University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations Theses, Dissertations, and Major Papers

2004

Adaptive space-time sharing with SCOJO.

Xuemin Huang
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Huang, Xuemin, "Adaptive space-time sharing with SCOJO." (2004). *Electronic Theses and Dissertations*. 892.
https://scholar.uwindsor.ca/etd/892

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

ADAPTIVE SPACE-TIME SHARING WITH SCOJO

by

Xuemin Huang

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

Windsor, Ontario, Canada

2004

© 2004 Xuemin Huang

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-92522-6
Our file Notre référence
ISBN: 0-612-92522-6

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canadä

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

Coscheduling is a technique used to improve the performance of parallel computer applications under time sharing, i.e., to provide better response times than standard time sharing or space sharing. Dynamic coscheduling and gang scheduling are two main forms of coscheduling. In SCOJO (Share-based Job Coscheduling), we have introduced our own original framework to employ loosely coordinated dynamic coscheduling and a dynamic directory service in support of scheduling cross-site jobs in grid scheduling. SCOJO guarantees effective CPU shares by taking coscheduling effects into consideration and supports both time and CPU share reservation for cross-site job. However, coscheduling leads to high memory pressure and still involves problems like fragmentation and context-switch overhead, especially when applying higher multiprogramming levels. As main part of this thesis, we employ gang scheduling as more directly suitable approach for combined space-time sharing and extend SCOJO for clusters to incorporate adaptive space sharing into gang scheduling. We focus on taking advantage of moldable and malleable characteristics of realistic job mixes to dynamically adapt to varying system workloads and flexibly reduce fragmentation. In addition, our adaptive scheduling approach applies standard job-scheduling techniques like a priority and aging system, backfilling or easy backfilling. We demonstrate by the results of a discrete-event simulation that this dynamic adaptive space-time sharing approach can deliver better response times and bounded relative response times even with a lower multiprogramming level than traditional gang scheduling.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

We thank all committee members of this thesis for their many suggestions and generous help to improve this thesis. We appreciate Dr. Schurko[1]'s, Mr. Ron Dumouchelle[2]'s, and Dr. Sodan[3]'s additional assistance on polishing up the English writing of this thesis.

[1] External reader of this thesis, Dept. of Chemistry and Biochemistry, University of Windsor
[2] Staff, Academic Writing Center, University of Windsor
[3] Advisor of this thesis, Dept. of Computer Science, University of Windsor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

LIST OF FIGURES

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1: INTRODUCTION

The general job-scheduling problem in parallel-multiprogrammed systems refers to assigning tasks from concurrent competing programs to multiple processors, in order to minimize the makespan, i.e., largest task completion time [Feitelson97] or average relative response time, i.e., the ratio of the response time (the time from task submittal to task termination) to the task execution time [Naik97]. One program can be thought of as one job or task, and each job can contain several processes. Therefore, the job-scheduling problem is really a very complex two-level issue: both on the operating system level and on application level.

On the operating system level, job-scheduling involves allocation of multiple resources among jobs, e.g. processors and memory, so as to decide when to run which job on what processors. Because processors are the most important resource, a lot of research only concentrates on processor allocation while ignoring or simplifying other resources. There are three basic approaches to processor allocation: time sharing, space sharing, and the combination of the time sharing and space sharing, i.e., space-time sharing. Time sharing means all processors serve the global job queue and the processors are quickly switched from one job to another after a certain time interval. Space sharing means that processors are partitioned statically or dynamically to satisfy different resource requirements of different jobs and tends to provide each job a more dedicated or exclusive processor allocation than time sharing [McCann93]. As the combination of time sharing and space sharing, space-time sharing has been widely proved [Tucker89][Feitelson97B] to gain better responsiveness and efficient use of resources than pure time sharing and space sharing. On the application level, job-scheduling involves scheduling all processes of a job among assigned processors efficiently. This needs both effort from application developer and runtime system support such as thread library, parallel compiler, etc. [Feitelson95A]. There are lots of scheduling techniques and algorithms that have been developed on both levels, and many factors affect their performance, such

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as machine architecture, characteristics of workload, job flexibility, application information, etc. Such related issues will be discussed in Chapter 2.

Ousterhout [1982] introduced coscheduling to improve the performance of parallel applications under time sharing, which tries to maximize coscheduled tasks. When a task is coscheduled all processes of this task are executed simultaneously on different processors. Gang scheduling and dynamic coscheduling are two main forms of coscheduling. Gang scheduling [Ousterhout82][Feitelson97] or explicit coscheduling ensures that no process will wait for non-scheduled process of the same task for communication or synchronization so as to minimize the waiting time at the synchronization point, i.e., all processes of the same job are executing or suspending simultaneously. On the other hand, dynamic coscheduling [Sobalvarro98][Sobalvarro97] tries to take advantage of application communication behavior to approximate coscheduled execution without the need for synchronization among processes, i.e., to decrease the coordination effort. For example, if one job is blocked for I/O operation, it can obviously improve overall job performance by overlapping another job that is computationally intensive. This advantage of dynamic coscheduling is also called latency (communication or I/O) hiding. Demand-based coscheduling [Sobalvarro97] is one mechanism of dynamic coscheduling, which only guarantees to coschedule those processes that communicate with each other. Implicit coscheduling [Sobalvarro98][Sobalvarro97] is another mechanism of dynamic coscheduling, which uses spin-block technique; this means that a blocked process will spin a pre-determined time for messages. If this blocking process can receive message before the time expires, then it will continue to run. Otherwise, it will be blocked and another one is scheduled. Details are described in Chapter 3.

Dynamic processor partitioning refers to dynamically changing the number of processors allotted to jobs during job execution according to the system workload changes and/or user requirement. It is fundamental to the design of adaptive scheduling strategies. Some existing adaptive scheduling techniques such as the general dynamic scheduling policy (DP) [McCann93] for shared-memory

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

multiprocessor systems and equipartition [McCann94] for distributed-memory message-passing multiprocessor systems will be discussed in Chapter 4.

This thesis consists of two parts: one is SCOJO (Sharing-based Job Coscheduling with Integrated Dynamic Resource Directory in Support of Grid Scheduling) [SodanHuang03], and adaptive SCOJO (Adaptive Space-time Sharing with SCOJO), which is a great improvement over SCOJO but with different focus.

SCOJO provides a local framework in support of grid computing. It is our own approach that combines time sharing and batch scheduling (scheduling a batch of parallel jobs). We employ dynamic coscheduling with loose coordination which takes coscheduling effects into consideration (i.e. takes advantage of dynamic coscheduling, e.g. latency hiding) as well as application characteristics. In addition, SCOJO guarantees the reservation in terms of both start time and CPU share for cross-site jobs, which might be scheduled and executed on multiple sites, and provides a dynamic directory service that keeps information about both application and machine. SCOJO is briefly introduced in Chapter 5.

However, SCOJO still has problems like memory pressure, context-switching overhead, and fragmentation, which are general problems of standard time sharing. Moreover, we assume that all jobs require all processor resources in SCOJO, which is not practical and needs to incorporate a certain degree of space sharing. Therefore, based on SCOJO, adaptive SCOJO goes to next level where it not only applies the combination of time sharing and space sharing but also employs adaptive resource allocation, i.e., it dynamically changes the number of processors allotted to jobs during runtime. However, due to the complexity and different goals of such dynamic adaptive space-time sharing approach from SCOJO, we keep all general considerations in SCOJO but exclude reservation for cross-site jobs and explicit coscheduling effects consideration; i.e., we use gang scheduling instead of dynamic coscheduling as the more directly suitable approach for combined space-time sharing. Then, the main focus of adaptive SCOJO is to try to achieve better job performance than standard gang scheduling by dynamically changing the processor allotment

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

during job runtime to reduce fragmentation and adapt to the constant changes of system workload. Adaptive SCOJO scheduling algorithm is proposed in Chapter 6 and the corresponding implementation and experimental results are shown in Chapter 7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2: BACKGROUND ISSUES

In this chapter, some important related background issues are explained. In general, there are three basic dimensions to job scheduling scheme design: time sharing vs. space sharing, non-preemption vs. preemption, and static partitioning vs. dynamic partitioning. Adaptive scheduling can only take advantage of certain types of jobs, e.g., moldable and malleable jobs. The more detailed and accurate the workload characteristics and application information the job scheduler can get either via runtime estimation or via application itself, then the more efficient schedule plan the job scheduler can make; i.e., the higher the overall job performance and system utilization.

2.1 TIME SHARING VS. SPACE SHARING

Time sharing is highly variable and can provide certain degree of fairness (e.g. many commercial operating systems use unlimited time slices in time sharing, i.e., jobs can be scheduled immediately after submission without starvation). It is especially suitable if the exact runtime or runtime estimation of jobs is unknown. However, if context switching and memory swapping are costly, time sharing will introduce a lot of overhead and performance loss due to the synchronization among processes of the same job.

Space sharing mainly tries to enhance the processor utilization by providing a dedicated or exclusive processor allocation among jobs. Most approaches for space sharing attempt to minimize the context-switching overhead against time sharing and reduce the loss of performance due to the synchronization problem of time sharing. The main drawback of space sharing is the fragmentation introduced by fixed processor allocation in the execution environment [Corbalan01].

Space-time sharing is the combination of time sharing and space sharing, which usually gains benefits from both time sharing and space sharing.

Figure 2-1 demonstrates the basic concept of time sharing, space sharing, and space-time sharing. In this example, some processors (marked with X) are idle, which means that these processors currently are not executing any jobs. We call

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

such idle processors as fragmentation, which wastes system resources. Therefore, fragmentation reduction is one of the main goals in job scheduling.

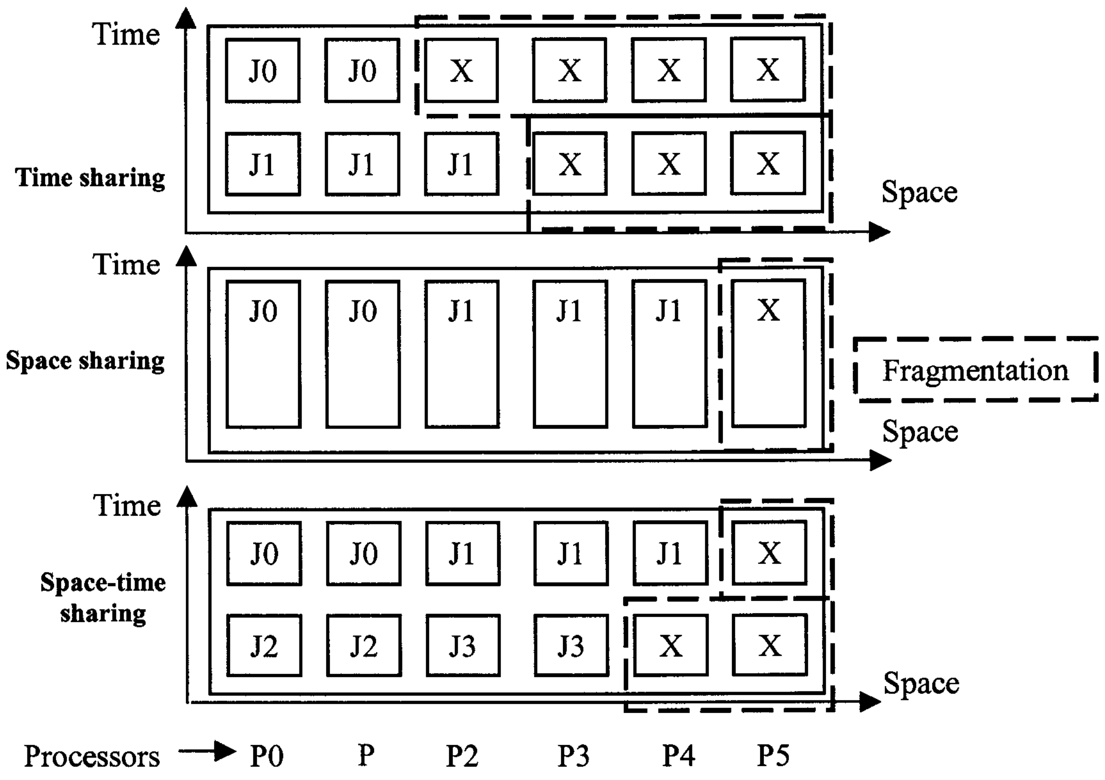


Figure 2-1. Time sharing vs. space sharing

2.2 NON-PREEMPTION VS. PREEMPTION

Non-preemption means that each job runs to completion without interruption on the set of processors initially allocated to it [Chiang94]. Standard space sharing implies non-preemption.

Preemption [Feitelson97B] means that job can be interrupted during its execution and be resumed on the same or a different set of processors initially allocated to it. Preemption will introduce significant overhead like context switching, memory swapping, etc. Standard time sharing implies preemption.

In real job scheduling-policy design, non-preemption or limited preemption is a general recommended direction [Feitelson97C][Chiang94] in order to avoid the overhead introduced by preemption. However, if application characteristics like execution time, are known before scheduling, then a scheduling policy that can take advantage of knowledge of application characteristics and adopt certain

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

degree of preemption or even time sharing would gain better performance in the situation where job parallelism is high [Majumdar88].

Figure 2-2 shows an example of preemption. Job 1 is executed in the first time interval ($T_{interval}$ - the period of time between two time slices in time sharing), preempted during the second and third time interval, and resumed in the fourth time interval.

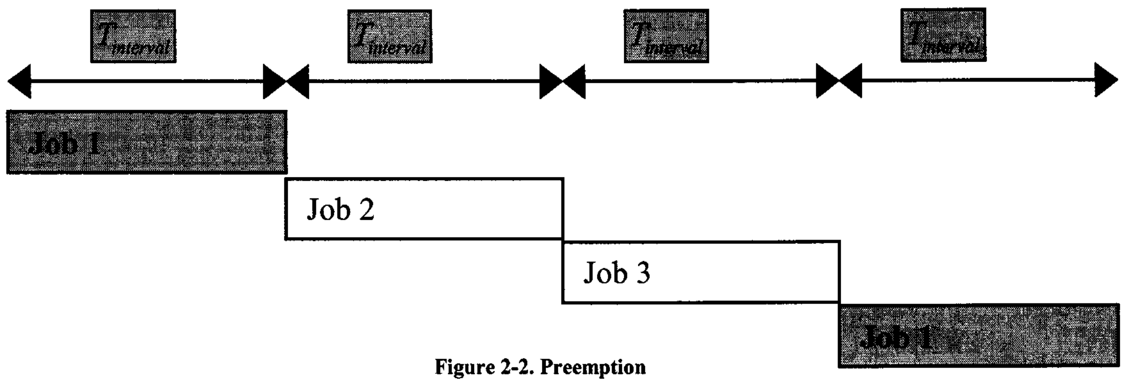


Figure 2-2. Preemption

2.3 PROCESSOR PARTITIONING

Each parallel job is executed on all or a subset of processors. The number of processors on which each job can run is called the size of the job. Processor partitioning means to partition all available processors among concurrently running jobs according to their sizes. Different computer architectures, operating systems, and application behavior determine the classification of processor partitioning. According to the work of D. G. Feitelson et al. [Feitelson97A][Feitelson97B], there are four basic processor-partitioning types:

- Static Partitioning

  The partition is preset by the system administrator and can only be changed by rebooting the system. It is simple and can keep high CPU cache locality, but will introduce internal fragmentation and has a limited degree of multiprogramming, i.e., limited number of jobs that can be executed concurrently.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Variable Partitioning

  The partition is set based on the user request when the job is submitted. It meets user's requirement and also has high CPU cache locality; however, it results in external fragmentation.
- Adaptive Partitioning

  The partition is determined by the scheduler according to the current workload when the job is initialized and also takes the user request into account. This approach can improve efficiency by its ability of adapting to workload and high CPU cache locality. Both external and internal fragmentation will be encountered.
- Dynamic Partitioning

  The partition can change dynamically during job execution to reflect the changes of workload and user requirement. This approach introduces little fragmentation, high efficiency, and extraordinary workload adaptation. However, it sets limitations on the programming model, and the communication cost associated with relocating code and data is very expensive.

