide the size of a job, first we use p_1 to decide whether it is a serial job or not. For a serial job, the size is one. If it is a parallel job, we use the two-phase-distribution to choose the logarithm size. After having a logarithmic size, we use p2 to decide whether it is a power-of-two job. For a power-of-two job, we round the logarithm size to an integer. At last, we use this logarithm size to compute the job size.

Table 6.1 shows the parameters used in the modeling. Where p_1 is the probability of serial jobs, and p_2 is the probability of power-of-two jobs within parallel jobs. The other four parameters are used in the two-phase-uniform distribution to decide the size of parallel jobs.

p_1	p_2	l	m	h	p
0.24	0.75	0.8	4.5	7	0.86

Table 6.1. Parameters for job size modeling.

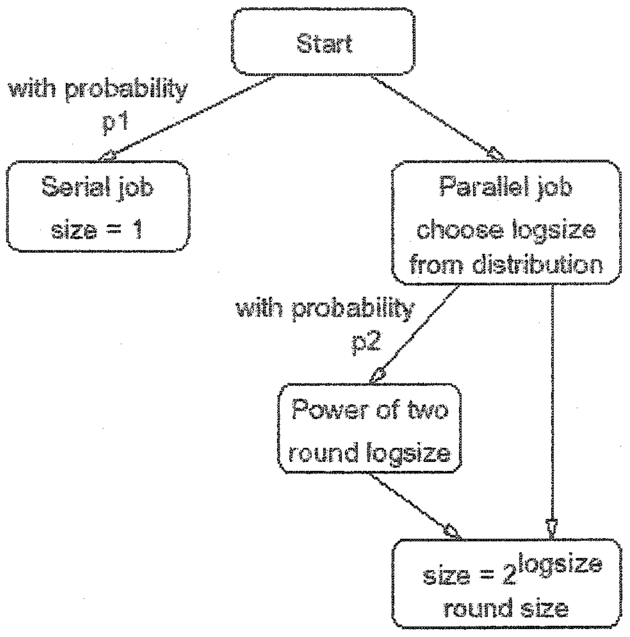

Start
with probability p1
Serial job size = 1
Parallel job choose logsize from distribution
with probability p2
Power of two round logsize
size = 2^logsize round size

Figure 6.2. Algorithm for modeling the size of a job. (from[Lublin01])

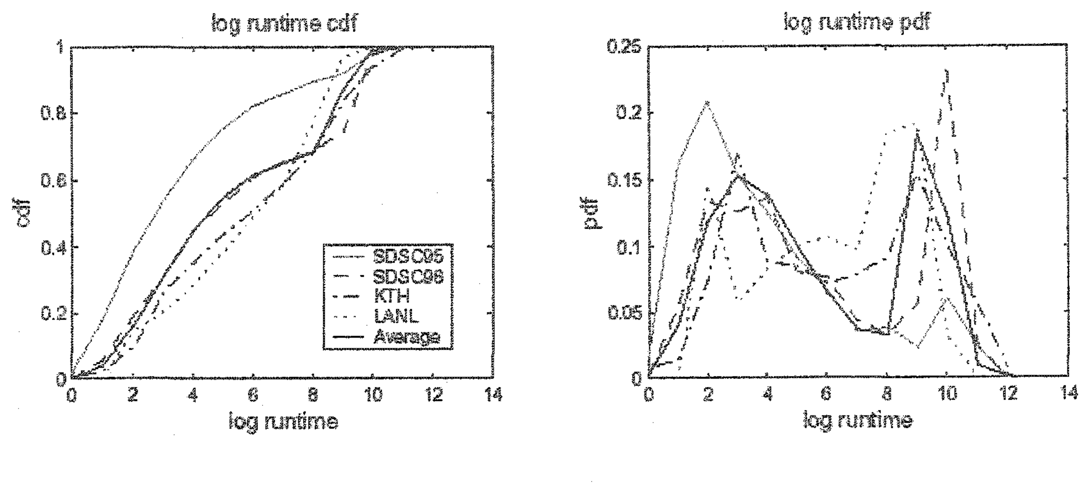
In the model, *l* is 0.8, and this gives the minimum size of a parallel job as 2. The maximum size of a job is 128, calculated as 2 to the power of 7, which is the *h* value in the model. It is important to note that the maximum size of a job is the same as the machine size. The mean value of job sizes, including serial jobs, is calculated as the following formula.

$$Mean = p_1 * 1 + (1-p_1) * (p * (2^l + 2^m)/2 + (1-p) * (2^m + 2^h)/2)$$

Using the values of parameter in the model, this formula gives a mean job size of 56.

6.2.2. Job runtime modeling

In job runtime modeling, the model uses a hyper-gamma distribution to generate the natural logarithm of runtimes. The reason for choosing a hyper-gamma distribution is that it can represent the bimodal curve of real distributions. Figure 6.3. [Lublin01] shows the logarithmic runtime distribution extracted from real workloads of three different sites, and the average model. SDSC95 and SDSC96 represent the workloads from SDSC in 1995 and 1996, respectively. Figure 6.3(a) shows the CDF of the workloads, and Figure 6.3(b) shows the probability density function (PDF) model derived from the CDF of these workloads.



(a) (b)

Figure 6.3. Logarithmic runtime distributions and the derived model (From[Lublin01])

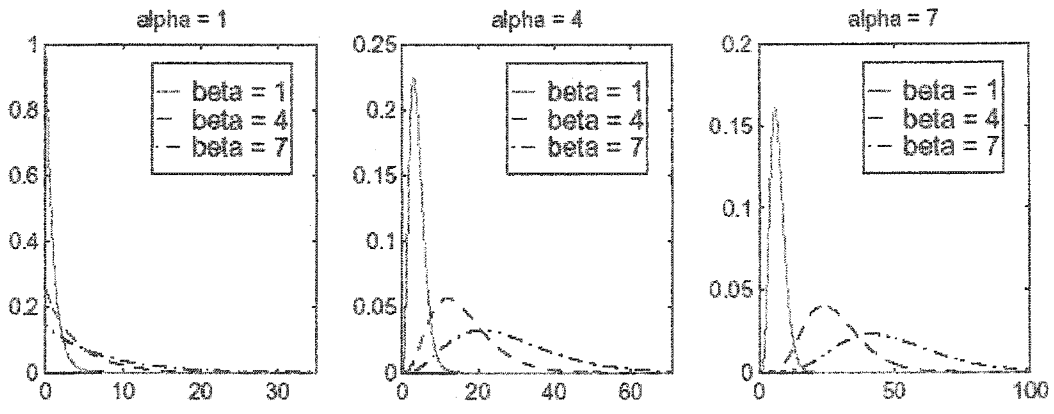
The mathematical definition of a gamma distribution is as follows [Lublin01].

$$f(x;\alpha,\beta)=\frac{1}{\Gamma(\alpha)\beta^{\alpha}}x^{\alpha-1}e^{-\frac{x}{\beta}}$$

where $x,\alpha,\beta>0$ and

$$\Gamma(\alpha)=\int_0^{\infty}t^{\alpha-1}e^{-t}dt$$

α and β are the parameters of the distribution, where α is the shape parameter and β is the scale parameter. The mean value of the distribution is the product of these two parameters: $m=\alpha*\beta$, and the variance is $\alpha*\beta^2$. Figure 6.4 [Lublin01] shows some examples of gamma PDF distributions. We can see how the two parameters influence the distribution from this figure.



(a) (b) (c)

Figure 6.4. Examples of gamma distributions (From [Lublin01])

The Hyper-gamma distribution is composed of two gamma distributions with a parameter *p* as the proportion of the distribution falling into the first gamma distribution. Besides job runtimes, the model also uses a hyper-gamma distribution to model the total work of a job, which is the product of job size and runtime. We use the parameters of job total work modeling for the prediction of future arrival jobs. The parameters used in the model are shown in Table 6.2 [Lublin01].

	α_1	β_1	α_2	β_2	p
Runtime	4.20	0.94	312.0	0.03	0.685
Total work	10.74	0.55	37.96	0.37	0.577

Table 6.2. Parameters for job runtime and total job work modeling. [Lublin01]

There is a correlation between the size of a job and its runtime, and that is a larger job usually tends to have a longer runtime. To represent this correlation, the model uses job sizes to calculate the *p* value with the following formula.

$$p=a*s + b$$

Where *s* is the job size, *a*= -0.0054 and *b*= 0.78. It is worth to note that *a* is a negative number, which means with the job size increasing, the probability of using the first gamma distribution, which has a smaller mean value than the second gamma distribution, will be decreased. For modeling the runtime of a job, we first use its size to compute the *p* value, and then use this value together with the other four parameters as the input of the hyper-gamma distribution. The final runtime is *e* to the power of the number generated from the distribution.