It is important to note that processor partitioning is mostly related to space sharing. Moreover, processor partitioning can combine non-preemption or preemption together resulting in several new derived scheduling policies (See Majumdar88).

2.4 JOB FLEXIBILITY

Job flexibility refers to how applications are written, which determines what class of processor allocation strategy or scheduling policy should be used to get best performance. Feitelson and Rudolph [Feitelson96A] classify applications into four categories:

- Rigid job

  The job requires certain number of processors explicitly and cannot run on less or utilize more processors. The scheduler can do nothing but assign

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the required number of processors to jobs. Static or variable processor partitioning might be suitable for scheduling rigid jobs.

- Moldable job

  The size of a moldable job can be determined by the job scheduler based on the current workload when the job is first activated. Then moldable jobs will use the same size through the entire execution. Adaptive processor partitioning could be used to scheduling such kind of jobs.

- Evolving job

  The execution of an evolving job is divided into several phases. At the beginning of each phase, the evolving job might require a different number of processors; at the end of each phase, the job releases them. Variable or dynamic processor partitioning is suitable.

- Malleable job

  W. Ludwig and P. Tiwari have stated [in Ludwig94] that a malleable job is one that can be run on any number of processors, i.e., the size of a malleable job can be dynamically changed during its execution. As a result, the OS can ask a malleable job to release some processors when the system workload is heavy; on the other hand, a malleable job can be given additional processors by the OS if the system workload is light or more processors are available. Much research has tried to take advantage of malleable jobs in order to enhance processor utilization and improve overall job performance. To make an application malleable, the application itself should be written in such way that it could dynamically adjust the set of processors initially assigned to it during execution. In addition, the job scheduler should be constantly aware of the workload changes, then expand or shrink the size of malleable jobs correspondingly. The dynamic processor partitioning must be used to provide such size adaptation capability.

It is important to note that some theoretical studies use different terminology. For example, most pure algorithmic research [Ludwig94][Turek92][Dutot01] on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the malleable job-scheduling problem speaks about “malleable” jobs that are equivalent to “moidable” jobs because only non-preemptive scheduling is considered. More detail is given in Chapter 6.

2.5 WORKLOAD CHARACTERISTICS

A lot of research on job scheduling is based on the simulation of system workload – a mix of different sizes and types of jobs. Feitelson [1995B] and Leutenegger [1990] show that most scheduling techniques only perform well only on certain kinds of workload models. Therefore, experiments of realistic workload become very important and the corresponding results determine the building of a meaningful workload model.

D. G. Feitelson and B. Nitzberg [Feitelson95B] have traced and analyzed the real parallel workload on a 128-node iPSC/860 located at NASA Ames. They found that most of the system resources were consumed by parallel jobs and most sequential jobs were for system administration. Statistics of experimental data shows that the job submission rate and resource utilization over the weekend are lower than on weekdays; the job submission rate during a peak day is high and the average job size is small; at night, the job submission rate is low but job size and system utilization are high. Finally, the jobs with high degree of parallelism tend to run longer.

Besides the job mix information of workload stated above, speedup (for each job, the ratio of its response time on a loaded system to the response time on a dedicated system) and job efficiency (the ratio of the speedup of this job to the number of processors allotted to it) [Nguyen96][SodanHuang03] information of the workload are also very critical to the job scheduler. If such information is available to the scheduler before scheduling, the overall performance will be greatly improved compared with the situation where such information is unknown. In fact, for simplicity, much research just assumes that such information is already known to the scheduler as a precondition [SodanHuang03]. On the other hand, Nguyen et al. [Nguyen96] have suggested a way to get speedup and job efficiency information during job execution, then provide such information to the job scheduler to make an efficient schedule plan.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Results show that this approach can achieve performance close to those situations where such information is known beforehand.

2.6 APPLICATION INFORMATION

Parallelism in applications might be the most important application information in parallel computing, and characterization of such parallelism is the only way to make your application run in a multiprogrammed parallel system. Characterization of parallelism mainly means decomposing the whole application into several small tasks first, and then defining communications among tasks in order to run them concurrently. Programmers can extract such parallelism explicitly through analysis of the application, or through some high performance parallel compiler such as OpenMP, which can extract parallelism from well-structured loops (e.g. explicitly specified by using OpenMP compiler directives) inside the application during execution.

Parallelism in the application can be represented by several parameters like fraction sequential, the fraction of the overall execution time that cannot be executed in parallel with other parts; average parallelism (avg), the average number of busy processors during an execution of the application when an unlimited number of processors are available [Sevcik89]; and, processor working set (pws), "the number of processors associated with the knee [sic] of the execution-time efficiency profile" [Ghosal91][Chiang94].

S.H. Chiang et al. [Chiang94] have improved several standard static non-preemption scheduling policies such as ASP (adaptive static partitioning), FCFS (first come first served), and SF (shortest job first) by integrating with avg, pws, and limited preemption. K.C. Sevcik [Sevcik89] discovered various rules to extract parallelism in applications and introduced two new parameters: the shape of application ("the proportions of time that the application would use various numbers of processors") and the minimum length ("the total execution time when the application has ample processors allocated"). He concluded that scheduling policies using more parameters would perform better than those using less parameters. Moreover, Julita Corbalan et al. [Corbalan01] show that besides those general parameters discussed above, job malleability (the capability of a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

job to dynamically adapt its parallelism to the number of processors allotted to it) and runtime-measured job performance (the job efficiency calculated based on runtime measurements) can be used to greatly improve the original gang scheduling technique.

However, how to get accurate and up-to-date application characteristics during execution time is a very difficult and challenging task. Therefore, as is the case with workload information, many researchers just assume that the job scheduler knows such application characteristics before scheduling.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3: COSCHEDULING

In order to decrease the overhead of context switching associated with standard time sharing and increase the processor utilization over standard space sharing, J.K. Ousterhout [1982] originally developed the breakthrough coscheduling technique under time sharing. A job is coscheduled if all processes of this job are simultaneously running on distinct processors allotted to them; otherwise this job is called a fragmented job. Normally, the coscheduling algorithm involves two steps: the first step is processor allocation (determine the size for jobs); and then the second step is scheduling. There are two main concrete forms of coscheduling: gang scheduling and dynamic coscheduling.

3.1 GANG SCHEDULING

Gang scheduling [Ousterhout82] has several unique features. For example, processes are grouped into gangs (all processes from the same job are treated as a single gang); all processes in a gang will execute simultaneously on distinct processors; time sharing is used among gangs. J.K. Ousterhout [1982] proposed a Matrix algorithm, which is widely studied by many subsequent researchers to continue improving the performance of the standard gang scheduling technique. Details of the Ousterhout Matrix algorithm are explained in Section 3 of Chapter 6.

The packing scheme of gang scheduling defines the mapping between processes of the same job and the set of processors (might contain one or more distinct processors) allotted to this job. Processes can be mapped to a fixed set of processors or migrated to a different set or even a set of different size from the original set. Efficient packing schemes have been studied by many researchers such as D.G. Feitelson [1997A][1996B].

Gang scheduling is a space-time sharing approach and has advantages such as the avoidance of blocking synchronization problem [Feitelson92], better system utilization and job responsiveness against standard time sharing and space sharing. However, gang scheduling has disadvantages such as poor CPU cache performance, fragmentation, and centralized scheduler

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Gupta91][Feitelson96B]. The fragmentation is the main problem of gang scheduling and also is one of the main goals of the research herein [Feitelson96B][Zhang00], which attempts to improve the performance of standard gang scheduling. For instance, D.G. Feitelson and L. Rudolph [Feitelson90] first addressed the potential efficiency and fairness problem associated with the centralized scheduler by proposing a distributed hierarchy control scheme, and then developed two approaches [Feitelson96B] focusing on solving fragmentation: mapping based on a buddy system, and migration upon each job arrival and termination, which can lead to a significant performance improvement.

3.2 DYNAMIC COSCHEDULING

Dynamic coscheduling [Sobalvarro97][Sobalvarro98] is another main approach of coscheduling, which is suitable for use on a message-passing distributed-memory multiprocessor system and does not require that all processes of the same job to run simultaneously. Therefore, dynamic coscheduling can decrease the coordination effort required by synchronization among all processes of the same job, which is a significant overhead of gang scheduling. This approach is dynamic, flexible, and decentralized; therefore it promises better performance, especially in achieving latency hiding (might get additional speedup by coscheduling one computation intensive job with another one that is communication or I/O intensive).

Demand-based coscheduling [Sobalvarro97] is a concrete approach of dynamic coscheduling. P. Sobalvarro treats the communication among processes as a demand for synchronization; and demand-based coscheduling only guarantees that those processes that communicate with each other will run simultaneously. For instance, if a message arrives at a node and this message is not addressed to the currently running process on that node, then preemption is forced on the running process and the process that the message is addressed to will run next. Figure 3-1 shows a simple example of such an approach. In this diagram, the process 1 of the job 1 (currently running on the node 1) sends a message to the process 2 of the same job, which is not currently running on the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

node 2. Then the process 1 of the job 2 (currently running on the node 2) is preempted and the process 2 of the job 1 will resume running on the node 2.

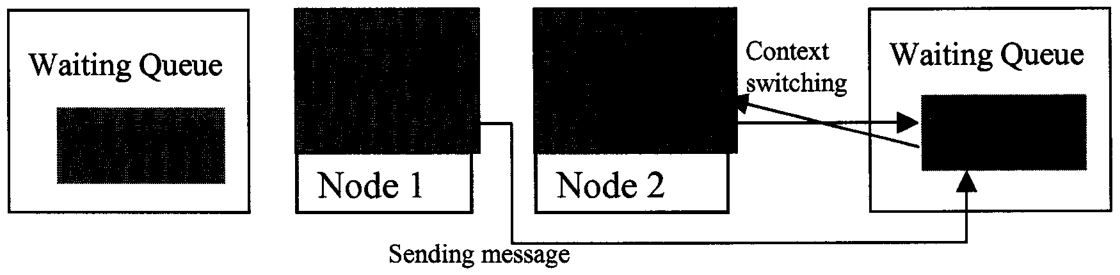


Figure 3-1. An example of dynamic coscheduling

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4: Adaptive scheduling

Adaptive scheduling mainly refers to dynamic processor partitioning under space sharing, i.e., the processor partition can be dynamically changed during job execution. The potential benefits of adaptive scheduling are size adaptation of jobs to the constant changes of workload and user requirement, high resource utilization, and little fragmentation. A number of researchers [Gupta91] [Leutenegger90] [Naik93] have proved that many multiprocessor environments would clearly benefit from adaptive scheduling. As described in Section 4 of Chapter 2, only malleable jobs can dynamically adjust their sizes during execution. In Figure 4-1 we show the size adaptation of a malleable job, J0. The original size of J0 is 4 at time T_1; at time T_2 suppose two new jobs, J1 and J2, are arriving, and then the size of J0 is shrunk to 2 in order to give a chance to execute these two new jobs; at time T_3 suppose both jobs, J1 and J2, are finished, and then the size of J0 is expanded to 6 in order to fully utilize all available processors.

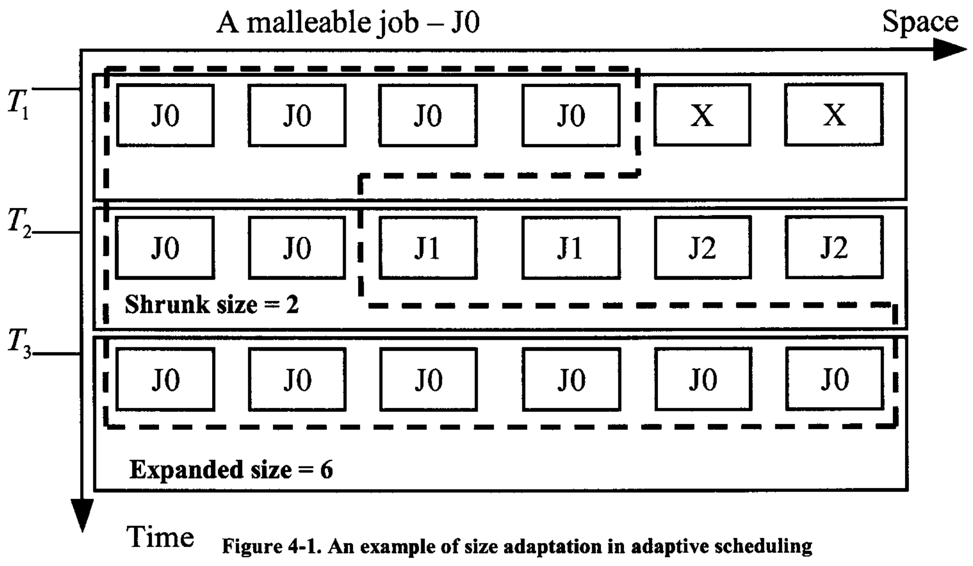


Figure 4-1. An example of size adaptation in adaptive scheduling

In addition, Cathy McCann et al. [1993] concluded that space sharing and dynamic processor partitioning were preferable to time sharing and static processor partitioning. In particular, they proposed an adaptive scheduling policy (DP) by combining space sharing, coordinated preemption, dynamic processor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

partitioning, and a priority scheme. Compared to other general adaptive scheduling policies, this policy had superior performance in a moderately loaded system. They thought that this policy could even be further improved by taking the CPU cache behavior of applications into account, i.e., to improve the CPU cache locality. Moreover, I.H. Kazi et al. have done a lot of research on adaptive scheduling policy design and implementation. Loop-Level Process Control (LLPC) [Kazi00] is a dynamic processor-partitioning technique based on parallelism of well-structured loops in applications, which can dynamically adjust the number of application processes according to the system workload by increasing or decreasing the number of iterations each process can have. K.K. Yue [1998] suggested a way to incorporate such LLPC into the Sun Solaris operating system, and then developed an adaptive scheduling policy [Kazi02], which could dynamically change the number of processors assigned to a task according to not only the system workload, but also the application behavior such as the varying number of loop iterations.

All above-mentioned adaptive scheduling approaches are developed for shared-memory machines. For distributed-memory message-passing machines, C. McCann and J. Zahorjan [McCann94] have proposed two dynamic processor-partitioning policies: equipartition (repartitioning all processors among currently running jobs as equally as possible whenever a new job arrives or an existing job departs) and folding (a new job is allocated on a partition of processors obtained by dividing the largest currently allocated partition in half). On the one hand, Vijay K. Naik et al. [Naik97] have proposed and examined a dynamic processor-partitioning policy by exploiting user-supplied job characteristics like resource requirements.

On the other hand, adaptive scheduling or dynamic processor partitioning policies incur more system overhead [McCann93][Sevcik89], which may lead to a degradation of system performance. Therefore, static scheduling or static processor partitioning and its variations will still be preferred for the sake of simplicity as will overhead avoidance in some environments or systems, where the system overhead resulting from frequent processor reallocations is high.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5: SCOJO

SCOJO provides a local framework in support of grid computing, i.e., to share geographically distributed computational resources. It is our own approach that combines time sharing and batch scheduling. We assume that all jobs require all processor resources in SCOJO; therefore no space sharing is considered.

5.1 GOALS AND SOLUTIONS

Our original intention to develop SCOJO is to meet the following goals:

- Control of multiprogramming level
- Choice between time sharing and exclusive execution
- Flexibility of scheduling cross-site jobs in support of grid scheduling
- Support start time and share reservation for cross-site jobs, which might be scheduled and executed on multiple sites
- Estimation of coscheduling cost
- Maintenance of detailed information about both application and individual site characteristics

In order to meet the above goals, we suggest the following solutions:

- Using effective CPU share by taking the slowdown or speedup information of applications into consideration
- Offering two-level global reservation protocol for cross-site jobs, providing multiple alternate scheduling choices
- Keeping both applications and individual site characteristics in database
- Combining NWS (Network Weather Service) [Wolski99] system to gather detailed dynamic site information, e.g., system load
- Estimating coscheduling cost by providing a performance model
- Applying a priority and aging scheme along with other standard job-scheduling techniques like backfilling

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 STRUCTURE

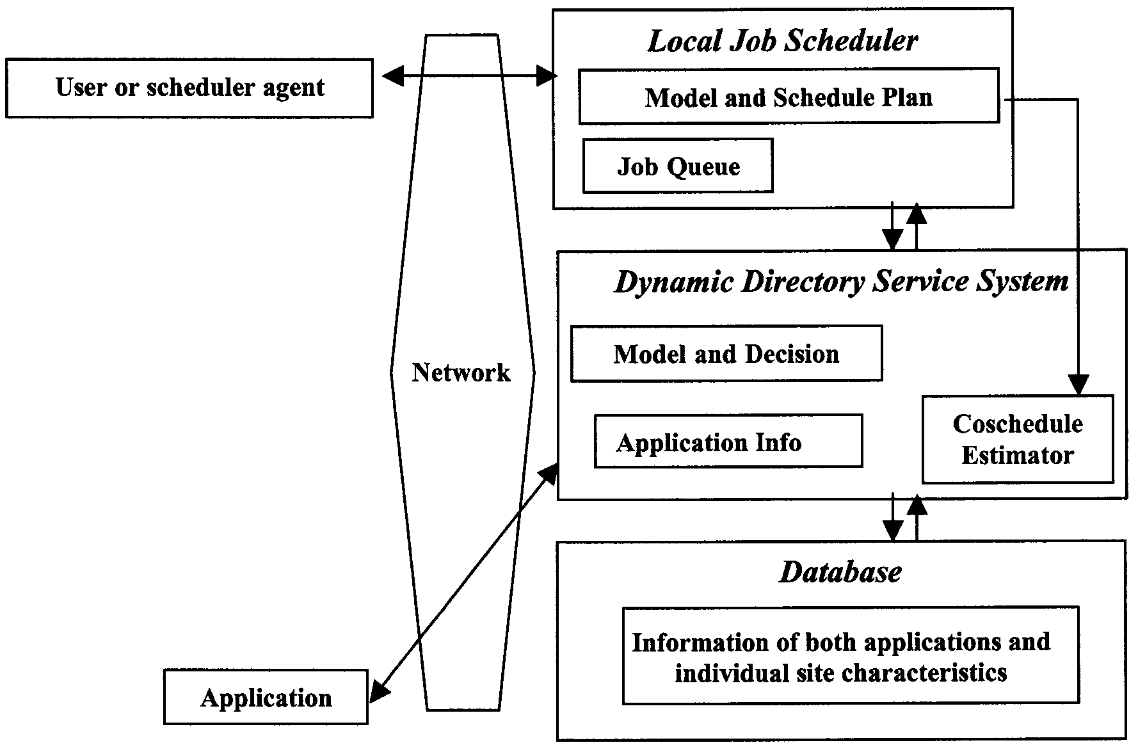


Figure5-1. Overall structure of SCOJO

Figure 5-1 shows the overall SCOJO system structure, which includes three key components: a local batch job scheduler, the dynamic directory service, and the coscheduling estimator. The operation mechanism of SCOJO consists of the following procedures:

1. Remote users contact the SCOJO job scheduler to obtain current site statistics such as load and available resources, which are gathered by the SCOJO dynamic directory service system.
2. If the remote user satisfies the current site statistics and decides to run the corresponding application; then the user needs to provide detailed application characteristics such as runtime and required CPU share to the SCOJO dynamic directory service system. Then, mainly based on application characteristics and potential coscheduling effects among applications, the coscheduling estimator will make a scheduling plan,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which actually is a very complex procedure. A further description is provided in section 5.4.

3. The SCOJO job scheduler will return multiple possible time slots together with available CPU share and potential speedup or slowdown to the remote user who is then asked to reserve or just pick a certain time slot for corresponding application. Reservation means the start time and associated CPU share can be guaranteed for the application, which otherwise will be scheduled to run on the same site without guarantee, i.e., might be executed earlier or later than its originally scheduled time.
4. The SCOJO system will keep application characteristics in a database together with static machine information.
5. In addition, the SCOJO mostly needs to schedule local jobs, which are treated similarly to cross-site jobs except for reservation.

Detailed information about each component is explained in the following sections.

5.3 GRID/LOCAL JOB SCHEDULER

The local batch job scheduler needs to deal with both local jobs and cross-site jobs. The main features of this job scheduler are:

- Enforces priorities on all jobs, mainly according to their runtime classes

  We specify each job into four different runtime classes, which are special (very short), short, medium, and long. Then we assign priorities of 15 for special jobs, 10 for short jobs, 5 for medium jobs, and 0 for long jobs. When a new job comes, it will be placed into a job queue based on its priority, i.e., the job queue is sorted by priorities in a descending order. In this way, we will create more chances for new special and short jobs to avoid them being greatly delayed by medium and long jobs.
- Applies aging scheme

  Priority based queuing and scheduling has benefits such as no delaying special and short jobs, i.e., to improve overall job responsiveness. However, it would introduce a significant starvation problem for medium

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and long jobs if there were many special and short jobs. Therefore, we apply an aging scheme on the priority-based queuing and scheduling. In other words, after a certain amount of time - T_{age}, all waiting jobs in the job queue will be aged by increasing their original priorities into a higher level.

- Guarantees requested CPU share

  For all jobs including both local and cross-site jobs, their requested CPU shares are guaranteed and reserved. However, we do not allow any job to require 100% CPU share, which gives a chance for coscheduling several jobs together, that is, to take advantage of the benefit of dynamic coscheduling like latency hiding. For example, there may be a job that requires a 40% CPU share and is scheduled (with a reservation of 40% CPU share) to run next. If there is no other job scheduled at the same time, this job can take 100% CPU share (i.e. to fully utilize all available resources). However, if later there is a new job coming with equal or higher priority, this implies there is a possibility for coscheduling this new job with the old one. If the coschedule estimator determines that these two jobs can be coscheduled, then the old job will continue running with decreased CPU share down to the reserved one (40%) while the new job is simultaneously running at least with its requested CPU share. More detail is given in Section 5.7.

- Guarantees start times for cross-site jobs

  For cross-site jobs, in addition to CPU share, their start times can also be guaranteed and reserved. Reservation of start times for cross-site jobs is really a major burden for the job scheduler. This task requires the job scheduler not only to apply a general job-scheduling algorithm for both local and cross-site jobs, but also to treat those start-time reserved cross-site jobs separately, which might involve the movement of these jobs in the job queue from their originally scheduled positions to new positions. When such a movement is necessary, several advanced movements for other jobs might be required due to the need for re-estimating coscheduling at new positions. In fact, start time reservation for cross-site

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

jobs results in several problems such as fragmentation, decreased overall job performance, and increased time complexity.

- Applies backfilling technique and allows flexible CPU share assignment

  Coscheduling might introduce fragmentation of CPU share. For instance, in Figure 5-2-1, job A is coscheduled with job B and job B finishes much earlier than job A. Then, after the termination of job B, the CPU share taken by job B can be considered as CPU share fragmentation along the remaining execution of job A. We try to solve this kind of problem by trying to follow two steps:

  1. At first, we try to use backfilling [Feitelson97B], which is originally developed for solving the space fragmentation problem in space sharing (for more detail, see Section 10 of Chapter 6). Basically it is a standard job scheduling technique, which allows a job to be started earlier than its originally scheduled time to fill empty spaces (unutilized processors) if this job does not delay other front jobs in the job queue. Since SCOJO is a pure time sharing approach, we exploit backfilling to fill empty CPU share. We only allow those jobs, which have the same or higher priority as current running job(s), to be the candidates for backfilling. It is important to note here that preventing a delay in other front jobs is not the only requirement for backfilling in SCOJO; we also consider that any backfilled job must be able to coschedule with the current running job(s).

     For example, in Figure 5-2-2, job E, which can be coscheduled with job A, is backfilled after the termination of job B. After the termination of job E, if no more jobs can be backfilled and no CPU share increase on job A, still some CPU share fragmentation will be encountered along with the remaining execution of job A. Then we do the second step – flexible CPU share assignment.
  2. If no more jobs can be taken from the job queue for backfilling, we allow running job(s) to take full utilization of all available resources

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

like CPU share so as to eliminate fragmentation and have a clear time cut for the next job(s). See example in Figure 5-2-3.

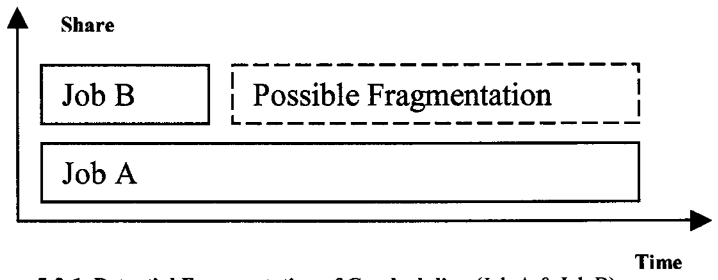


Figure 5-2-1. Potential Fragmentation of Coscheduling (Job A & Job B)

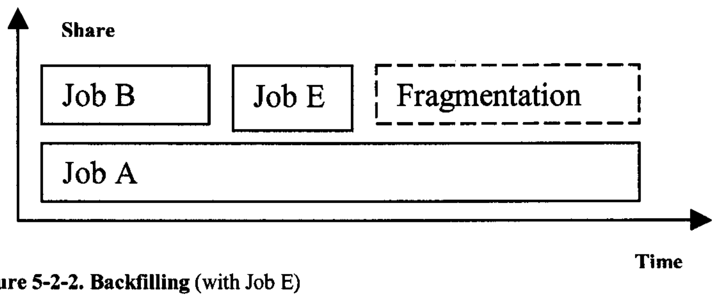


Figure 5-2-2. Backfilling (with Job E)

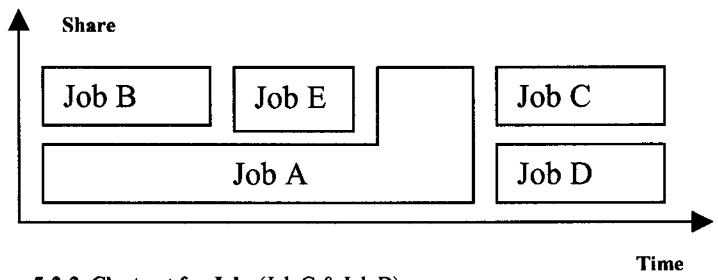


Figure 5-2-3. Cleat cut for Jobs (Job C & Job D)

Figure 5-3 gives a more complicated scheduling example to demonstrate the backfilling and flexibility of the CPU share assignment. The CPU share of job 0 (J0) varies from 40% to 100%, then 50%, and finally 40%. Job 5 (J5) is backfilled (i.e., to be started earlier than job 3 and job 4) after the termination of job 2 (J2); job 6 (J6) is backfilled (i.e., to be started earlier than job 3 and job 4) after the termination of job 5 (J5); and job 9 (J9) and job 8 (J8) are backfilled (i.e., to be started earlier than job 3 and job 4) after the termination of job 7 (J7).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

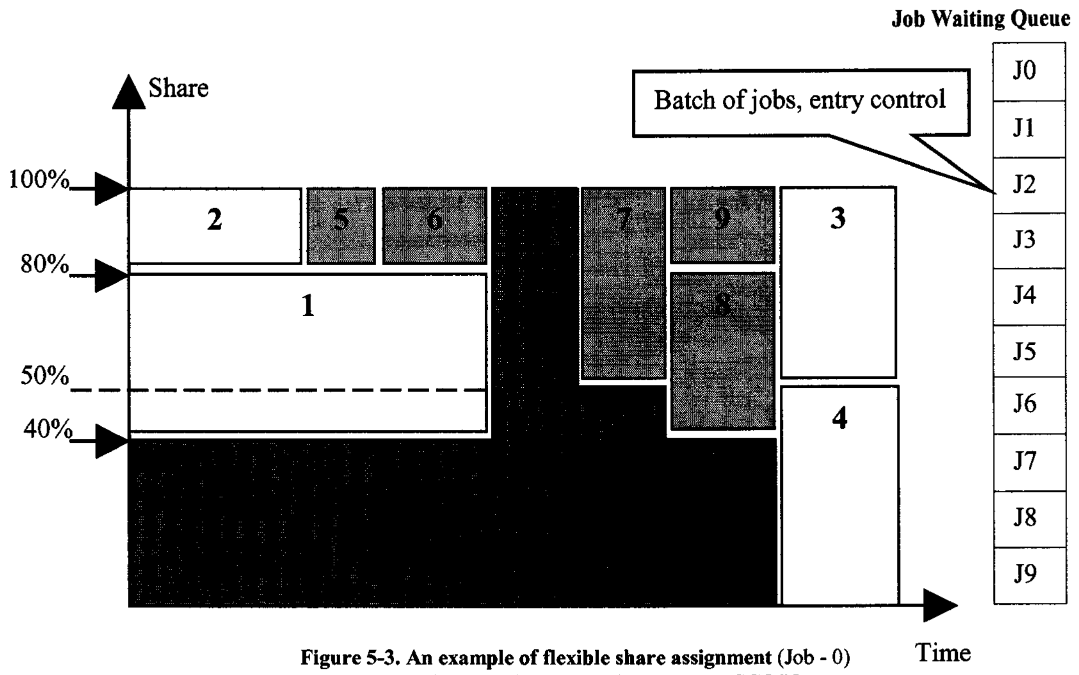


Figure 5-3. An example of flexible share assignment (Job - 0) **and backfilling** (Job - 5,6,7,8,9) **adopted in SCOJO**

- Makes updated schedule plan

The schedule plan specifies the execution order of all waiting jobs and is represented by a list. Each element of this list is a coschedule plan, which specifies either an exclusive execution of a single job (which currently cannot coschedule with others) or a simultaneous execution of several coscheduled jobs. In fact, at each time when an old job terminates or a new job comes, the SCOJO job scheduler will update the current schedule plan into a new schedule plan by considering the possibility of backfilling, characteristics of new jobs, potential coscheduling effects, and the existing start-time reservations of cross-site jobs. Figure 5-4 shows an example of the schedule plan, which consists of four elements. The first element of this plan specifies an exclusive execution of job 0 and the second element specifies that job 4 and job 5 can start together after the termination of job 0. A similar explanation applies for the third and fourth elements.

Schedule Plan – [<coschedule: 0><coschedule: 4,5><coschedule: 1,3,6><coschedule: 2>]

Figure 5-4. A sample schedule plan of SCOJO

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Schedules and runs real applications or simulated processes

  When the schedule plan is complete, the SCOJO job scheduler will schedule jobs to run according to this plan. As seen in Section 5.8, the SCOJO can demonstrate its performance by scheduling real MPI (Message Passing Library) applications or via simulation.