For the whole workload, the mean logarithmic runtime is calculated as follows.

$$mean = pe^{\alpha_1\beta_1} + (1-p)e^{\alpha_2\beta_2}$$

This gives a mean runtime of 3690 seconds. Using the hyper-gamma distribution of runtime modeling with the given parameters, the probability of a random number being larger than 12 converges to zero, so in our implementation of runtime modeling, we set the maximum value of logarithmic runtime as 12, and this gives the maximum runtime of 45 hours. From Figure 6.3, we can see this also represents the real distribution in a workload.

We classify jobs according to their runtime based on the following definition.

- Short job: with the runtime in the range of [1sec, 1min].
- Medium job: with the runtime in the range of (1min, 1hr].
- Long job: with the runtime in the range of (1hr, 45hr].

According to our classification, the model generates around 30% long jobs, 28% medium jobs and the rest are short jobs. This classification is for the purpose of assigning different priorities to jobs. For jobs that have runtimes shorter than 1 minute, it is important to assign the highest priority to them and hence to reduce the overall response time. In addition, for job coscheduling, we do not consider the jobs that have runtime less than 1 minute, because it will not provide meaningful benefit to coschedule these jobs due to their short execution time.

6.2.3. Job arrival time modeling

The workload model that we use can represent both the overall distribution of inter-arrival times and the daily cycle of different densities of job arrivals across hours of a day. During one day, job arrival densities are different in different hours. Usually more jobs arrive during daytime than nighttime. Figure 6.5 [Lublin01] shows one example of job arrival numbers in a daily cycle.

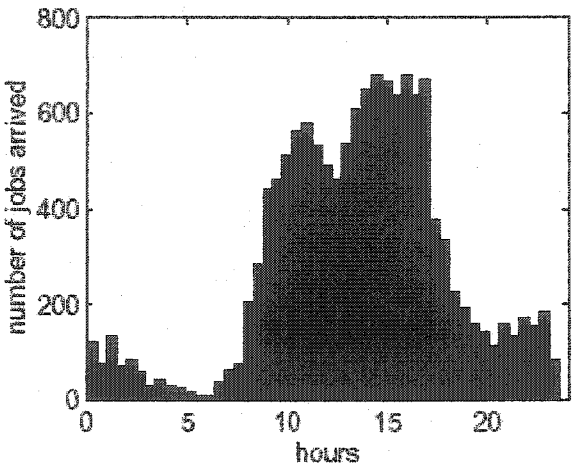


Figure 6.5. Number of job arrivals in a daily cycle (from [Lublin01])

To incorporate the characteristics of both overall inter-arrival time and the job arrival daily cycle, the model first uses one gamma distribution to represent the job inter-arrival time in peak hours, which is between 8AM to 7PM. Based on this distribution, the inter-arrival times are then adjusted according to different weights in different time slots. To calculate the weights in different time slots, the model uses another gamma distribution that represents the daily cycle. At first, we split one day into 48 half-hour time slots. For each slot *t*, the probability of job arrival then can be computed as:

$$w(t) = F(t+0.5) - F(t-0.5)$$

where *F* is the CDF of the gamma distribution model, and *F(t)* represents the probability of a number falling into the range of [0, t]. $F(t+0.5) - F(t-0.5)$ calculates the probability of a number falling into the range of [t-0.5, t+0.5]. It is important to note that *t* is in the range of [10, 57]. As we can see from Figure 6.5, the minimum arrival number appears around 5AM, so the ten time slots before 5AM are shifted by adding 48 to match the gamma distribution model. The weight for each time slot is then calculated as $w(t)/w_{avg}$, where w_{avg} is the average of all 48 $w(t)$ values. Figure 6.6 shows the weights curve generated by the model. The time slots that have higher weight values can be seen as having longer virtual time, so the probability of a job arrival falling into these slots is higher.

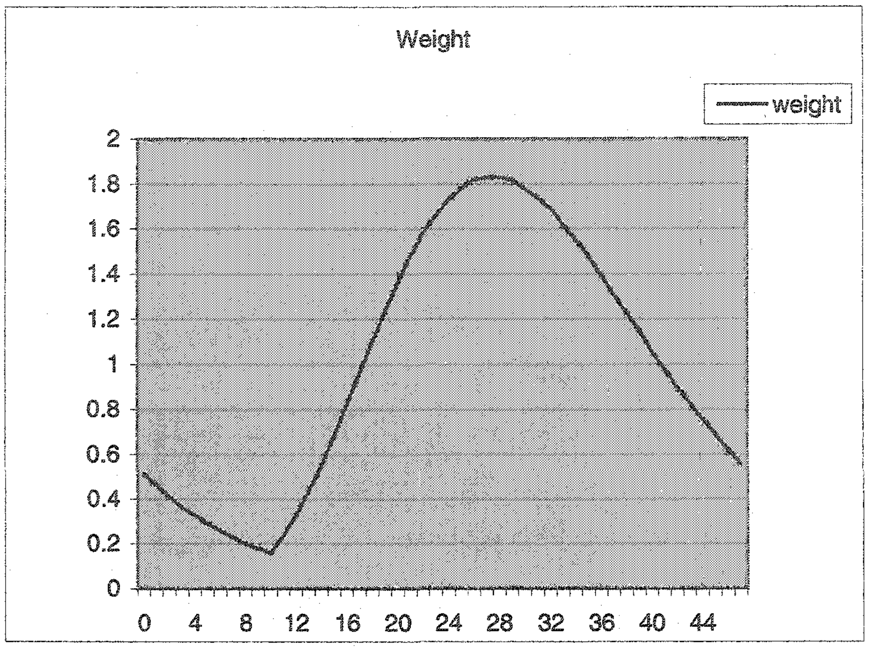


Figure 6.6. Modeled time slot weights in a daily cycle

The parameters of the gamma distributions used for peak hour job inter-arrival time and the daily cycle modeling are shown in Table 6.3 [Lublin01]. By using these two gamma distributions, we can model job arrival times that represent both the overall inter-arrival time distribution and the different job arrival numbers in a daily cycle.

	α	β
Peak hour	10.23	0.49
Daily Cycle	8.17	3.96

Table 6.3. Parameters for job arrival time modeling. [Lublin01]

Given α and β values, we calculate the mean job arrival time as 150 seconds. Having all the information about job size, runtime and inter-arrival time, we can use the following formula to calculate the expected workload.

$$Load = (r*n)/(P*a)$$

where r is the mean runtime, n refers to mean job size, P is the total number of processors and a is the mean job arrival time. A higher Load value means a heavier workload. For the model using given parameters, we have a Load value around 10.6. Table 6.4. shows the summary information of job size, runtime and inter-arrival times for the workload model.

	Job size (number of processors)	Job runtime (seconds)	Job inter-arrival time (seconds)
Mean value	56	3690	150

Table 6.4. Summary information of the workload model.

6.2.4. Job resource consumption characteristics modeling

Job characteristics in resource consumption are important to our approach. To the best of our knowledge, currently there is no statistic model available for representing the job resource consuming characteristics in real workloads, such as what is the fraction range of I/O intensive jobs or communication intensive jobs in a real workload. In our simulation, we model this kind of information based on our own assumptions.

Considering job resource consuming types, we classify jobs into three classes: CPU intensive jobs; I/O intensive jobs; and communication intensive jobs. We model 40% CPU intensive jobs, 30% I/O intensive jobs, and 30% communication intensive jobs of the whole workload, and the arrival sequence of different classes of jobs is randomly generated using a uniform distribution. For each class of jobs, we model their fractions of CPU, I/O and communication times, f_{cpu}, f_{io}, and f_{comm} respectively, as follows:

- CPU intensive jobs:

f_{cpu} in the range of [0.5, 0.8), f_{io} in the range of [0.1, 0.4), with $f_{cpu}+f_{io}$ in the range of [0.6, 0.9). This leaves f_{comm} in the range of [0.1, 0.4).

- I/O intensive jobs:
  f_{io} in the range of [0.5, 0.8), f_{cpu} in the range of [0.1, 0.4), with $f_{io}+f_{cpu}$ in the range of [0.6, 0.9). This leaves f_{comm} in the range of [0.1, 0.4).
- Communication intensive jobs:
  f_{comm} in the range of [0.5, 0.8), f_{cpu} in the range of [0.1, 0.4), with $f_{comm}+f_{cpu}$ in the range of [0.6, 0.9). This leaves f_{io} in the range of [0.1, 0.4).

Within each range, we use a uniform distribution to randomly generate the specific numbers for job characteristics.