Table 5-1 and Figure 5-5 demonstrate the SCOJO scheduling by a real simple test example. In Table 5-1, ***JOB*** represents the unique job ID given to each job; ***PRI*** is the priority assigned to each job based on its execution time; ***SHARE*** is the CPU share consumed by each job during execution; ***TYPE*** is used to distinguish local and cross-site jobs (1 – a local job, 2 – a cross-site job); ***SUBMIT***, ***RES_TIME***, ***START***, and ***FINISH*** represent the submission time, reserved start time, actual start time, and finish time of each job correspondingly; ***RES*** is the response time.

JOB	PRI	RUNTIME	SHARE	TYPE	SUBMIT	RES_TIME	START	FINISH	RES
1	0	300	40%	1	11:42:39		11:42:40	11:57:07	868
2	5	60	40%	1	11:43:09		11:43:10	11:46:23	193
3	5	60	20%-40%	1	11:43:39		11:43:40	11:50:02	382
4	15	10	20%	1	11:44:09		11:46:23	11:47:29	199
5	15	10	20%	1	11:44:39		11:47:29	11:48:34	234
6	15	10	40%	1	11:45:09		11:50:02	11:50:35	325
7	10	30	20%-40%	1	11:45:39		11:50:51	11:52:42	422
8	15	10	20%-40%	1	11:46:09		11:50:02	11:50:51	281
9	10	30	20%-40%	1	11:46:39		11:51:16	11:55:05	505
10	15	10	20%-40%	1	11:47:09		11:50:35	11:51:16	246
11	0	300	40%	2	11:56:49	11:57:00	11:57:07	12:12:56	966
12	5	60	40%	1	11:57:19		11:57:20	12:00:34	194

Table 5-1. A concrete SCOJO scheduling example

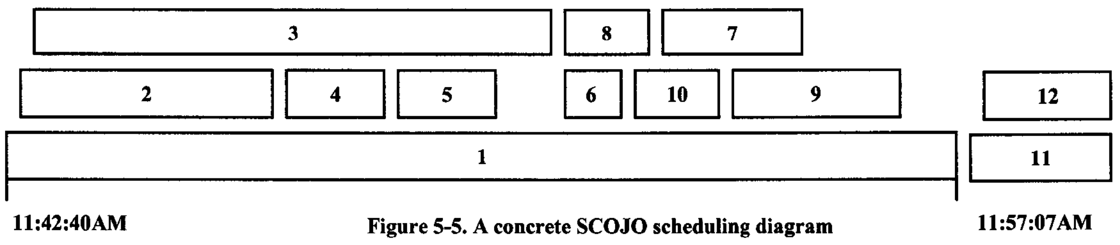


Figure 5-5. A concrete SCOJO scheduling diagram

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4 DYNAMIC DIRECTORY SERVICE

The dynamic directory service is designed to dynamically gather and store application characteristics and machine statistics during runtime.

The following application characteristics are maintained and can be gathered mainly from the application itself (or potential historical data from the database)

- Owner (user)
- Requested CPU share
- Runtime estimation
- Communication pattern and frequency that describes the communication behavior among all processes of the same job
- Other system resource requirements such as memory, I/O, etc.

Where machine statistics are concerned, SCOJO provides an interface to an embedded resource monitoring system like NWS (Network Weather System) [Wolski99], which can periodically monitor the system resources and dynamically forecast the performance that could be delivered over a given time period. The system statistics measured via NWS include

- Available CPU percentage
- Available non-paged memory
- Available disk storage
- TCP-IP performance (latency and bandwidth)

SCOJO will store user information, characteristics of frequently invoked applications, and some static system information like the total number of CPUs, the total amount of memory and the total disk storage into a database.

At last, we need to enforce a certain degree of security into this dynamic directory service system. It means that, on the one hand, we could make use and take advantage of application characteristics and system statistics; but on the other hand, we should not disclose such information to other users or sites. Figure 5-6 represents the structure of the SCOJO dynamic directory service

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system, which keeps two kinds of information - general system information, and application information that consists of two parts: registration-part (static information like requested CPU share) and execution part (dynamic information like runtime measurement).

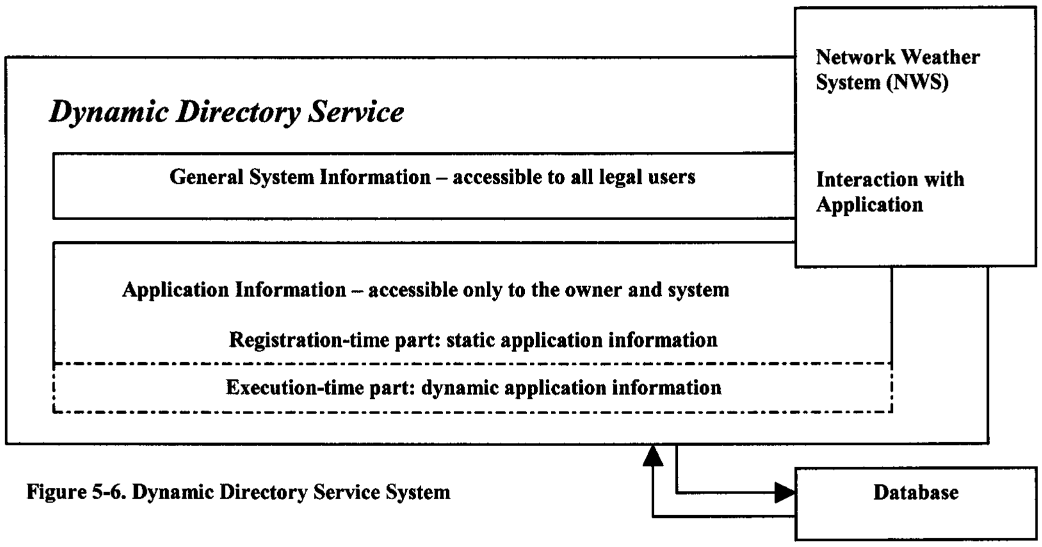

Dynamic Directory Service
General System Information – accessible to all legal users
Application Information – accessible only to the owner and system
Registration-time part: static application information
Execution-time part: dynamic application information
Network Weather System (NWS)
Interaction with Application
Database

Figure 5-6. Dynamic Directory Service System

5.5 COSCHEDULING ESTIMATOR

As described in Chapter 4, benefits such as latency hiding can be obtained from the dynamic coscheduling if I/O or long-distance communication delays are involved. In order to take advantage of such benefits, we estimate the coscheduling effect - the potential speedup or slowdown when coschedules multiple jobs together. Table 5-2 [1] shows different slowdowns measured from coscheduling different application combinations where each application uses 9 CPUs of a Solaris shared-memory machine (SUN Ultra-Enterprise-6500 with 12 processors and 8 GB of SMP memory). The left value represents the slowdown for corresponding row application and the right value represents the slowdown for corresponding column application. The applications used are ***grid*** (heat distribution calculation in a two dimensional matrix, 4-neighbor communication) with different granularities (problem sizes, which are represented by the numbers appearing in parenthesis) and different matrix sizes (e.g., Grid-300 means the

[1] Directly took the experimental results from Dr. Sodan with permission

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

heat distribution calculation in a 300*300 matrix), ***central*** (synthetic, iterative master-slave), ***stream*** (synthetic pipelining, one-way data dependency among processes), and ***random*** (synthetic, random point-to-point with probing). Applications are implemented in MPI. As can be seen, a different combination sometimes has a significant different coscheduling effect than other combinations. For example, if Grid-1200 coschedules with Central, the slowdown is 1.1; however, if Grid-1200 coschedules with Grid-2400, the slowdown is 1.4. For more explanation, see Sodan & Riyadh [2002].

	Grid-300 (1.3msec)	Grid-1200 (27.9msec)	Grid-2400 (116msec)	Central (29.3msec)	Random (4msec)	Stream (3.5msec)
Grid-300	1.2	1.4/1.1	2.2/0.9	0.9/1.6	1/1.3	1.8/0.8
Grid-1200		1.2	1.4/0.8	1.1/1.7	1/1.4	1.3/0.8
Grid-2400			1.1	1/3.1	1.5/2.3	0.8/0.8
Central				1.3	1.3/0.9	2.5/0.8
Random					2	1.8/0.9
Stream						0.8

Table 5-2. Slowdowns in different application combinations

The coscheduling estimator that takes the coscheduling effects among applications into consideration is responsible for:

- Determining whether coscheduling is possible

  If the job scheduler knows the coscheduling effects among applications from the coscheduling estimator, it will make a schedule plan with avoidance of coscheduling two applications together such that there is a significant slowdown on their execution. In fact, the coscheduling estimator can get the estimation of the coscheduling effects through a performance model, which takes the application information and relevant cost factors like $P_{ncosched}(Env)$ (the probability of not being coscheduled) into consideration. This issue has been addressed in depth in previous research of Sodan & Riyadh [2002]. For simplification, in SCOJO the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coscheduling effects provided by the coscheduling estimator are either previous experimental results or assumptions if using mere simulations.

- Calculating the effective CPU share

  For each job, the coscheduling estimator calculates the effective CPU share, which is the multiplication of this job's real requested CPU share and the potential coscheduling effect. Then the job scheduler will reserve and assign the effective CPU share to this job.

  $$SH_{effective} = SH_{request} * Slowdown$$

  The above formula gives the calculation of the effective CPU share. For example, if a job requests 40% of the CPU share ($SH_{request}$= 40%) and the slowdown with another coscheduled application is 1.2 ($Slowdown$= 1.2), then 48% of the CPU share ($SH_{effective}$= 48%) – the effective CPU share will be assigned to this job.

5.6 EXPERIMENT RESULTS

We have done two test cases (Case1 and Case2) to demonstrate the performance gained by the SCOJO scheduling algorithm compared with other standard job scheduling policies like the first-come-first-served policy. Moreover, we have tested the third test case (Case3) to show coscheduling benefits gained by taking coscheduling effects into consideration. All test cases are experimented on a SUN Ultra-Enterprise (6500) machine with 12 processors and 8 GB of SMP memory. The performance metrics used in all test cases are: average response time (AR_{time}), which is an average of the response times of all jobs, and average relative response time (ARR_{time}), which is an average of relative response times of all jobs. For definitions of the response time and the relative response time, see Chapter 1.

For Case1 and Case2, we have compared ***FCFS*** (first come first served), ***Pri*** (mere priority-based scheduling), and ***PriCo*** (priority + coscheduling) with our SCOJO approach, which is ***PriCoB*** (priority + coscheduling + backfilling). And

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the maximum multiprogramming level is set to 2; i.e., at most 2 jobs run at the same time. The equal CPU share assignment is adopted.

- Case1

  In this test case, we use our real sample MPI programs (described in Section 5.5) as jobs and submit them to the SCOJO job scheduler, and the actual coscheduling effects - slowdowns are taken from Table 5-2. Besides, the workload used in this test case is similar (with respect to actual percentages of the different job runtime classes, however, not with respect to actual runtimes) to a real workload measured in [Feitelson95B] on a distributed-memory machine (iPSC/860).

  We have used 36 jobs: 11% of long jobs (***Grid***-2400, 30 min of runtime), 11% of medium jobs (***Random*** and ***Grid***-300, 8-10 min), 16% of short jobs (***Grid***-300 and ***Central***, 3-5 min), and 60% of special jobs (***Grid***-1200 and ***Stream***, 1-1.5 min). Two of the long jobs are cross-site jobs. Job submission is such that the long job is submitted every 40 minutes and the others are equally spread. As can be seen in Figure 5-7 and Figure 5-8, ***PriCoB*** provides the best performance both on AR_{time} and ARR_{time}. For example, the former is reduced from 42.08 minutes to 25.57 minutes and the latter is reduced from 14.47 to 3.22 against ***FCFS***. However, due to the slowdown effects of coscheduling, the total execution time of ***PriCoB*** is increased from 3.57 hours to 4.48 hours compared to ***FCFS***. Moreover, ***PriCo*** performs worse than ***Pri***, which means only coscheduling (even taking coscheduling effects into consideration during scheduling) is not enough (there is potential significant fragmentation left); and then backfilling can play an important role (i.e. to reduce fragmentation).

- Case2

  In this test case, we use full simulation instead of scheduling actual programs, and the workload is similar to the one in [Chiang01] on a DSM machine (Origin 2000). In addition, we assume that all applications have a slowdown of 1.2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We have simulated 40 jobs: 15% of long jobs (5 min of runtime), 20% of medium jobs (1 min), 30% of short jobs (30 sec), and 35% of special jobs (10 sec). The long jobs are submitted every 10 min followed by various mixtures of other jobs. Also can be seen in Figure 5-7 and Figure 5-8, ***PriCoB*** provides the best performance again both on AR_{time} and ARR_{time} for this test case. For example, the former is reduced from 5.57 minutes to 3.27 minutes and the latter is reduced from 12.64 to 1.78 against ***FCFS***. However, the total execution time of ***PriCoB*** is increased from 45 minutes to 55 minutes compared to ***FCFS***.

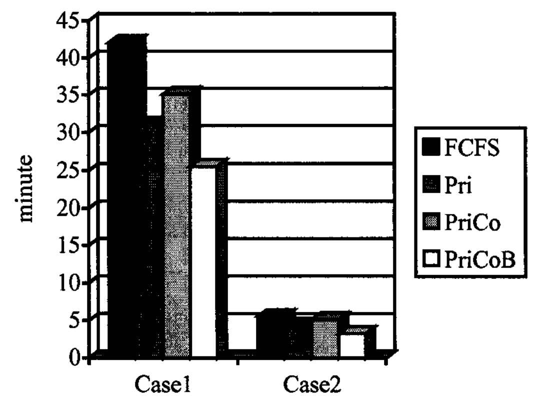


Figure 5-7. Average response time (case 1 & 2)

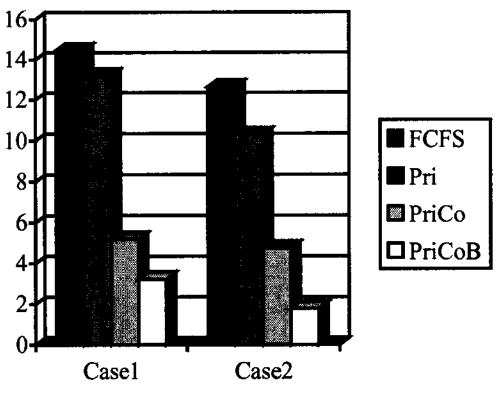


Figure 5-8. Average relative response time (case 1 & 2)

For Case3, we have tested our SCOJO approach (***PriCoB***) under flexible CPU share assignment (40% for each of the first two coscheduled jobs and 20% for the third coscheduled one) and different multiprogramming levels (maximum of 2 and maximum of 3) through simulation. Slowdown is set to 1.2 for all jobs if coscheduling 2 jobs (***C2***), and 1.3 if coscheduling 3 jobs (***C3***). We have simulated 40 jobs: 10% of long jobs (5 min of runtime), 20% of medium jobs (1 min), 20% of short jobs (30 sec), and 50% of special jobs (10 sec). The long jobs were submitted about every 14 min, immediately followed by medium jobs. Short and special jobs were submitted arbitrarily. Figure 5-9 and Figure 5-10 have shown that the coscheduling provides potential benefits (e.g. better performance gained by properly taking coscheduling effects into consideration, even with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

higher programming levels), especially if the percentage of short and special (very short) jobs in the workload is high (i.e., there is more chance for coscheduling or backfilling new short jobs with currently running long or medium jobs). For instance, the average response time of ***C3*** drops from 8.49 minutes to 6.07 minutes and the corresponding average relative response time drops from 8.39 to 3.73 compared to ***C2***.