For the memory consumption modeling, we randomly generate numbers in the range of [0.05, 1), which represent the ratio of job memory requirement per process to the available physical memory space per node, which is 512MB in our system model. We model 70% of the jobs with memory consumption in the range of [0.05, 0.5], 25% of the jobs with memory consumption in the range of (0.5, 0.8] and 5% of jobs with memory consumption in the range of (0.8,1). Within each range, we use a uniform distribution to generate the random numbers. This model should reasonably represent the memory consumption characteristics in real workloads, because it roughly matches the result from previous studies of job memory consumption as in [Chiang01].

6.3. Experiments and Result Analysis

6.3.1. Overview of the experiments

In our experiments, we used the workload model described in the previous section as input and tested the workload with 5000 jobs. In order to show how a workload can have an impact on the performance, we also adjusted the parameters for job inter-arrival time to generate heavier workloads. We tested the workload with four different job arrival rates.

To prove better performance of our approach in both response time and utilization efficiency, we compared our approach with the following three approaches.

- PSS: Pure Space Sharing with backfilling and priority aging.
- AC: Always Coscheduling two medium or long jobs on HT processors without considering their resource consuming types.
- AM: Adjacent Match, coscheduling two adjacent medium or long jobs if they are matchable under our matching scheme.

For our approach, we tested LOMARC for both HT processors and standard CPUs, i.e. without HT. To see how finding the best match helps to improve the overall performance, we also tested the following variations for the LOMARC with HT processors.

- Varying WEIGHT value. We tested LOMARC with WEIGHT value 1(considering utilization only), 0 (considering response time only) and 0.5(combining utilization and response time).
- A simplified version that only looks-ahead to find the first match for coscheduling, without looking for the best match with the consideration of utilization gain or response time impact on other jobs.

In addition to the overall performance of the entire workload, we also tested individual performance of each job runtime class and job type to see how our approach has different impacts on different job categories.

6.3.2. Performance metrics applied

We applied the following metrics in our experiments to evaluate the performance of our approach and compare it with other approaches.

- Average response time: the average value of response times for all jobs.

- Average bounded relative response time: the average value of bounded relative response time for all jobs. We choose this metrics rather than relative response time to eliminate the extreme impact from very short jobs, and we set the lower bound of runtime as 60 seconds.

- Utilization: the ratio of total nodes occupied time to the product of makespan and number of nodes, i.e. total available processing time.

- Utilization efficiency: defined in Chapter 2. It is worth noting that in a pure space sharing approach, this value will be the same as utilization and will not be larger than 1. In an approach with job coscheduling on HT CPU or using time sharing, utilization efficiency can be larger than 1, and this value compared to utilization can show the benefit we get from job coscheduling.

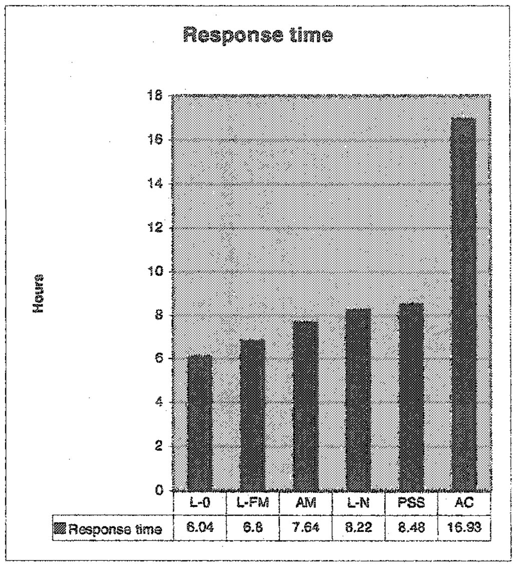
6.3.3. Workload impact

To examine how a workload can have an impact on scheduling approaches, we tested four sets of workloads with different job arrival rates. The parameters for job inter-arrival time modeling and the Load values of generated workloads are listed in Table 6.5. Note that we only changed the α value for varying job inter-arrival times. The Load values are calculated as described in Section 6.2.3. Workload 1 is the same as the model described in Section 6.2.

	Workload 1	Workload 2	Workload 3	Workload 4
α	10.23	9.83	8.83	8.03
Load	10.6	13	21	32

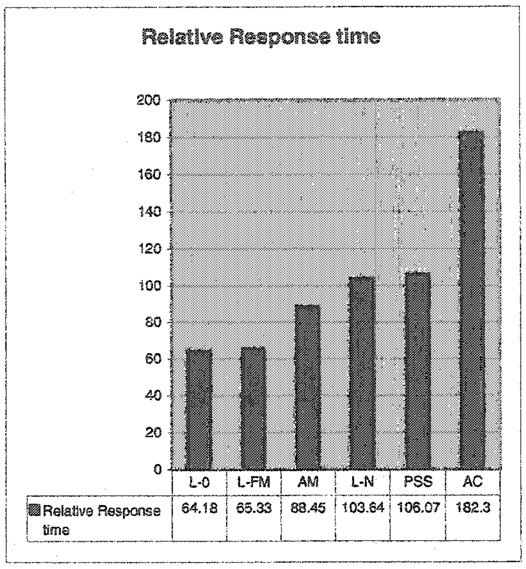
Table 6.5. Workload information.

Figure 6.7 shows the results of LOMARC in comparison to other scheduling approaches involved in our experiment under Workload 1. L-0 stands for LOMARC with the WEIGHT value as 0. L-FM stands for the LOMARC variant that only finds the first match to coschedule. AM, PSS and AC are the other scheduling approaches as explained in the previous section. Figure 6.7(a) shows the average response times (in hours), (b) shows the average relative bounded response time and (c) shows utilization and effective utilization.



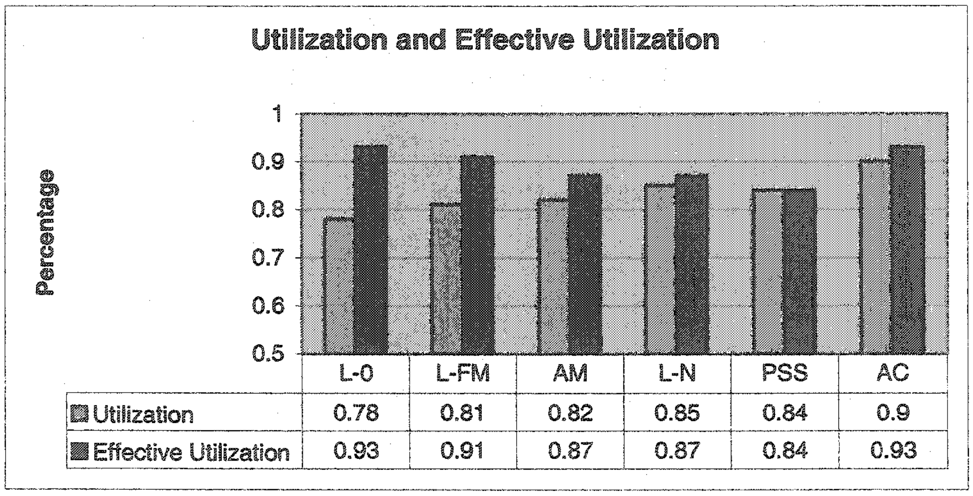
	L-0	L-FM	AM	L-N	PSS	AC
▨ Response time	6.04	6.8	7.64	8.22	8.48	16.93

(a)



	L-0	L-FM	AM	L-N	PSS	AC
▨ Relative Response time	64.18	65.33	88.45	103.64	106.07	182.3

(b)



	L-0	L-FM	AM	L-N	PSS	AC
▨ Utilization	0.78	0.81	0.82	0.85	0.84	0.9
■ Effective Utilization	0.93	0.91	0.87	0.87	0.84	0.93