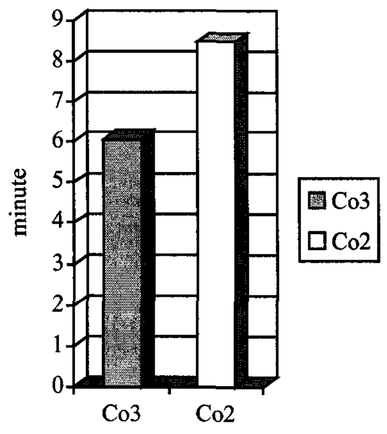


Figure 5-9. Average response time (case 3)

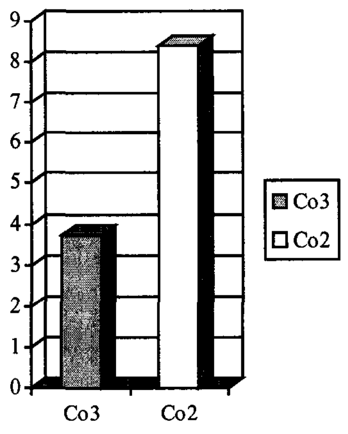


Figure 5-10. Average relative response time (case 3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 6: ADAPTIVE SPACE-TIME SHARING WITH SCOJO ALGORITHM

Based on SCOJO, adaptive SCOJO (Adaptive Space-Time Sharing with SCOJO) incorporates adaptive resource allocation into gang scheduling, which is the more directly suitable approach for combined space-time sharing.

6.1 GOALS AND SOLUTIONS

Adaptive SCOJO has the following goals:

- Adaptive resource allocation

  Adaptive resource allocation mainly means dynamic resource allocation, which dynamically allocates system resources such as processors and memory during job execution, and aims at improving the overall utilization of system resources and providing better overall job performance. In adaptive SCOJO, we only focus on job size adaptation; i.e., we only consider to dynamically changing the number of processors assigned to jobs during job execution. We also assume that the operating system can provide enough support for dynamic processor partitioning.
- Employ realistic workload

  As described in Section 5 of Chapter 2, jobs are classified into three main types: rigid, moldable, and malleable. In order to take advantage of size adaptation, jobs must be either moldable, (i.e., the sizes can be decided at startup), or malleable, (i.e., the sizes can be changed dynamically during execution). Most other related research assumes that all jobs belong to the same type, which is either moldable or malleable. However, this assumption does not reflect the realistic workload, which is mixed with various types of jobs. In addition, we cannot expect that all jobs are malleable – this requires a significant effort from developers on constructing and formulating their programs, which is very difficult and sometimes is impossible. Therefore, adaptive SCOJO considers the realistic workload, which is a mixture of rigid, moldable, and malleable jobs. More precisely, we assume that most jobs are moldable, some are rigid, and some are malleable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Adapt to workload

  The workload keeps changing during job scheduling with the termination of old jobs and arrival of new jobs. Sometimes the workload is high, and sometimes it is low. If we can allocate system resources in a way to adapt to such changing workload, i.e., to release some processors from currently running jobs at high workload in order to schedule new jobs quickly and to give more processors to currently running jobs at low workload in order to take full utilization of all available processors, we might deliver overall better job responsiveness, higher system utilization, and lower multiprogramming level.
- Reduce fragmentation

  Fragmentation in space-time sharing means that not all processors and CPU share can always be fully utilized by jobs, as this results in decreased utilization of system resources. In addition to workload adaptation, adaptive resource allocation can also be used to help solve fragmentation problems, especially on space (unutilized processors).
- Lower multiprogramming level to obtain good performance

  Multiprogramming level (MPL) in space-time sharing means the number of time slices that is applied on a physical processor, i.e., the maximum number of jobs that can be run concurrently on this physical processor in a time sharing manner. A higher MPL normally implies better job responsiveness but severe context-switching overhead. Again, due to the flexibility of dynamic adaptive resource allocation and other applied standard job-scheduling techniques like backfilling, a lower multiprogramming level is expected in adaptive SCOJO to still gain good performance.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In adaptive SCOJO, we provide the following solutions to meet the above goals:

- Combine the adaptive resource allocation with gang scheduling
- Employ size adaptation by taking advantage of both moldable and malleable jobs
- Treat fragmentation reduction and workload adaptation separately in order to maximize the benefits of adaptive resource allocation while minimizing the overhead associated with frequent context switching and intensive resource adaptation
- Exploit other standard job scheduling techniques like priority and aging system, backfilling or EASY backfilling, etc.
- Provide a clear criterion to determine when, to what degree and how to do adaptive resource allocation
- Take application characteristics like runtime estimation and processor working set into consideration

6.2 SELECTED RELATED WORKS

Almost all work on adaptive scheduling is mere space sharing only. Furthermore, most adaptive approaches only exploit moldable applications and aim at minimizing the makespan while focusing on the provision of tight worst-case bounds [Turek1992][Dutot2001].

Naik [1997] presents one of few approaches that exploit malleable applications to adapt system resources assigned to jobs to varying workload. Resource adaptation is only considered for medium and long running jobs; and a certain reconfiguration time interval is applied to avoid configuration thrashing. EQUI partitioning (i.e. evenly partitioning resources among jobs) is applied to adjust the jobs' sizes at each time of workload adaptation when the workload is high; otherwise the jobs' requested sizes are considered. There is another principal approach to determine how to adjust the jobs' sizes: efficiency-based partitioning, which uses the concept of the processor working set [Ghosal91] to reflect the applications' different speedup curves.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There are several approaches [Zhang01][Zhang00][Frachtenberg03] proposed to improve the performance of the traditional gang scheduling [Ousterhout82] technique. For example, Zhang [2001] applies backfilling and migration and Frachtenberg [2003] applies EASY backfilling to solve the fragmentation problem associated with gang scheduling.

There is little work combining gang scheduling with adaptive resource allocation. Corbalan [2001] presents two approaches to do so. The first approach adapts the number of processors allotted to each job for its optimal efficiency calculated based on runtime measurements. The second approach compresses the sizes of both currently running jobs and any other non-started previously scheduled jobs, and then allocates available processors to new jobs. However, certain limitation and drawbacks exist in this work. For example, all jobs are assumed to be malleable; no clear criterion is provided to decide when to stop size adaptation; and no other standard job scheduling techniques are combined.

To summarize, the main contribution of this thesis is:

- Apply to realistic workloads (i.e., mixture of all types of jobs)
- Combine adaptive resource allocation with gang scheduling (space-time sharing) on clusters
    - Employ adaptive resource allocation for both fragmentation reduction and workload adaptation
    - Trade space vs. time based on a clear model (including overhead)
    - Apply other standard job-scheduling techniques like backfilling or EASY backfilling, etc.

6.3 OUSTERHOUT MATRIX

As mentioned in Chapter 3, J.K. Ousterhout [1982] developed the original coscheduling technique and proposed a two-dimensional Ousterhout Matrix, which was used to visually represent the job-scheduling problem of a parallel machine in space-time sharing. In the Ousterhout Matrix, rows represent the number of time slices used or the multiprogramming level, that is, the number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

jobs coscheduled together on a physical processor, and columns represent the total number of processors that a parallel machine has. We can view each row as a virtual parallel machine, which has the same number of processors as the real physical machine. Then the job-scheduling problem of space-time sharing is kind of attempting to fill such Matrix with parallel jobs while keeping the Matrix as full as possible to reduce fragmentation and enhance the system utilization. More precisely, Ousterhout describes a two-step scheduling strategy for Matrix filling:

- Processor allocation

  Every parallel job requires certain number of processors and on each assigned processor there is a process associated with this job. When scheduling such a job, the job scheduler first tries to fill this job into the Matrix at the first row if there is enough unused processors left; otherwise, try the second row, and so on until a row is found that can accommodate all processes of this job.

- Scheduling

  After filling the Matrix, scheduling all processes inside this Matrix is time sharing enforced, which means at time slice 0, each process of row 0 is executed on the corresponding processor. After a certain time period, at time slice 1, each process of row 1 is executed on the corresponding processor, and so on until the last row. Then, return to time slice 0 and repeat.

Figure 6-1 gives an example of the Ousterhout Matrix representation of a parallel machine, which consists of 16 physical processors and applies the multiprogramming level of 5.

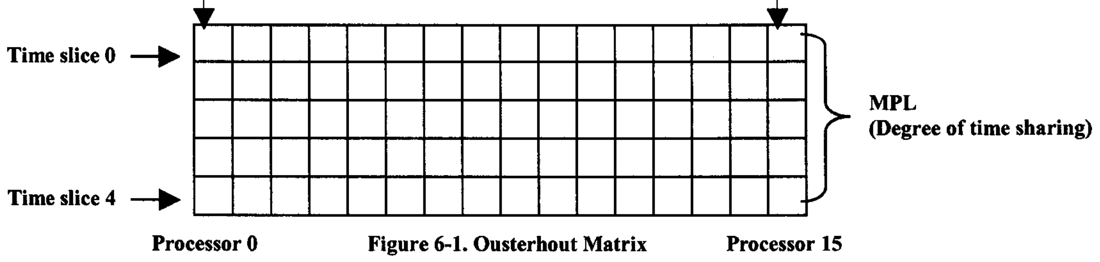


Figure 6-1. Ousterhout Matrix

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As mentioned in Chapter 3, gang scheduling guarantees all processes of a job are running or suspending simultaneously in a time-shared manner; i.e., all processes of the same job are synchronous. A simple demonstration of gang scheduling is described in Figure 6-2. Suppose there is a parallel machine of 10 physical processors, where Job 0 (J0) contains 8 processes that require 8 processors, and Job 1 (J1) contains 4 processes that require 4 processors. After allotting J0 at time slice 0 (T0) on processors from P0 to p7, instead of continuously assigning two left unused processors (P8 and P9) at T0 and two front processors (P0 and P1) at time slice 1 (T1) to J1 in Choice A, Choice B is the correct processor allocation in traditional gang scheduling that assigns four processors from P0 to P3 at T1 to J1.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
T0	J0	J0	J0	J0	J0	J0	J0	J0	***J1***	***J1***
T1	***J1***	***J1***								

Choice A

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
T0	J0	J0	J0	J0	J0	J0	J0	J0		
T1	***J1***	***J1***	***J1***	***J1***						

Choice B

Figure 6-2. Simple demonstration of Gang Scheduling

6.3.1 MULTIPROGRAMMING LEVEL

As mentioned before, the Multiprogramming Level (**MPL**) of Ousterhout Matrix refers to the degree of the time sharing, i.e., the total number of time slices applied in gang scheduling. A MPL of 1 implies pure space sharing.

In general, the MPL determines the number of jobs that can run concurrently and is limited by the system resources like memory. Higher MPL normally means less job waiting time (the time period between the job submission time and the job startup time); i.e., jobs can be scheduled sooner than that of lower MPL.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

However, higher MPL also means more frequent context switching, i.e., more context switching overhead, and higher memory pressure.

Moreira [1998] found that a multiprogramming level of 5 could provide almost the same responsiveness as an infinitely high level applied in gang scheduling. Therefore, we use maximum MPL of 5 in adaptive SCOJO.

6.3.2 CONTEXT SWITCHING OVERHEAD

In time sharing, when the time slice expires after certain time interval the scheduler of the operating system needs to stop and exchange the running process at the current time slice for the process at next time slice per processor. This procedure is called context switching. The cost associated with it mainly refers to the processor time needed on such operation. The more frequent the context switches, the more processor time is needed (more context switching overhead).

SCore-D [Ishikawa99] is a well-known operating system for workstation and PC clusters. Ishikawa99 et al. conclude that the job scheduler of SCore-D can get less than 10% overhead for 40 millisecond time intervals (the time period between two time slices in time sharing) and there are few other research papers addressing this issue. Therefore, we take the worst 10% of the time interval as the context-switching cost in adaptive SCOJO.

6.4 ADAPTIVE SCOJO SCHEDULING ALGORITHM

In SCOJO, we have tested the performance of our approach via scheduling real parallel applications. However, it is limited to the size of test cases. Therefore, in order to comprehensively test various heavy loads of realistic job mixes that consist of thousands of jobs and various combinations of different scheduling strategies, we build our test through a discrete-event simulation in adaptive SCOJO.

We treat every new job arrival or every old job departure as an event, which requires the job scheduler re-compute and re-update the scheduling Matrix. Then the job scheduler will schedule jobs according to this updated Matrix.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The corresponding algorithm for re-computing and re-updating scheduling Matrix is described in Figure 6-3, which consists of 9 steps. Detailed explanation of each step and overall time complexity analysis are provided in the following sections.

```
//Step1: Sum up to this event, the fragmentation and the context switch overhead encountered.
        sumFragmentationAndOverhead();
//Step2: If this is a job departure event, free corresponding processors.
        for i = 0 to number of processors assigned to this departure job
          add the corresponding freed processor ID into the emptySlots at corresponding time slice
//Step3: Classify the current workload.
        workloadStatus=classifyWorkload();
//Step4: Determine the job target size according to the current workload.
        if(workloadStatus == high)
          jobTargetSize=(currentJobSize-minJobSize)/2+minJobSize;
        else if(workloadStatus == low)
          jobTargetSize=(maxJobSize-currentJobSize)/2+currentJobSize;
        else //workloadStatus == normal
          jobTargetSize=optimalJobSize;
//Step5: if the current workload is high, shrink running malleable jobs to the target size. Otherwise
//      expand running malleable jobs to the target size. This is called workload adaptation.
        workloadAdaptation()
        {
          if(workloadStatus == high) shrinkMalleableJobs();
          if(workloadStatus == low) expandMalleableJobs();
        }
//Step6: Populate Matrix with new jobs taken from the job waiting queue, using jobTargetSize.
        populateMatrix()
        {
          while a new job with its target size fits into the Matrix
            allocate this new job;
        }
//Step7: Reduce fragmentation by backfilling or EASY backfilling new jobs from the job waiting queue,
//      which could be scheduled earlier than their original scheduled time.
        backfilling() or easyBackfilling();
//Step8: Continue to reduce fragmentation by taking advantage of new started moldable and new
//      started medium malleable jobs.
        eliminateFragmentaion()
        {
          for all new started moldable jobs and new started medium malleable jobs
            reduce fragmentation per time slice by expanding the sizes of those corresponding jobs
        }
//Step9: Update the time slice and job execution time correspondingly.
        timeSliceChange();
        executionTimeChange();
```

Figure 6-3. Adaptive SCOJO Scheduling Algorithm

6.5 SCHEDULING EVENT

Although each job arrival or departure event can happen at any time, however the job scheduler only considers interruptions at the beginning of the next time interval, which is equal to or later than the actual event time; i.e., the job

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

scheduler can not be interrupted between two time slices. Figure 6-4 helps to illustrate this.