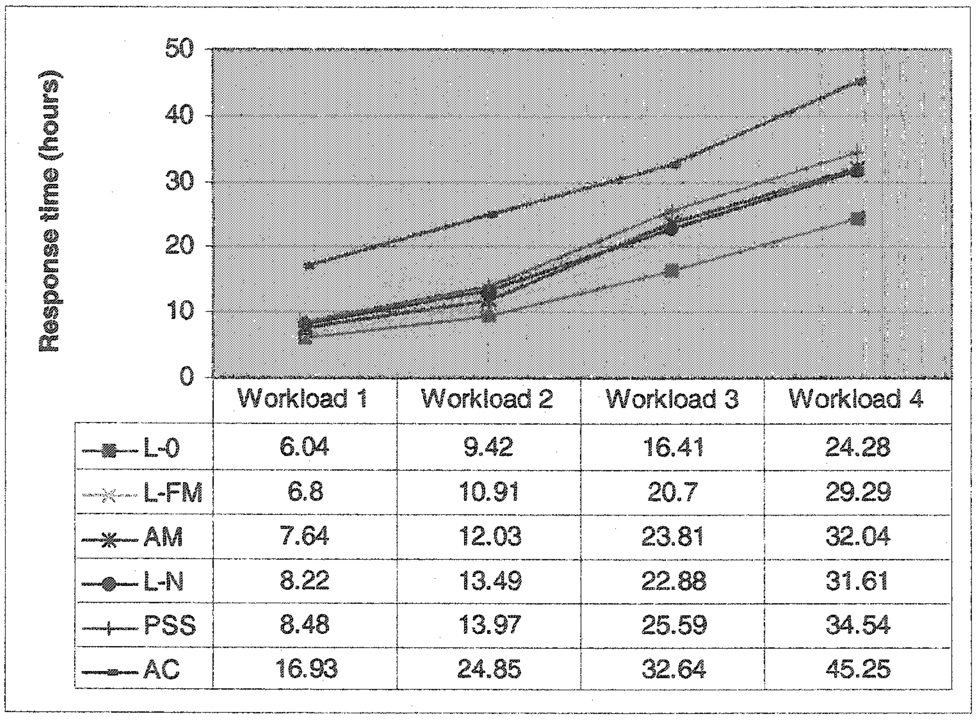
(c)

Figure 6.7. Experimental results under Workload 1.

From these results, we observe that our approach, L-0, has an improvement of 29% compared to standard space sharing and over 65% compared to AC in response time. For relative bounded response time, L-0 shows an improvement of 40% compared to PSS, and 65% improvement over AC. The result shows that utilization for our approach is under 80%, so we can see that under Workload 1, the system is underloaded. We observe that L-0, which has the best response time performance, has the lowest utilization. The explanation of this is that under the same workload, if jobs finish more quickly, there will be more processors left idle, i.e. have no jobs running on it. L-0 shows the most significant effective utilization improvement compared to its utilization value, and this means that L-0 got the best benefit from coscheduling.

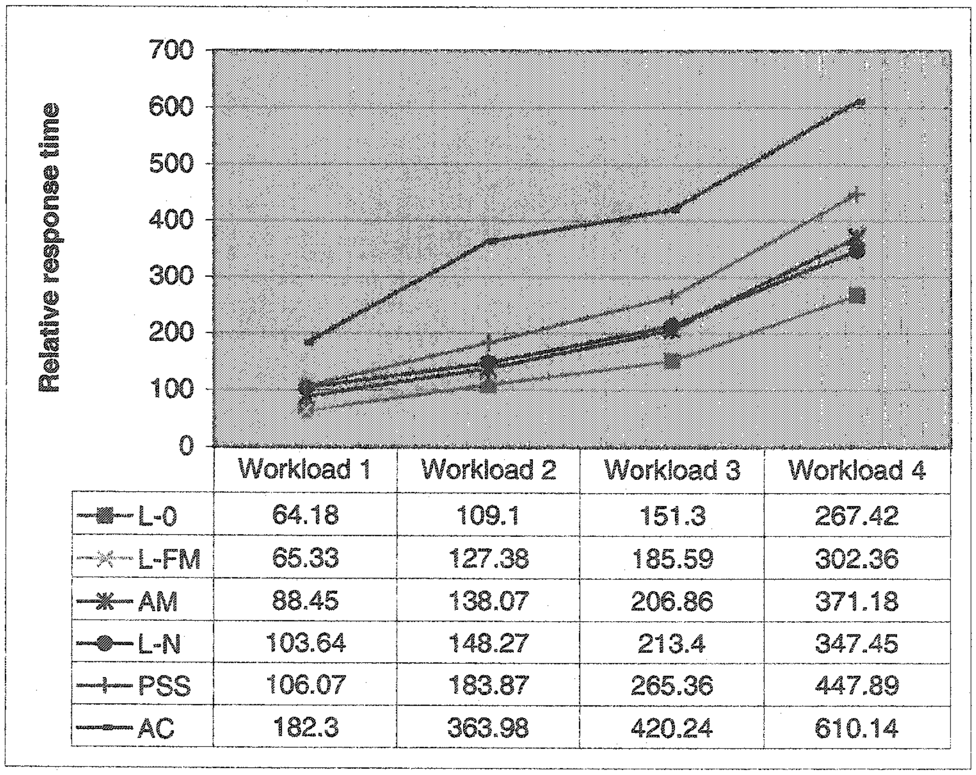
Observing that the system is underloaded to a certain degree with Workload 1, and our approach shows around 30% improvement in response time compared to the standard space sharing, we model Workload 2 with a Load value increased by 30% compared to Workload 1. In addition, we model Workload 3 and Workload 4 with Load values twice and three times respectively as the Load value of Workload 1.

Figure 6.8 and Figure 6.9 show how the increased Load can have an impact on the performance of all scheduling approaches involved in our experiment. Figure 6.8(a) presents the response time results and (b) shows relative bounded response time results. Figure 6.9(a) shows the comparison of utilization among all approaches, and Figure 6.9(b) shows the effective utilization results. We can see that with workload becoming heavier, our approaches, L-0, L-FM and L-N, show more obvious improvement over other approaches in response time, relative bounded response time and effective utilization. The reason for this is intuitively clear. The meaning for designing more sophisticated schedulers is to schedule jobs more efficiently under even a heavier workload, because if workload is very light, for an extreme example, all schedulers will behave same as the basic FCFS (First Come First Serve) scheduler. With the Load value being increased, the improvement in response time of L-0 increases from 29% to 36%, and the improvement in relative bounded response time of L-0 increases from 40% to 46% compared to PSS. Under all workloads, all approaches have similar utilization values, because utilization, due to its definition, is mainly decided by workload, packing scheme (backfilling) and job size characteristics, which are the same for all approaches. We can see that the utilization values for Workload 3 and Workload 4 do not show much difference, and this suggests that the system is saturated under Workload 3.



	Workload 1	Workload 2	Workload 3	Workload 4
L-0	6.04	9.42	16.41	24.28
L-FM	6.8	10.91	20.7	29.29
AM	7.64	12.03	23.81	32.04
L-N	8.22	13.49	22.88	31.61
PSS	8.48	13.97	25.59	34.54
AC	16.93	24.85	32.64	45.25

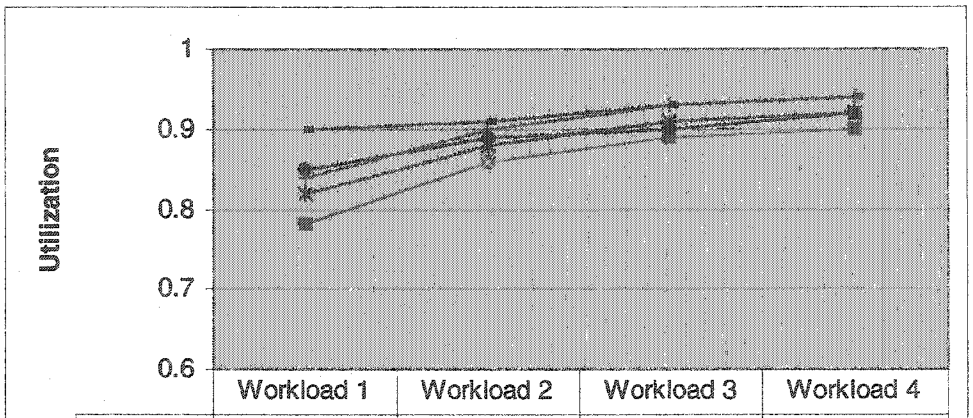
(a)



	Workload 1	Workload 2	Workload 3	Workload 4
L-0	64.18	109.1	151.3	267.42
L-FM	65.33	127.38	185.59	302.36
AM	88.45	138.07	206.86	371.18
L-N	103.64	148.27	213.4	347.45
PSS	106.07	183.87	265.36	447.89
AC	182.3	363.98	420.24	610.14

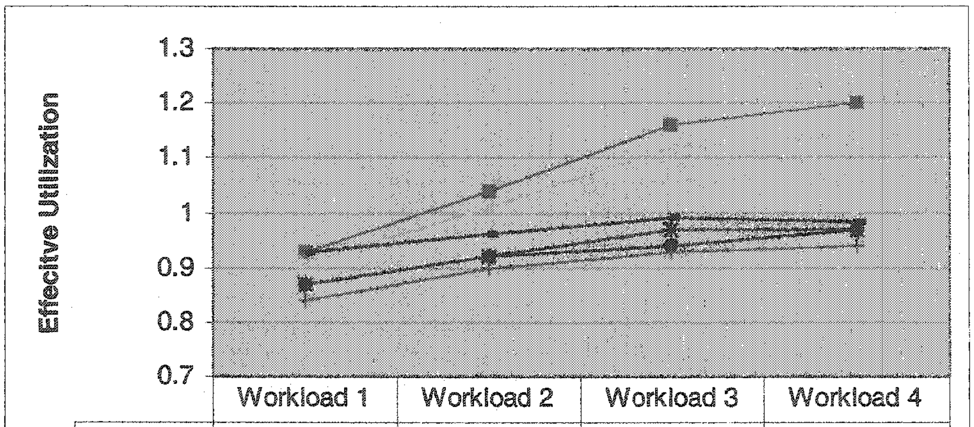
(b)

Figure 6.8. Workload impact on response time and relative bounded response time.