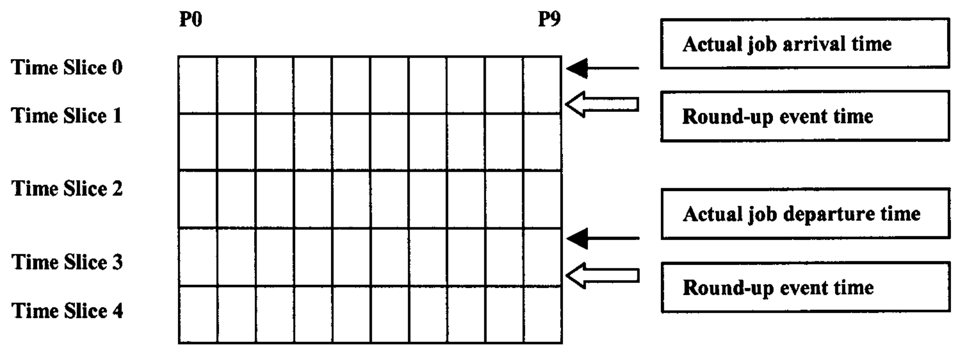


Figure 6-4. Scheduling event

6.6 APPLICATION INFORMATION AND MODELING

As mentioned in Section 7 of Chapter 2, accurate application information really can help the job scheduler to make an efficient schedule plan, to improve job performance and enhance system utilization. However, to acquire such information about applications during execution time is difficult. Most research assumes this information can be provided by the application itself, or can be estimated during job runtime. In adaptive SCOJO, we assume that

- The following general information is provided by every application:
  - TYPE – local job or cross-site job
  - PRIORITY – details in the next section
  - RUNTIME – execution time estimation

    Although we assume accurate estimation of execution time of jobs, our adaptive scheduling can deal with wrong or incorrect runtime estimation as well because in adaptive SCOJO we ignore the reservation for cross-site jobs.
  - FLEXIBILITY – the flexibility of job, i.e. rigid, moldable, or malleable
  - PROCESSOR WORKING SET – ***pws***

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

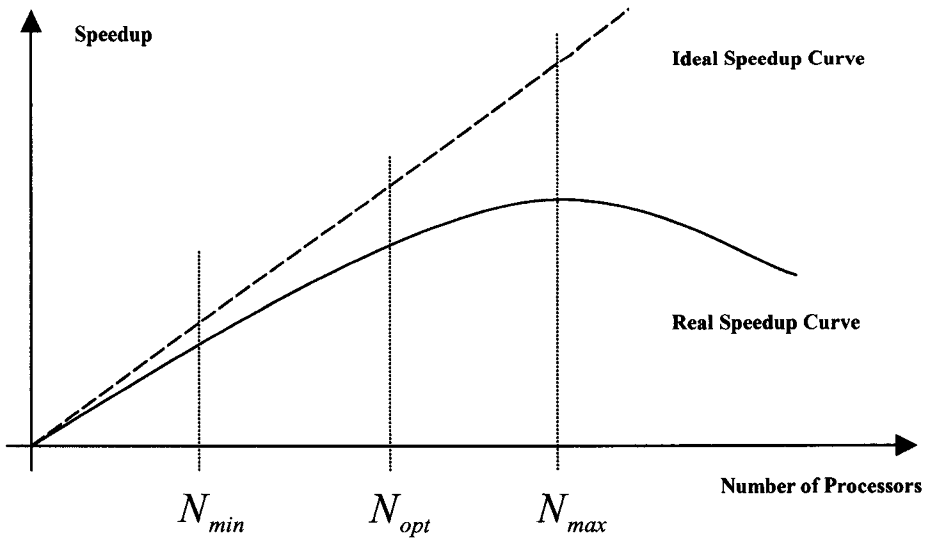


Figure 6-5. Speedup Curve

Figure 6-5 gives an example of the speedup curve of an application. Ideally, if an application runs T_1 time to finish on single processor, it will need $T_N = \frac{T_1}{N}$ time to finish on N processors; i.e., the ideal speedup **S** is defined as $\frac{T_1}{T_N}$, which is N. Therefore, the dashed line of an ideal speedup curve has linear shape as shown in Figure 6-5. However, mainly due to the cost of communication and synchronization among all processes or processes of the same application, the typical speedup curve only has a convex and sub-linear shape like the solid line of the real speedup curve in Figure 6-5; i.e., the speedup S can not reach N when the application runs on N processors.

More precisely, in Figure 6-5 the sampled real speedup curve has the following features:

1. When the corresponding application runs on fewer processors (less than N_{min}), its real speedup curve is close to the ideal speedup curve, and can be thought of as linear.
2. When the corresponding application runs on an increased number of processors (between N_{min} and N_{max}), its real speedup curve

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

becomes flattened; i.e., the real speedup does not increase linearly with the number of processors increased.

3. From a certain number of processors (greater than N_{max}) on, the real speedup curve of the corresponding application drops, i.e., the real speedup does not increase anymore.

The job efficiency **E** is the ratio of the speedup **S** to the number of processors **N** allotted to the job ($E = \frac{S}{N}$), which in turn reflects the utilization of machine (for example, ideally S = N, then E = 1 or 100%; i.e., machine is fully efficiently utilized). Then, the processor working set – ***pws*** is defined as the set of optimal number of processors on which the job can gain best efficiency.

pws = {N_{used} | with (T_{Nused}/E) is minimal}

Where, N_{used} is the number of processors (size) used by the job, T_{Nused} is the execution time (runtime) needed on size N_{used} for the job, and **E** is the job efficiency on size N_{used}.

In addition, in adaptive SCOJO we use N_{min} to represent the number of processors from which the increase of real speedup becomes flattened, where S_{min} is the corresponding speedup, N_{opt} represents the processor working set, S_{opt} represents the speedup at N_{opt}, N_{max} represents the number of processors from which on the real speedup drops and S_{max} is the corresponding speedup.

- The speedup curve of each application has been estimated according to the following application model
  - We assume that the speedup S_{min} is 80% of the ideal speedup at N_{min}, i.e. S_{min}=0.8* N_{min}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- We assume that the speedup S_{opt} is 65% of the ideal speedup at N_{opt}, i.e. S_{opt}=0.65*N_{opt}.
- We assume that the speedup S_{max} is 50% of the ideal speedup at N_{max}, i.e. S_{max}=0.50*N_{max}.
- We assume a linear approximation between N_{min} and N_{opt}, and a linear approximation between N_{opt} and N_{max} as shown in Figure 6-6. For example, for any N in the processor interval (N_{min}, N_{opt}) we can have such approximation on the speedup curve: 0.8*N_{min}+(0.65*N_{opt}-0.8*N_{min})/(N_{opt}-N_{min})*(N-N_{min})

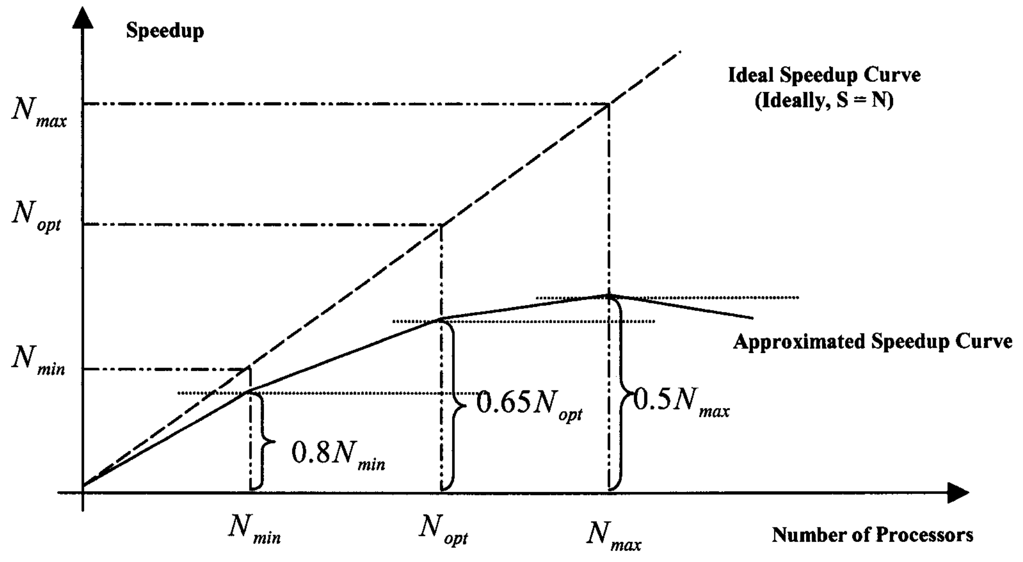


Figure 6-6. Speedup Curve Approximation

6.7 PRIORITY AND FLEXIBILTY ASSIGNMENT

The priority and aging scheme in adaptive SCOJO is the same as that in SCOJO (see Section 4 of Chapter 5) except that in adaptive SCOJO we classify jobs into three classes instead of four based on their runtimes, i.e., short job with priority of 10, medium job with priority of 5, and long jobs with priority of 0.

With respect to flexibility, we permit

- A rigid job can belong to any runtime class; i.e., any of short, medium, or long jobs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- A moldable job can be a short or a medium job

  The reason that we do not permit long jobs being moldable is to avoid the disadvantages of scheduling a small number of processors to long jobs at the startup while the system workload is heavy. This will force long jobs running to complete with squeezed sizes; i.e., the response time for long jobs will be greatly increased and the system utilization will be possibly decreased especially when the system workload becomes light, later.
- A malleable job can be either a medium or a long job

  Since a short job (short execution time) is supposed to finish within a small amount of time, it is not worthy making effort to program the corresponding application as malleable. For this reason, we only permit medium and long jobs to be malleable jobs.

6.8 WORKLOAD MODELING AND GENERATION

We are going to generate two different workloads: one is purely synthetic; another is a loose copy of a real workload described in [Chiang01] by differentiating the runtimes of jobs (i.e., we model the same percentage of each runtime class, however, only permit the longest job runtime to be 30 hours instead of several hundred hours). Detailed information of these two workloads is shown in Table 7-1 of Chapter 7. No matter what kind of workload we are modeling, the following general features apply:

- Realistic job mix

  To reflect a realistic job mix, the modeled workload consists of lots of moldable jobs, some rigid jobs, and some malleable jobs. Different job mixes (wherein the contributed percentage of each job runtime class varies in a small range) have been used and tested on different workloads. In addition, the job runtime class (short, medium, long) and the job flexibility (rigid, moldable, malleable) are totally randomly generated; i.e., there is no forced or sequenced order on the combination or generation of the whole workload.
- Realistic processor size requirement

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We set a limitation on the optimal processor size (N_{opt}) interval for different job runtime classes; e.g., the N_{opt} interval of medium jobs is [4,24] and the N_{opt} interval of long jobs is [8,32]. The N_{opt} for each job will be randomly generated based on this interval, then the corresponding N_{min} is set to equal to N_{opt}/2 (or 1 which is greater), and the corresponding N_{max} is set to equal to N_{opt}*2 (or the total number of physical processors which is less). By doing this, actually we allow the processor size requirement for different kinds of jobs to be overlapped with each other; i.e., short jobs can require more processors than medium, even long jobs, and vice versa.

With respect to the workload generation, we intend to create a randomly generated heavy workload in order to comprehensively test the performance of our adaptive SCOJO scheduler, wherein an improved scheduling technique has been adopted over the standard gang scheduling strategy. However, to avoid overload at the beginning, we generate jobs one after another based on an average of inter-arrival time ($T_{inter-arrival}$); i.e., the next job can be generated randomly at any time in this time range [1, 2*$T_{inter-arrival}$], in which 1 represents 1 second (the smallest time unit we have used in this thesis). $T_{inter-arrival}$ is set by the following formula

$$T_{inter-arrival} = (\sum AVG_{weighted-size} * AVG_{weighted-runtime})/N_{nodes}$$

where $\sum$ means we calculate $AVG_{weighted-size} * AVG_{weighted-runtime}$ per job runtime class, i.e., short jobs, medium jobs, and long jobs are calculated separately, and then take the sum. $AVG_{weighted-size}$ is the average weighted processor size and $AVG_{weighted-runtime}$ is the average weighted job runtime. For instance, suppose every medium job's runtime is in this range (1min, 30min]; then the average runtime of medium jobs is 15min. In addition, if medium jobs count for 35% of all jobs, then we take the weight factor, which is 0.35, into consideration for medium jobs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore, for medium jobs, $AVG_{weighted-runtime}$=15min*0.35. A similar calculation applies for $AVG_{weighted-size}$. N_{nodes} represents the total number of physical processors (nodes).

6.9 WORKLOAD CLASSIFICATION AND ADAPTATION

The workload classification and adaptation is related to the Step3, Step4 and Step5 of our adaptive scheduling algorithm. We discuss them separately in this section.

6.9.1 WORKLOAD CLASSIFICATION

This is the Step3 of our adaptive scheduling algorithm. Workload classification aims to check the current workload status of the system; e.g., whether the current workload is high or low. We then use the current workload status to direct further actions fired in the following steps so that we can adapt our scheduling to the frequent changes of the system workload in order to improve overall job performance and system utilization.

We classify the system workload into three statuses - low, normal, and high according to the algorithm described in Figure 6-7.

```
// Step1: Calculate the $Nodes_{needed}$
            $Nodes_{needed} = (N_{jobsR} * AvgSize_{RwVShort}) + (N_{jobsW} * AvgSize_{WwVShort})$
//Step2: If the following condition satisfies, then consider the current system
//       workload is low
            $Nodes_{needed} + N_{malleable} * AvgSize_{increase} < N_{nodes}$
//Step3: If the following condition satisfies, then consider the current system
//       workload is high
            $Nodes_{needed} > N_{nodes}$*MPL
//Step4: If both above conditions fail, then consider the current system
//       workload is normal
```

Figure 6-7. The workload classification algorithm

The workload classification algorithm includes the following 4 steps:

1. Estimate the number of nodes (processors) required during the next scheduling interval; this gives the Step1 calculation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$Nodes_{needed} = (N_{jobsR} * AvgSize_{RwVShort}) + (N_{jobsW} * AvgSize_{WwVShort})$$

$Nodes_{needed}$ is the estimation of the total number of nodes required during the next scheduling interval for both running and waiting jobs. $N_{jobR} * AvgSize_{RwVShort}$ estimates the number of nodes needed by the currently running jobs (using ***R*** to represent) and $N_{jobsW} * AvgSize_{WwVShort}$ estimates the number of nodes needed by the waiting jobs (using ***W*** to represent). However, both estimations exclude very short jobs (***VShort***) because they are supposed to complete very quickly e.g. runtime is less than the reconfiguration time interval, $T_{reconfig}$ ($T_{reconfig}$ is explained in Section 6.9.3), and therefore do not contribute too much to the system workload. N_{jobs} represents the number of jobs, which are either currently running (***R***) or waiting (***W***). $AvgSize_{wVShort}$ represents the average size (number of processors) request for both currently running jobs (***R***) and waiting jobs (***W***) without very short ones (***wVShort*** – ***w***ithout ***V***ery ***Short***).

2. If all jobs that are currently in the system (either running or waiting) can be scheduled to run during the next time interval without multiprogramming (the total number of physical processors can accommodate the space request of all jobs) and there is still sufficient empty space (unused processors) left to expand all malleable jobs, the current workload can be regarded as low.

$$Nodes_{needed} + N_{malleable} * AvgSize_{increase} < N_{nodes}$$

The above formula is the Step2 of the workload classification algorithm. $N_{malleable} * AvgSize_{increase}$ gives the nodes request from all malleable (either running or waiting) jobs. $N_{malleable}$ represents the total number of malleable jobs and $AvgSize_{increase}$ is the average size increase when expanding these malleable jobs to reduce fragmentation and adapt to the workload. More detail is provided in Section 6.9.3. Again, N_{nodes} represents the total number of physical processors (nodes).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. If all jobs that are currently in the system (either running or waiting) cannot be scheduled to run during the next time interval even with the maximum multiprogramming level, the current workload can be considered as high.

   $$Nodes_{needed} > N_{nodes} * \mathrm{MPL}$$

   The above formula corresponds to the Step3 of the workload classification algorithm. It is clear by itself. Again, MPL represents the maximum multiprogramming level (we use 5 in our approach, which is described in Section 6.3.1)

4. Otherwise, the current workload is classified as normal.

   If both conditions for checking high workload and low workload fail, then we consider the current workload status to be normal, i.e., there is no need to do workload adaptation in the following steps.