	Workload 1	Workload 2	Workload 3	Workload 4
■ L-0	0.78	0.86	0.89	0.9
× L-FM	0.81	0.86	0.91	0.92
✳ AM	0.82	0.88	0.91	0.92
● L-N	0.85	0.89	0.9	0.92
+ PSS	0.84	0.9	0.93	0.94
▪ AC	0.9	0.91	0.93	0.94

(a)



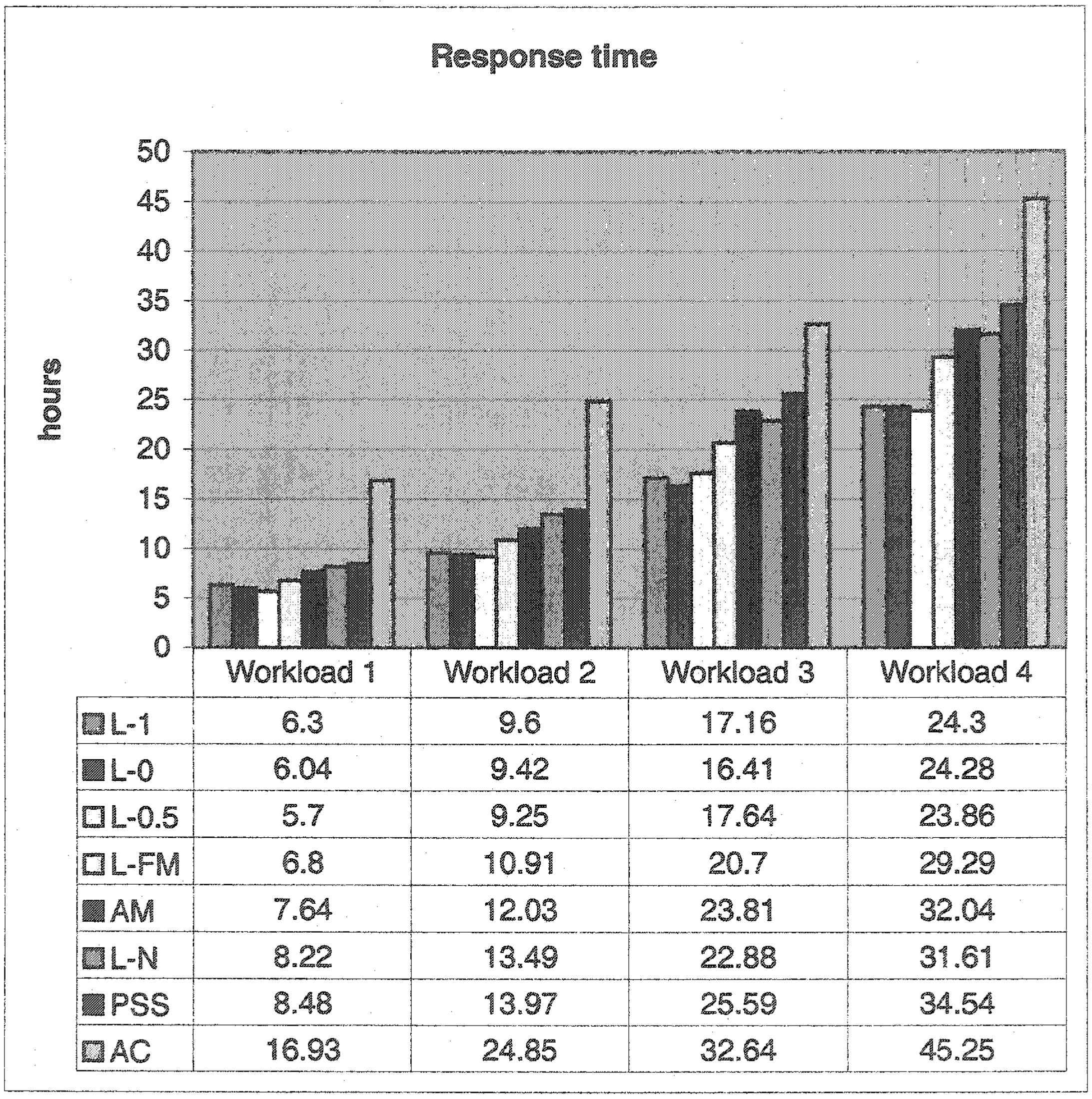
	Workload 1	Workload 2	Workload 3	Workload 4
■ L-0	0.93	1.04	1.16	1.2
× L-FM	0.91	1.01	1.12	1.14
✳ AM	0.87	0.92	0.97	0.97
● L-N	0.87	0.92	0.94	0.97
+ PSS	0.84	0.9	0.93	0.94
▪ AC	0.93	0.96	0.99	0.984

(b)

Figure 6.9. Workload impact on utilization and effective utilization.

6.3.4. Comparison of different approaches

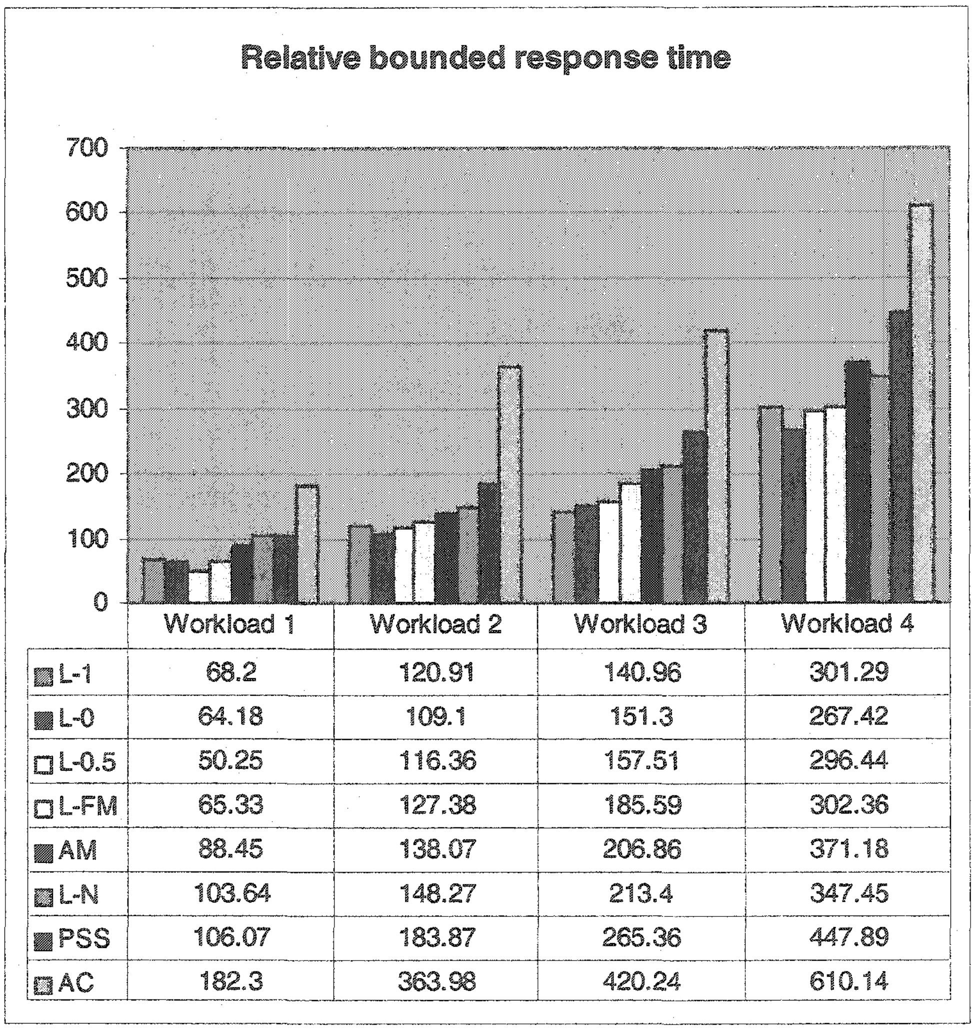
We compared several variants of LOMARC with PSS, AC and AM under all four workloads. In addition to L-0, L-FM and L-N, we also tested L-1, LOMARC with the WEIGHT value as 1, and L-0.5, LOMARC with the WEIGHT value as 0.5. Figure 6.10 shows the response time comparison under the four workloads.