6.9.2 DETERMINE THE JOB TARGET SIZE

After classifying the current workload status, the next step is to determine the job target size for the new job, which are taken from the waiting job queue to attempt to be scheduled next by the job scheduler. This is the Step4 of our adaptive scheduling algorithm. As already described in previous section (Section 6.6), we assume the speedup curve of each job is known based on a simplified application model; i.e., we can know the processor size interval $[N_{min}, N_{max}]$ per job according to the N_{opt} that is provided by each job. We also know that moldable jobs can determine their processor sizes at startup (then keep these sizes fixed afterwards) and malleable jobs can change their sizes dynamically during execution time. This implies that both moldable and malleable jobs have the ability of size adaptation. Then we try to assign the number of processors to each new moldable or new malleable job according to its optimal size request - N_{opt} when the workload is normal, and expand or shrink N_{opt} when the workload is low or high; i.e., the job target size is defined different from any of N_{min}, N_{opt}, and N_{max} as in the following:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. When the current workload is normal

   The job target size is set to equal to N_{opt} for any kind of job (rigid, moldable, or malleable)
2. When the current workload is low

   The job target size for new moldable or malleable job is set to equal to the middle of N_{opt} and N_{max}, which is $(N_{max} - N_{opt})/2 + N_{opt}$. The new rigid job has no choice but N_{opt}.
3. When the current workload is high

   The job target size for new moldable or malleable job is set to equal to the middle of N_{min} and N_{opt}, which is $(N_{opt} - N_{min})/2 + N_{min}$. Again, the new rigid job has no choice but N_{opt}.

The above procedure is used to determine the target sizes of new jobs that are going to be scheduled to run in next time interval by fitting them into the Matrix with their target sizes. The main reasons that we specify the size of new moldable or malleable job in this way are: firstly, we try to take advantage of its ability of size adaptation to workload; secondly, we leave space for further size adaptation (take the middle instead of N_{min} or N_{max}). For old jobs already scheduled and currently running, their further size adaptation to the system workload is discussed in the next section.

6.9.3 WORKLOAD ADAPTATION

This is the Step5 of our adaptive scheduling algorithm, which mainly concerns the possibility of further size adaptation to the system workload for currently scheduled and running jobs. However, not all kinds of jobs can do such size adaptation after they have been scheduled – only malleable jobs have such an advantage. Therefore, this step actually describes dynamically changing the sizes (number of processors) of malleable jobs during their execution for adapting to the changes of system workload, in order to improve overall job performance and enhance the system utilization. We perform such size adaptation to the workload by the following way:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- If the current workload is low, then the size of running malleable jobs will be expanded to (N_{max}-$N_{current}$)/2+$N_{current}$. $N_{current}$ is the current size (currently assigned number of processors) of running malleable jobs. If the workload remains stably low, this size adaptation will lead the sizes of running malleable jobs expand to N_{max} eventually. The main reason why we do not expand the size to N_{max} immediately is to
    1. Leave space for other jobs, especially a chance for new jobs so that they can be scheduled to run earlier, i.e., more fair
    2. Leave space for further size adaptation, i.e., more flexible
- If the current workload is high, then the size of running malleable jobs will be shrunk to ($N_{current}$-N_{min})/2+N_{min}. If the workload remains stably high, this size adaptation will lead the sizes of running malleable jobs shrink to N_{min} eventually. The reason that we do not shrink the size to N_{min} immediately is the same as for expanding, which is explained above
- If the current workload is normal, there is no size adaptation to workload on running malleable jobs no matter what their current sizes ($N_{current}$) are

6.9.4 RECONFIGURATION INTERVAL AND ADAPTATION OVERHEAD

The above subsection describes how to do size adaptation to the workload for running malleable jobs. This subsection will talk about how often we do such adaptation and how we deal with the overhead associated with it.

On the one hand, the main overhead of size adaptation is that it costs some time and effort to reconfigure the program (repartitioning data among changed processors, and so on). On the other hand, frequent reconfiguration of a program might result in configuration thrashing (thrashing memory too much). Therefore, we only allow size adaptation in certain time intervals - $T_{reconfigure}$ - to make sure that the benefit of size adaptation overweighs the overhead associated with it. In addition, we model the adaptation overhead by the following formula

$$N_{nodes-difference} * O_{reconfigure}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where $N_{nodes-difference}$ represents the actual size change, which is the absolute value of the difference between the size before adaptation and the size after adaptation, and $O_{reconfigure}$ gives a fixed overhead per node.

6.10 GANG-SCHEDULING MATRIX FILLING

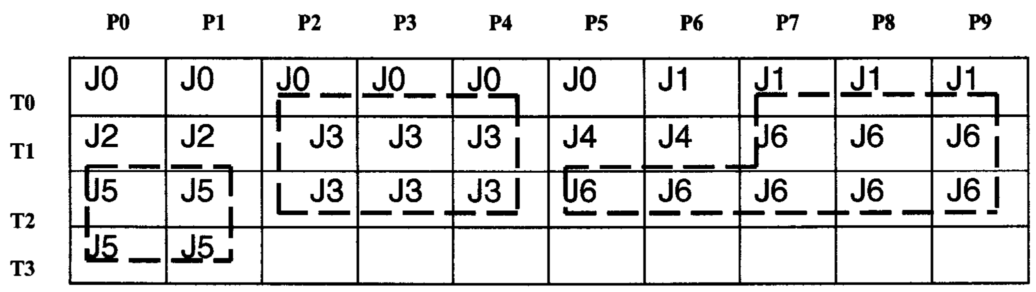
Gang-scheduling Matrix filling is Step6 of our adaptive scheduling algorithm. In this step, the job scheduler tries to bring and fit new jobs with their target sizes (determined according to current workload status) into the gang matrix. The following main features applies:

- Focus on allocation of CPU resources

  We focus on the allocation of CPU resources while ignoring the allocation of other resources such as the memory, I/O devices, etc.
- We do not consider flexible time share assignment; instead, equal time slices are used. Figure 6.8 shows the equal time slice assignment for all jobs, which is Choice A; and flexible time share assignment for Job3, Job5, and Job6, which is Choice B.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
T0	J0	J0	J0	J0	J0	J0	J1	J1	J1	J1
T1	J2	J2	J3	J3	J3	J3	J3	J3		
T2	J4	J4	J5	J5	J5	J5				
T3	J6	J6	J6	J6	J6	J6	J6	J6		

Choice A: The equal time slices assignment



	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
T0	J0	J0	J0	J0	J0	J0	J1	J1	J1	J1
T1	J2	J2	J3	J3	J3	J4	J4	J6	J6	J6
T2	J5	J5	J3	J3	J3	J6	J6	J6	J6	J6
T3	J5	J5								

Choice B: The flexible time slices assignment

Figure 6-8. Time Slices (Time Share) Assignment

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- We use first-fit strategy to allocate processors to new jobs

  All new jobs are placed in the job waiting-queue and sorted by their priorities in descending order. However, for those jobs that have the same priority we place them in first-come, first-served order.

  When scheduling new jobs, we take the first job in waiting queue and try to allocate this job with its target size in the Matrix. The allocation attempt begins from the first time slice of Matrix. If there is enough unused space for this job then allocate it; otherwise, try the second time slice until find the first time slice that can fit this job in. If the first job can be allocated in the Matrix, then remove this job from the waiting queue (the previous second job becomes the first job in current waiting queue) and place it into the tail of the job working-queue. We then repeat all above procedures until we cannot allocate the first job of waiting queue in Matrix.

- Non-continuous allotment

  For simplicity and also to avoid severe fragmentation problems associated with continuous allotment (allocating continuous processors to each job) for jobs, we allow non-continuous allotment (allocating non-continuous processors to each job); however, each job must be at the same time slice. Figure 6-9 demonstrates the idea.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
T0	J0	J0	J0	J0	J0	J0	J1	J1	J1	J1
T1	J2	J2	J3	J3	J3	J3	J3	J3		

Time interval A: J0 – J3 are scheduled

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
T0	J0	J0	J0	J0	J0	J0	J1	J1	J1	J1
T1	J4	J4	J3	J3	J3	J3	J3	J3	J4	J4

Time interval B: J2 is finished; J4 is allocated at the T1 on non-continuous processors

Figure 6-9. Non-continuous processor allotment

- Independent jobs

  We assume there is no any dependency relationship between two or more jobs, i.e., only independent jobs are considered in this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- No preemption and migration

  For current implementation, we do not consider either preemption (jobs can be check-pointed and suspended during execution, then resumed at a later time either on the same processor set or a different processor set) or migration (move jobs from the current allocated processors at current time slice to different processors even at different time slice) because of the serious overhead associated with them, see details in Section 2 and Section 3 of Chapter 2. However, by incorporating a certain degree of preemption and a certain format of migration, our adaptive scheduling algorithm might get additional benefits such as better overall job performance and more efficient system utilization. This is potential future work for our adaptive SCOJO scheduling system.
- Fragmentation and context-switching overhead calculation

  This is the first step of our adaptive scheduling algorithm. Before re-computing and re-updating the scheduling Matrix for the next time interval, we calculate the fragmentation and context-switching overhead during last time interval, and then add up to the total fragmentation and context-switching overhead encountered so far.

6.11 BACKFILLING OR EASY BACKFILLING

In space sharing or space-time sharing, not all physical processors in space sharing or all virtual processors of the same time slice in space-time sharing can be utilized all of the time, i.e., some of them have not been used during some time interval, which is called as space fragmentation. Figure 6-10 shows the fragmentation problem (marked with **X**) both in space sharing (left) and space-time sharing (right).

P0	P1	P2	P3	P4
J0	J0	J1	J1	**X**

A. Space sharing

	P0	P1	P2	P3	P4
T0	J0	J0	J1	J1	**X**
T1	J2	J2	J2	**X**	**X**
T2	J3	J3	J3	J3	**X**

B. Space-time sharing

Figure 6-10. Fragmentation in space-time sharing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Backfilling [Feitelson97B][Zhang00], originally developed for reducing space fragmentation in space-time sharing, is a technique that allows those jobs being scheduled earlier than their normal scheduled times to fill space holes (unused processors) unless they do not delay other jobs. EASY backfilling [Lifka95] [Frachtenberg03] is the same as backfilling except it does not guarantee there is no delay for other jobs, i.e., only focus on reducing the fragmentation. Figure 6-11 demonstrates the backfilling concept by en simple example. In this example job waiting queue consists of 6 waiting jobs represented by job ID and size (the number appears in bracket beside the job ID), and job 6 and Job 8 might be backfilled (scheduled before job 4, job 5, and job 7) into the Matrix to reduce fragmentation if they do not delay other jobs (for instance job 4, job 5, and job 7).

	P0	P1	P2	P3	P4
T0	J0	J0	J1	**X**	**X**
T1	J2	J2	J2	**X**	**X**
T2	J3	J3	J3	J3	**X**

Before backfilling

	P0	P1	P2	P3	P4
T0	J0	J0	J1	**J6**	**J6**
T1	J2	J2	J2	**J8**	**J8**
T2	J3	J3	J3	J3	**X**

After backfilling

Waiting Queue
J4 (3)
J5 (4)
J6 (2)
J7 (4)
J8 (2)
J9 (3)

Figure 6-11. Backfilling

6.12 FRAGMENTATION ELIMINATION

This is the Step 8 of our adaptive scheduling algorithm. Although we already use backfilling or EASY backfilling to reduce fragmentation in Step 7 of the previous section, there still can be some fragmentation left in Matrix because of some restrictions such as no-delay other jobs, suitable job size, etc. Especially because we sort the job waiting queue in descending order by priority (classified only according to job runtime class), there is little chance for those jobs at a rear position in the waiting queue having a small processor size request than jobs at a forward position in waiting queue. Although we allow the size interval of each runtime class can overlap another, it is still commonly true that long runtime jobs have more size requests than short runtime jobs.

Therefore, we continue to reduce fragmentation by taking advantage of newly scheduled (scheduled but not started) moldable and medium malleable jobs. Newly scheduled jobs refer to those jobs just taken from the job waiting-queue

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and allocated into the Matrix in previous steps for executing at next time interval. More precisely,

- Step1 – We first try to expand the sizes of newly scheduled moldable jobs

  If there is a fragmentation at a certain time slice and there is a newly scheduled moldable job at the same time slice, we expand this job's target size (determined in previous step of our adaptive scheduling algorithm) up to N_{max}. If possible, repeat the same procedure for other newly scheduled moldable jobs located at the same time slice.

- Step2 – We then consider expanding the sizes of newly scheduled medium malleable jobs (expand corresponding job target sizes up to N_{max})

  A malleable job is either a long runtime job or a medium runtime job. The reason that we exclude newly scheduled long malleable jobs here mainly is to prevent a long malleable job from expanding its size to N_{max} even with the high workload.

6.13 TIME SLICE AND JOB EXECUTION TIME UPDATE

The last step of our adaptive scheduling algorithm is to update the multiprogramming level and job runtime correspondingly after re-computing and re-updating the scheduling Matrix each time.

The multiprogramming level varies from 1 to MPL (the maximum multiprogramming level) by 1. For instance, when current time slices cannot allocate any more new jobs, the multiprogramming level will be increased by 1 up to MPL; and when certain time slice becomes empty (running jobs terminate and no new jobs wait in job waiting-queue), the multiprogramming level will be decreased by 1 down to 1.

The job runtime is influenced by many factors such as multiprogramming level, different time share assignment, job size (number of processors on which the job is running), etc. Currently, we only consider the multiprogramming level (equal time share) and job size; i.e., the job runtime will increase when the multiprogramming level is increased and the job size is decreased.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.14 TIME COMPLEXITY ANALYSIS

Time complexity analysis in computer science is normally expressed as an order of magnitude, which reflects "the way in which the number of steps required by an algorithm varies with the size of the problem it is solving" [Ludwig94]. For example, if an algorithm has $O(N^2)$ time complexity, it means that if the size of the problem (N) doubles, then this algorithm will take four times (2^2) as many steps to completely solving the corresponding problem.

As described in Section 6.4, the proposed adaptive SCOJO scheduling algorithm consists of 9 steps. We will analyze the time complexity for each step and sum them up to give the overall time complexity for the entire algorithm. In the following analysis, the problem size N refers to the total number of jobs. Then, the time complexity of the proposed adaptive SCOJO scheduling algorithm is analyzed to be in the worst case as following:

Steps	**Time complexity**	**Explanation**
Step1	$O(1)$	Executes in constant time K
Step2	$O(N)$	Executes $O(K*N)$ times
Step3	$O(N)$	Executes $O(N)$ times
Step4	$O(N)$	Executes $O(K*N)$ times
Step5	$O(N)$	Executes $O(K*N)$ times
Step6	$O(N)$	Executes $O(K*N)$ times
Step7	$O(N)/O(N^2)$	Executes $O(K*N)$ times if using EASY backfilling, otherwise executes $O(N^2)$ if using backfilling
Step8	$O(N^2)$	Executes $O(N^2)$ times
Step9	$O(N)$	Executes $O(K*N)$ times

Table 6-1. Time complexity analysis

The sum of the time complexity of the above 9 steps gives us $O(N^2)$ (no matter what kind of backfilling is applied). Hence, the overall time complexity of the adaptive SCOJO scheduling algorithm is $O(N^2)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 7: IMPLEMENTATION AND EXPERIMENT

We have chosen a discrete-event simulation to demonstrate the performance of our adaptive SCOJO scheduling algorithm, which is implemented in JAVA and experimented with on a cluster. In order to fulfill the previously mentioned goals and prove the promised better job performance and system utilization of our approach, the major testing dimensions are:

- Single vs. different multiprogramming levels

  We compare the performance of our approach with others by varying the multiprogramming levels. The maximum multiprogramming level used is 5.
- Different realistic job mixes

  We differentiate the realistic job mix by differentiating the percentage of different job runtime classes and different job types.
- Separate tests on each job runtime class and job type

  In addition to the overall performance of entire workload, we also test the individual performance of each job runtime class and job type.
- Comparison of our approach and its variants with other relevant job scheduling techniques

  The main comparison will be between our approach and the standard gang scheduling. Moreover, many variants of our approach and gang scheduling are also generated and tested.