	Workload 1	Workload 2	Workload 3	Workload 4
▨ L-1	6.3	9.6	17.16	24.3
■ L-0	6.04	9.42	16.41	24.28
□ L-0.5	5.7	9.25	17.64	23.86
□ L-FM	6.8	10.91	20.7	29.29
■ AM	7.64	12.03	23.81	32.04
▨ L-N	8.22	13.49	22.88	31.61
■ PSS	8.48	13.97	25.59	34.54
▨ AC	16.93	24.85	32.64	45.25

Figure 6.10. Response time comparison

We observe that for all workloads, L-0 shows the best response time, while AC shows the worst performance in response time. This demonstrates the importance of taking job resource consuming types into consideration for coscheduling. In AC, any two medium or long jobs can be coscheduled on HT processors, so the possibility of that two coscheduled jobs interfering with each other is high, and hence the performance is degraded seriously. The result shows that AC even performs worse than PSS, and this justifies that if job coscheduling on HT CPUs is not applied properly, system performance could be degraded seriously [Leng02]. AM is an approach that only coschedules two adjacent jobs if they are matchable under LOMARC matching scheme. The performance of AM is in between of LOMARC approaches and PSS; this is because only considering two adjacent jobs without looking ahead on the waiting queue, there will be less opportunity for job coscheduling.



	Workload 1	Workload 2	Workload 3	Workload 4
▨ L-1	68.2	120.91	140.96	301.29
■ L-0	64.18	109.1	151.3	267.42
□ L-0.5	50.25	116.36	157.51	296.44
□ L-FM	65.33	127.38	185.59	302.36
■ AM	88.45	138.07	206.86	371.18
▨ L-N	103.64	148.27	213.4	347.45
■ PSS	106.07	183.87	265.36	447.89
▨ AC	182.3	363.98	420.24	610.14

Figure 6.11. Relative bounded response time comparison

L-N shows around 15% improvement compared to PSS under saturated workloads. This suggests that by using time sharing to coschedule two jobs on standard CPUs, the performance can be improved due to I/O latency hiding. L-FM outperforms all other approaches except other LOMARC variants. This shows finding a match for every medium or long job is important for improving performance. However, without considering the utilization gain and response time impact on other jobs, L-FM has worse response time than other LOMARC variants. From Figure 6.11, we can see L-FM also has worse performance in relative bounded response time than other LOMARC variants. This implies that finding the best match plays a role in improving the overall performance.

	Average Queue Length	Medium or Long Job	SizeB ≤SizeA	Memory Fit	Matchable	Slowdown ≤MAX
Workload 1	46	29	16	12	5	5
Workload 2	67	36	20	14	6	5
Workload 3	111	51	22	14	6	5
Workload 4	209	56	25	16	7	6

Table 6.6. Average queue lengths and average numbers of jobs suitable for coscheduling

Figure 6.10 and Figure 6.11 show that there are no obvious differences among L-1, L-0 and L-0.5, while L-0 has a slightly better performance than the other two. The explanations for this result are as follows. First, even though utilization and response time are two different goals and there is a trade-off between them, these two goals do not contradict with each other, and the fact is that in most cases, improved utilization can lead to better response time. Second, in the utilization calculation, we also considered the slowdown factor of two coscheduled jobs. This means if one match candidate has better utilization gain from job coscheduling, it also has lower slowdown from job coscheduling, and this leads to less delay to other jobs, i.e. better response impact on other jobs. Third, to further analyse this result, we need to know what happens when searching for the best match. Table 6.6 shows the average queue lengths and the number of jobs left in each step for finding the match. It is important to note that the average queue length is the average of the whole waiting queue not the search length. For finding the match of a job, we only search the jobs behind it in the queue. After meeting all the constraints for coscheduling, the number of jobs left as the candidates to choose from is small. Even in the heaviest workload, the average number of candidates is only 5. With this small number of match candidates, the possibility of choosing the same one under different optimization goals will be high. This also explains why when using different WEIGHT value, there is little difference in performance result.

Figure 6.12 shows the comparison of utilization (the left set of data) and effective utilization (the right set of data) for all approaches under the four workloads. For utilization, all approaches have similar values under certain workloads for the reason we discussed in the previous section. For effective utilization, our approaches show improvement especially under heavier workloads. In addition, with our approaches, the effective utilization increases more compared to utilization under the same workload. This means that our approaches have more efficient usage of the machine.

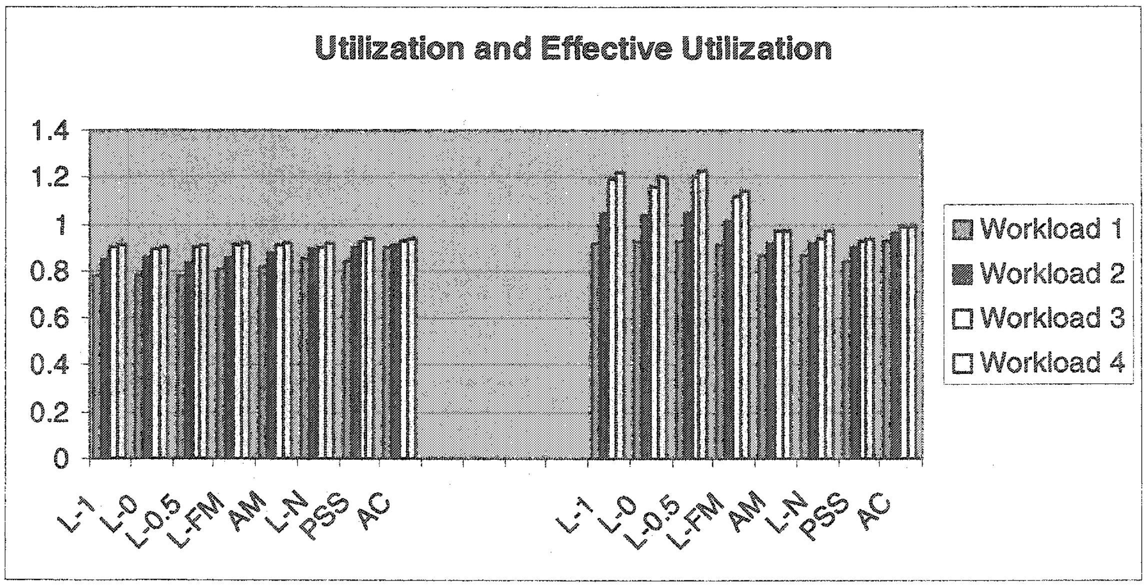


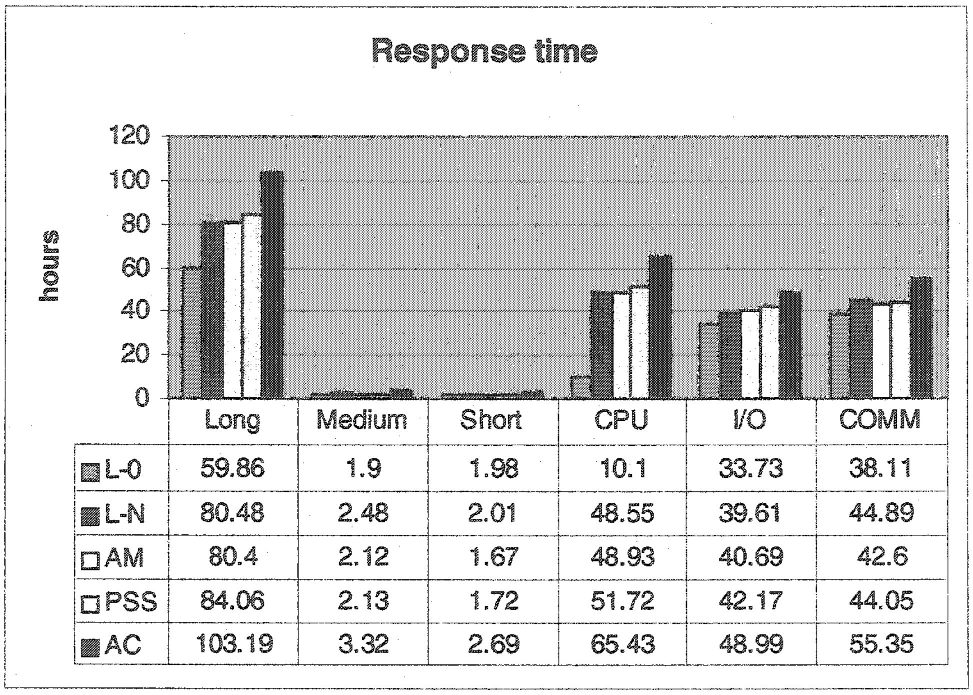
Figure 6.12. Comparison of utilization and Effective utilization

6.3.5. Performance for different job classes

To show how different approaches have impacts on different job runtime classes and resource consuming types, we tested the individual performance for each job class. Figure 6.13 shows the performance comparison among long, medium, short, CPU intensive, I/O intensive and communication intensive (COMM) jobs respectively. Figure 6.13(a) shows the response time comparison and Figure 6.13(b) shows relative bounded response time comparison. The scheduling approaches involved in the comparison are L-0, L-N, AM, PSS and AC, and the workload tested is Workload 3. CPU, I/O and communication intensive jobs are either medium or long jobs.

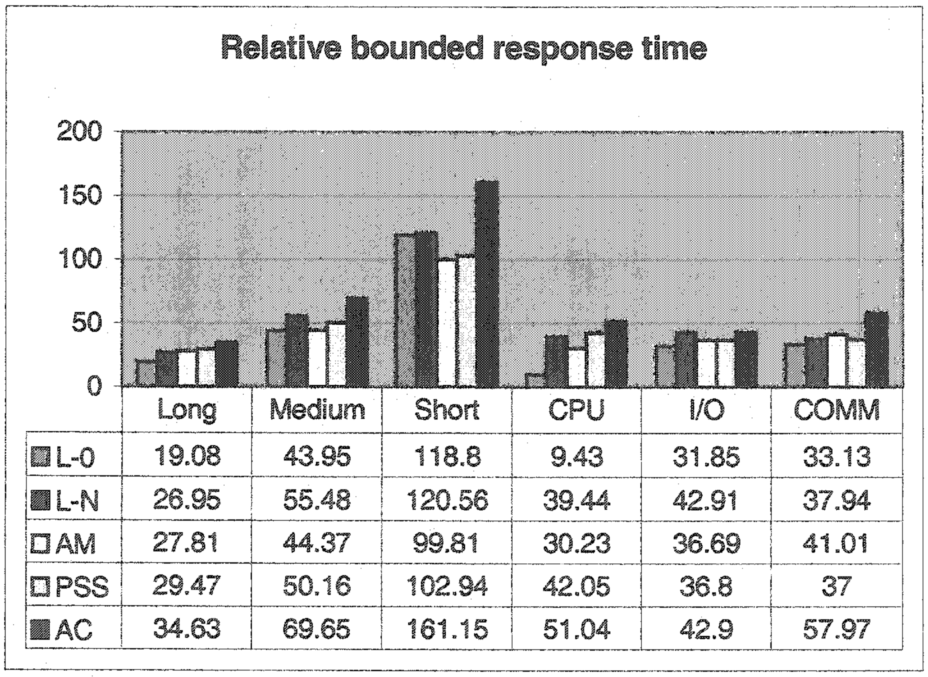
From the result, we observe that in general, long jobs and medium jobs have better relative bounded response time compared to short jobs. This is the common feature of schedulers with no preemption, because when all the nodes are occupied by large jobs, a short job tends to wait much longer than its runtime. The result shows that our approach, L-0 has worse response time and relative bounded response time for short jobs than PSS. The reason for this is that our approach favors medium and long jobs by moving them ahead for job coscheduling.

For different job resource consuming types, we can see in our approach, CPU intensive jobs get the most benefit. This is because under the LOMARC matching scheme, CPU intensive jobs have more opportunity to coschedule with other jobs. This result implies that coscheduling two matchable jobs is the main reason for performance improvement in our approach.



	Long	Medium	Short	CPU	I/O	COMM
▨ L-0	59.86	1.9	1.98	10.1	33.73	38.11
■ L-N	80.48	2.48	2.01	48.55	39.61	44.89
□ AM	80.4	2.12	1.67	48.93	40.69	42.6
□ PSS	84.06	2.13	1.72	51.72	42.17	44.05
■ AC	103.19	3.32	2.69	65.43	48.99	55.35

(a)



	Long	Medium	Short	CPU	I/O	COMM
▨ L-0	19.08	43.95	118.8	9.43	31.85	33.13
■ L-N	26.95	55.48	120.56	39.44	42.91	37.94
□ AM	27.81	44.37	99.81	30.23	36.69	41.01
□ PSS	29.47	50.16	102.94	42.05	36.8	37
■ AC	34.63	69.65	161.15	51.04	42.9	57.97

(b)

Figure 6.13. Comparison among different job classes.

6.3.6. Summary

The above experiment results provided sound evidence that the LOMARC algorithm can deliver better performance compared to PSS, AC and AM for almost all metrics measured such as average response time, average relative bounded response time and effective utilization. With heavier workload, our approach can show more improvement over the other approaches in overall performance. As regards to utilization, our approach has similar results with other approaches. This is due to the fact that utilization mostly depends on workload, packing scheme and job size characteristics, which are common for all approaches involved in our experiment.

L-N shows less improvement compared to other LOMARC variances. The reason for this is that in normal CPU environment, LOMARC only coschedules CPU and I/O intensive jobs, and this reduces the chance for job coscheduling. In addition, coscheduling two jobs on standard CPUs tends to have a higher slowdown factor, compared to jobs coscheduled on HT processors, according to our slowdown estimation model.

Among L-0, L-1 and L-0.5, there is no significant difference in the results, while L-0 shows slightly better performance. In addition, we tested a simplified LOMARC variant L-FM, which only finds the first match without considering utilization and response time impacts. The results show L-FM performs constantly worse than the other three variances. This suggests that considering utilization and response time for choosing a match does play a role in improving overall performance.

For different job runtime classes, our approach favors medium and long jobs more compared to standard space sharing with priority and backfilling. The reason for this is the LOMARC gives medium and long jobs more opportunity to move ahead for coscheduling. Among different job resource consuming types, CPU intensive jobs get most benefit because they have more chances of coscheduling with other jobs.

7. Conclusion and Future Work

7.1. Conclusion

Most scheduling approaches only focus on the CPU without considering the impact on other resources. The goals of our approach, LOMARC, are to take all application resource requirements, such as for CPU, I/O, memory and network, into consideration, and exploit space-time sharing provided by the HT technology. To improve utilization and reduce response times, LOMARC matches two jobs with complementary resource requirements for coscheduling. LOMARC partially reorders the waiting job queue by lookahead to increase the possibility of finding a good match. When choosing a match for one job, we estimate the utilization gain and response time impact on other jobs resulting from the matching, and choose the best match according to the combined utilization gain and response time impact value. LOMARC generalizes for standard CPUs, using an adjusted matching scheme and only focusing on hiding I/O latency. In addition, LOMARC incorporates standard scheduling approaches such as priority ordering, aging and backfilling.

The experimental results show that our approach can deliver better overall performance compared to standard space sharing with priority and backfilling. We also compared our approach with two other approaches for HT processors; one coschedules any two jobs on HT processors without considering job resource consuming types, and the other only coschedules two adjacent jobs if they are complementary regarding their resource types. The results show that LOMARC outperforms these approaches, and can show more improvement under heavier workloads. The performance gain is mainly due to increased possibility of coscheduling two complementary jobs by looking ahead on the waiting queue. Varying the WEIGHT value in choosing the best match does not affect the overall performance obviously. This is different than our expectation. However, a simplified LORMAC variant that only finds the first match shows worse performance. This suggests that considering both response time and utilization impact from coscheduling plays a role in performance improvement.

7.2. Future Work

The future work of this thesis can involve the following three aspects.

First, our current slowdown modeling is based on some previous research results, which are not comprehensive to cover all possible coscheduling cases. It would be meaningful to do more experiment in a real HT processor environment with different job resource types, and test the slowdown factor for different coscheduling possibilities.

Second, as we have seen in our experiment, our current heuristic for choosing the best match using different WEIGHT values does not show obvious difference. It would be interesting to try other simplified strategies in choosing the best match. One possible solution is to only consider the first three match candidates according to the slowdown factors and their sizes and runtimes comparing with the current job to be scheduled.

Third, the current workload modeling in resource consumption characteristics is simple. In future research, it would be important to examine job resource consumption characteristics in real workloads and extract a statistical model to represent the real workload.

References

[Batat00] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations", Proceedings of 14th International Parallel & Distributed Processing Symposium (IPDPS'2000), May 2000, pp. 109-114.

[Burger96] D. Burger, R. Hyder, B. Miller and D. Wood. "Paging Tradeoffs in Distributed Shared-Memory Multiprocessors". In Journal of Supercomputing, vol. 10, 1996. 594, 596

[Chandra94] R. Chandra, S. Devine, and B. Verghese. "Scheduling and page migration for multiprocessor computer servers". In 6th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1994

[Chiang01] S.-H. Chiang and M.K. Vernon. "Characteristics of a Large Shared Memory Production Workload". Proc. JSSPP, 2001.

[Dorai02] G. K. Dorai and D. Yeung, "Transparent Thread: Resource Sharing in SMT Processors for High Single-Thread performance". In proceedings of the 11th Annual International Conference on Parallel Architectures and Compilation Techniques, Charlottesville, VA, Septemper 2002.

[Dusseau96] C. Dusseau, R. H. Arpaci, and D. E. Culler. "Effective Distributed Scheduling of Parallel Workloads". In Proceedings of the 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, Philadelphia, PA, May 1996.

[Dusseau98] C. Dusseau, D. Culler, and A. M. Mainwaring. "Scheduling with Implicit information in Distributed Systems". In Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, Madison, WI, June 1998.

[Fabricio99] Fabricio Alves Barbosa da Silva and Isaac D. Scherson, "Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler", 11th Symposium on Computer Architecture and High Performance Computing, Natal, Brazil, September 1999.

[Feitelson92] D. G. Feitelson and L. Rudolph. "Gang Scheduling Performance Benefits for Fine-Grained Synchronization". Journal of Parallel and Distributed Computing, 16(4):306–18, December 1992.

[Feitelson97A] D.G. Feitelson. "A Survey of Scheduling in Multiprogrammed Parallel Systems". Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, Second Revision, February 1997

[Feitelson97B] D. G. Feitelson and Morris A. Jette. "Improved Utilization and Responsiveness with Gang Scheduling". In Dror G. Feitelson and Larry Rudolph, editors, Job Schedul- ing Strategies for Parallel Processing, volume 1291 of Lecture Notes in Computer Science, pages 238-261. Springer-Verlag, 1997.

[Feitelson97C] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. "Theory and Practice in Parallel Job Scheduling". In D. Feitelson and L. Rudolph, editors, 3rd Workshop on Job Scheduling Strategies for Parallel Processing, volume 1291 of LNCS, pages 1--34. Springer-Verlag, 1997.

[Feitelson98A] D. G. Feitelson and Larry Rudolph. "Metrics and Benchmarking for Parallel Job Scheduling". In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), pp. 1-24, Springer-Verlag, Lecture Notes in Computer Science vol. 1459, 1998.

[Feitelson98B] D. G. Feitelson and A. M.Weil. "Utilization and predictability in scheduling the IBM SP2 with backfilling". In 12th International Parallel Processing Symposium, pages 542–546, April 1998.

[Frachtenberg03] Frachtenberg E., Feitelson D. G., Petrini F. and Fernandez J., "Flexible CoScheduling: mitigating load imbalance and improving utilization of heterogeneous resources", 17th Intl. Parallel & Distributed Processing Symp., Apr 2003.

[Hori98] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, NoriYuki Soda, Hiroki Konaka, and Muneori Maeda. "Overhead Analysis of Preemptive Gang Scheduling". In Dror G. Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1459 of Lecture Notes in Computer Science, pages 217--230. Springer-Verlag, 1998.

[Lee97] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. "Implications of I/O for Gang Scheduled Workloads". In D. G. Feitelson and L. Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1291 of Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Leng02] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. "An Empirical Study of Hyper-Threading in High Performance Computing Clusters". Linux HPC Revolution, 2002.

[Lifka95] D. Lifka. "The ANL/IBM SP Scheduling System". Proc. Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science, Springer Verlag, Vol. 949, 1995.

[Lublin01] U. Lublin and D. G. Feitelson, The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. Technical Report 2001-12, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Oct 2001.

[Magro02] Wiliam Magro, Paul Peterson, and Sanjiv Shah. "Hyper-Threading Technology: Impact on Compute-Intensive Workloads". Intel Technology Journal Q1, Vol. 6, No. 1, 2002.

[Nakajima02] Jun Nakajima and Venkatesh Pallipadi. "Enhancements for Hyper-Threading Technology in the Operating System – Seeking the Optimal Scheduling". Proc. USENIX 2nd Workshop on Industrial Experiences with Systems Software, Boston/MA, USA, Dec. 2002.

[Ousterhout82] Ousterhout, J. "Scheduling techniques for concurrent systems". Proceedings of the 3rd International Conference on Distributed Computing Systems. Oct. 1982, pp. 22--30.

[Peris94] Vinod G. J. Peris, Mark S. Squillante, and Vijay K. Naik. "Analysis of the impact of memory in distributed parallel processing systems". In Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 5–18, 1994.

[Perkovic00] Dejan Perkovic and Peter J. Keleher. "Randomization, Speculation, and Adaptation in Batch Schedulers". Proc. Supercomputing (SC2000), Dallas/TX, Nov. 2000.

[Petrini99] Fabrizio Petrini, Federico Bassetti, and Alex Gerbessiotis. "A New Approach to Parallel Program Development and Scheduling of Parallel Jobs on Distributed Systems". In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), volume I, pages 546-552, Las Vegas, NV, July 1999.

[Petrini00] F. Petrini and W. Feng. "Time-Sharing Parallel Jobs in the Presence of Multiple Resource Requirements". In Proc. of IPDPS 2000 Workshop on Job Scheduling Strategies for Parallel Processing, Cancun, Mexico, May 2000. Springer.

[Rosti98] Rosti E., Serazzi G., Smirni E., Squillante M.S., "The Impact of I/O on Program Behavior and Parallel Scheduling", in Proc. Of the Joint ACM SIGMETRICS'98 Conference on the Measurement and Modeling of Computer Systems and Performance'98, pp 57-65, 1998.

[Setia99] S. Setia, M. S. Squillante and V.K. Naik. "The Impact of Job Memory Requirements on Gang-Scheduling Performance". In Performance Evaluation Review,1999.

[Shmueli03] E. Shmueli and D. G. Feitelson, "Backfilling with lookahead to optimize the performance of parallel job scheduling". In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.), pp. 228-251, Springer-Verlag, 2003. Lecture Notes in Computer Science Vol. 2862.

[Smirni98] E. Smirni and D. Reed. "Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications". Performance Evaluation: An International Journal, 33(1):27--44, June 1998.

[Snavely02] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor'. in Proceedings of the International Conference on Measurement and Modeling of Computer Systems, June 2002.

[Sobalvarro98] P.G.Sobalvarro, S.Pakin, W.E.Weihl and A.A.Chien. "Dynamic coscheduling on workstation clusters". In Job Scheduling Strategies for Parallel Processing, pp.231-256, Springer-Verlag, 1998.

[Sobalvarro97] Patrick Sobalvarro. "Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors". Job Scheduling Strategies for Parallel Processing, 1997.

[SodanHuang04] Angela C. Sodan and Xuemin Huang. "Adaptive Time/Space Sharing with SCOJO". Accepted for HPCS, Manitoba, 2004.

[Subhlok96] Jaspal Subhlok, Thomas Gross, Takashi Suzuoka. "Impact of job mix on optimizations for space sharing schedulers". In Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) November 1996

[Talby99] Talby D and Feitelson D G. "Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling". Proc. IPPS, 1999.

[Tullsen00] D. M. Tullsen and A. Snavely. "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor". In 9 th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[Wiseman03] Y. Wiseman and D. G. Feitelson, "Paired Gang Scheduling". IEEE Trans. Parallel & Distributed Systems, 2003.

[Zhang02] Y. Zhang, "Scheduling and Resource Management for Next Generation Clusters". Ph.D. thesis dissertation, Department of Computer Science and Engineering The Pennsylvania State University, August 2002.

[Zhou98] B. Zhou, R. Brent, D. Walsh, and K. Suzaki. "Job scheduling strategies for networks of workstations". In Job Scheduling Strategies for Parallel Processing, volume 1459 of LNCS, pages 143-- 157, Berlin, 1998. Springer

Vita Auctoris

NAME:	Lei Lan
PLACE OF BIRTH:	Beijing, China
YEAR OF BIRTH:	1972
EDUCATION:	The Ninth High School, Beijing, China 1987-1990
	Northeastern University, Shenyang, China 1990-1994 B.En.
	University of Windsor, Windsor, Ontario, Canada 2001-2004 M.Sc.