7.1 EXPERIMENTAL ENVIRONMENT

All experiments are performed on our research HoRus cluster. The cluster has 14 nodes each contains a 2.0 GHZ Intel Xeon processor with 512 Mbyte of memory; and one front-end node that has four 700 MHZ Intel Pentium III Xeon processors. All nodes are interconnected with Myrinet.

As described in previous sections, there are many environmental parameters used by the job scheduler. The concrete values applied in simulation for these parameters are listed in Table 7-1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parameter	**Modeled values**	**Explanation**
MPL	1-5	Maximum multiprogramming level
C_{expand}, C_{shrink}	$O_{reconfigure}$ * $N_{nodes-difference}$ ($O_{reconfigure}$ = 0.0001 sec)	Cost for size adaptation (expanding or shrinking), considering size difference
N_{nodes}	64	The total number of nodes in machine
T_{slice}	2 sec * MPL/ $MPL_{current}$	The time interval between two time slices in gang scheduling; it is increased if $MPL_{current}$ (current MPL) is different from MPL
$T_{reconfigure}$	5 min	The reconfiguration time interval in which load is reclassified and size adaptation are allowed

Table 7-1. Parameters used by the job scheduler

7.2 WORKLOADS TESTED

We have tested two different workloads: ***Workload1*** and ***Workload2***, which are described in Table 7-2. The ***Workload1*** is purely synthetic and the ***Workload2*** is similar to the workload described in [Chiang2001]. Different realistic job mixes are modeled in two workloads. More precisely, ***Workload1*** models a lower percentage of long jobs and less extreme job execution times than ***Workload2***.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	Workload 1	Workload 2	Explanation
$\%_{short}$	40%	30%	The percentage of short jobs
$\%_{medium}$	40%	35%	The percentage of medium jobs
$\%_{long}$	20%	35%	The percentage of long jobs
T_{short}	[1sec, 1min]	[1sec, 3min]	Runtime interval for short jobs
T_{medium}	(1min, 30min]	(3min, 1h]	Runtime interval for medium jobs
T_{long}	(30min, 1h]	(1h, 30h]	Runtime interval for long jobs
$Size_{short}$	[1,4]	[1,4]	Size interval for short jobs
$Size_{medium}$	[4,24]	[4,24]	Size interval for medium jobs
$Size_{long}$	[8,32]	[8,32]	Size interval for long jobs
N_{jobs}	8,000	3,000	Number of jobs in the workload
$\%_{moldable}$	60%	60%	The percentage of moldable jobs
$\%_{malleable}$	30%	30%	The percentage of malleable jobs

Table 7-2. Workloads tested

7.3 PERFORMANCE METRICS APPLIED

We have comprehensively tested and measured the performance of our adaptive SCOJO scheduling algorithm. The performance metrics applied include:

- Average response time and bounded slowdown

  The average response time is the average of response times of all jobs. The bounded slowdown is the average of relative response times of all jobs; however, to avoid the misleading influence of very short jobs (i.e. in our case, corresponding execution times below 1 minute), the actual execution times of these very short jobs used in calculation are adjusted to 1 minute.

- Utilization and effective utilization of machine

  We measure the overall utilization of the machine during the entire execution of workload. The utilization of the machine is defined as the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

percentage of the processing time utilized by jobs divided by the total available processing time. The effective utilization of the machine is defined as the difference between the utilization of machine and the cost of context switching and size adaptation, which are represented in the form of processing time. The effective utilization of the machine can reflect how productive the machine is.

- Accumulated job efficiency

  The accumulated job efficiency ($E_{accumulated}$) is defined as the flowing:

$$E_{accumulated} = \sum S * T_{execution}$$

  Where, S represents the speedup and $T_{execution}$ represents the execution time of an individual job, and the accumulated job efficiency is the sum of the multiplication of the speedup and the execution time of all jobs. The accumulated job efficiency expresses how effectively the machine is utilized toward the entire computation progress of all jobs.

- Makespan

  The makespan is defined as the time from the start of the first started job to the termination of the last finished job. In other words, the makespan reflects the total time needed to finish the execution of the entire workload.

7.4 SCHEDULING STRATEGIES TESTED

We have generated many variants of our approach and gang scheduling. Since the standard gang scheduling applies the ***FCFS*** (first come first served) policy to the job queue, we use ***FCFS*** to represent the standard gang scheduling technique. Our approach – adaptive space-time sharing with SCOJO is represented by ***PRI-B-WA-FA*** in which PRI means priority, B means backfilling, WA means workload adaptation, and FA means adaptation for fragmentation reduction.

For gang scheduling, the following variants are generated and tested:

- ***FCFS-B***

  Gang scheduling with backfilling

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- ***PRI-B***

  Gang scheduling with priorities, and backfilling
- ***PRI-EB***

  Gang scheduling with priorities, and EASY backfilling that is represented by EB

For our approach, the following variants are generated and tested:

- ***PRI-WA-FA***

  Adaptive SCOJO approach with priorities, workload adaptation, and fragmentation adaptation
- ***PRI-B-WA***

  Adaptive SCOJO approach with priorities, backfilling, and workload adaptation
- ***PRI-B-FA***

  Adaptive SCOJO approach with priorities, backfilling, and fragmentation adaptation
- ***FCFS-B-WA-FA***

  Adaptive SCOJO approach with FCFS, backfilling, workload adaptation, and fragmentation adaptation
- ***PRI-EB-WA-FA***

  Adaptive SCOJO approach with priorities, EASY backfilling, workload adaptation, and fragmentation adaptation

7.5 EXPERIMENTAL RESULTS ANALYSIS

Our comprehensive experimental results provided sound evidence that the adaptive SCOJO scheduling algorithm could deliver better overall performance even with a lower multiprogramming level than the standard gang scheduling. All results are shown in Figure 7-1 to Figure 7-7:

- Figure 7-1

  This figure compares our approach (***PRI-B-WA-FA***) and gang scheduling (***FCFS***) by varying the multiprogramming level from 1 to 5. There are 8 diagrams. The left 4 diagrams show the average response time (in hours)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the right 4 diagrams show the average bounded slowdown. The upper 4 diagrams represent ***Workload1*** and the lower 4 diagrams represent ***Workload2***.

We observe the following:

- For both workloads, ***PRI-B-WA-FA*** performs much better than ***FCFS*** for the same multiprogramming level with respect to both average response time and average bounded slowdown. For example, for ***Workload2*** and a multiprogramming level of 2, the average response time of ***PRI-B-WA-FA*** is 381.02 hours vs. 2203.99 hours and the corresponding average bounded slowdown is 53.27 vs. 31017.44 of ***FCFS***.
- For both workloads, ***PRI-B-WA-FA*** performs best at a multiprogramming level of 1 with respect to average response time. This means 17.99 hours for ***Workload1*** and 362.98 hours for ***Workload2***. However, as regards the average bounded slowdown, for ***Workload1***, ***PRI-B-WA-FA*** also performs best with a multiprogramming level of 1; but for ***Workload2***, ***PRI-B-WA-FA*** performs best with a multiprogramming level of 4. Since we compare two different workloads but set the same percentage of moldable and malleable jobs for them, this tells us that the percentages of the different job runtime classes and the different job execution times play a role for the average bounded slowdown in our adaptive approach. For example, ***Workload1*** consists of 20% long jobs vs. 35% long jobs of ***Workload2***; the maximum execution time of long jobs in ***Workload1*** is 1 hour vs. a maximum of 30 hours in ***Workload2***. More precisely, the increase of the multiprogramming level sometimes does not help to decrease the average bounded slowdown, whereas it does help in conventional time sharing.
- For both workloads, ***FCFS*** performs best with a multiprogramming level of 5 for both average response times and average bounded

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

slowdowns. This means 130.97 hours (the average response time) and 3551.37 (the average bounded slowdown) for ***Workload1***, and 2102.03 hours (the average response time) and 29336.64 (the average bounded slowdown) for ***Workload2***. The results confirm that standard gang scheduling performs better with higher multiprogramming levels as other research [Feitelson97C] [Feitelson95A] discovered.

- Figure 7-2 and Figure 7-3

  These two figures show the comparison of all approaches with respect to average response time (in hours, Figure 7-2) and average bounded slowdown (Figure 7-3). A multiprogramming level of 5 is applied to gang scheduling and its 3 variants; and a multiprogramming level of 2 is applied to adaptive SCOJO scheduling and its 5 variants.

  We can observe that:

  - For both workloads and with respect to both average response time and average bounded slowdown, several adaptive approaches like ***PRI-B-WA-FA***, ***PRI-EB-WA-FA***, and ***PRI-B-WA*** perform similarly and much better than the other approaches. Other adaptive approaches like ***PRI-B-FA***, ***FCFS-B-WA-FA***, and ***PRI-WA-FA*** perform badly, even worse than priority-based gang scheduling variants like ***PRI-EB*** and ***PRI-B***. This tells us that:
    - In general, our adaptive approach (***PRI-B-WA-FA***) including its variants performs much better than standard gang scheduling (***FCFS***). For example, in ***Workload2***, ***PRI-B-WA-FA*** yields 404.12 hours of average response time and 62.26 of average bounded slowdown but ***FCFS*** yields corresponding values of 2121 hours and 29001.63.
    - Only using adaptive resource allocation for fragmentation reduction (***FA***) is not enough and dose not improve the performance much. For example, comparing ***PRI-B-WA-FA*** with ***PRI-B-FA*** in ***Workload1***, the former yields 17.61 hours

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as the average response time and 26.61 as the average bounded slowdown vs. 37.37 hours and 55.04 for the latter.

- From the three best-performing approaches - ***PRI-B-WA-FA***, ***PRI-EB-WA-FA***, and ***PRI-B-WA*** -, we can conclude that workload adaptation (***WA***), priority and corresponding aging scheme (***PRI***), and backfilling (***B***) or EASY backfilling (***EB***) play an important role in our adaptive scheduling approach. Furthermore, the combination of them delivers the best results.
- Comparing the performance of the gang scheduling variants, we can see that priority (***PRI***) and backfilling (***B***) or EASY backfilling (***EB***) can greatly improve the performance of the standard gang scheduling approach (***FCFS***). For example, in ***Workload1***, ***PRI-B*** yields 26.49 hours and 39.92 for the average response time and the average bounded slowdown, whereas ***FCFS*** yields 129.4 hours and 3473.02. This is consistent with previous research like [Zhang00] [Frachtenberg03].
- From the comparison between identical approaches with backfilling (***B***) or EASY backfilling (***EB***), we find that EASY backfilling and backfilling perform similarly in our tested workloads. For example, in ***Workload1***, EASY backfilling yields 17.05 hours of average response time and 25.59 of average bounded slowdown, whereas backfilling yields 26.61 hours and 17.61. However, due to the fairness consideration and in order to keep the original order of the job-waiting queue, backfilling is preferable than EASY backfilling.

- Figure 7-4

  This figure shows the average bounded slowdowns for different job runtime classes (i.e. short, medium, and long) and different job flexibilities (i.e. rigid, moldable, and malleable). Three approaches were tested***: PRI-EB, PRI-B-WA,*** and ***PRI-B-WA-FA***. The left diagram represents ***Workload1*** and the right diagram represents ***Workload2***.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We found that:

- In general, for both workloads and for all three approaches, the average bounded slowdowns for short jobs are smallest (i.e. short jobs perform best) compared with medium and long jobs. The same applies to moldable jobs if comparing them with rigid and malleable jobs. It tells us that our adaptive approach favors short jobs and moldable jobs. Moreover, long jobs perform worst compared with medium and short jobs.
- The priority plays the most important role here, which means higher priority jobs generally can be scheduled quicker than lower priority jobs.
- Since moldable jobs mainly consist of short jobs that are assigned the highest priority, both of them (moldable jobs and short jobs) perform best and the results show consistency in both workloads as regards average bounded slowdown. For example, in ***Workload1*** and for the ***PRI-B-WA-FA*** approach, the average bounded slowdown for short jobs is 1.54 and for moldable jobs it is 2.73, whereas the average bounded slowdown for long jobs is 118.4 and for malleable jobs it is 80.79.
- Malleable jobs perform worst compared with rigid and moldable jobs. Firstly, malleable jobs consist of many long jobs and some medium jobs and, therefore, lower priorities (vs. short jobs) are assigned to them. Secondly, even malleable jobs have the ability of dynamic size adaptation (i.e. shrinking or expanding) during execution, they more often have to shrink their sizes during the high workload since both simulated workloads are very heavy.
- Comparing rigid jobs with malleable jobs, in ***Workload1***, the average bounded slowdown for rigid jobs is 7.45 and for malleable jobs it is 80.79, which means that rigid jobs perform much better than malleable jobs; however, in ***Workload2***, the average bounded slowdown for rigid jobs is 136.25 and for malleable jobs it is 139.78,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which means that rigid jobs perform worse than malleable jobs. The reason is that rigid jobs have different percentages of the job runtime classes for the two different workloads. For instance, for ***Workload1***, rigid jobs (10% of all jobs) are only medium jobs (see Table 7-2); and for ***Workload2***, rigid jobs (10% of all jobs) consist of 5% long jobs and 5% medium jobs.

- Figure 7-5, Figure 7-6, and Figure 7-7

  These three figures show the comparison of all approaches as regards the effective utilization of the machine (in percentage, Figure 7-5), the makespan (in hours, Figure 7-6), and the accumulated job efficiency (in percentage, Figure 7-7). A multiprogramming level of 5 is applied to gang scheduling and its 3 variants; and a multiprogramming level of 2 is applied to adaptive SCOJO scheduling and its 5 variants. In order to fit the two workloads into one diagram, the time axis for ***Workload2*** in Figure 7-6 is scaled down by a factor of 10.

  We can observe that:

  - All approaches accomplish a very similar and high (above 90%) effective utilization of the machine. For adaptive approaches, high system utilization is one of the main goals and techniques like adaptive resource allocation and backfilling are applied to help to achieve this. Therefore, it is not surprising that all adaptive approaches gain high system utilization. For example, in ***Workload1***, the effective utilization of the machine is 90.32 for ***PRI-B-WA-FA***. However, even standard gang scheduling (***FCFS***) obtains high system utilization (e.g. 91.44% in ***Workload1***). The main reason is that the simulated workloads are very heavy and always keep the machine very busy. Another reason is that we apply a multiprogramming level of 5 to standard gang scheduling and its variants, which is found to provide almost the same responsiveness as an infinitely high level [Moreira1998]. The third reason for both workloads is that we have a large percentage of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

short and medium jobs (e.g., 80% in ***Workload1*** and 65% in ***Workload2***), which helps to decrease fragmentation.

- In general, almost all adaptive approaches obtain slightly worse effective system utilization than standard gang scheduling and its variants. This is mainly due to the cost of adaptive resource allocation. For example, in ***Workload2***, ***PRI-B-WA-FA*** obtains 90.31%; ***PRI-WA-FA*** obtains 90.78%; ***FCFS-B*** obtains 90.6%; and ***FCFS*** obtains 91.67%.
- Except ***PRI-B-FA***, all other adaptive approaches have a smaller makespan (see Figure 7-6) than standard gang scheduling and its variants. For instance, in ***Workload1***, the makespan of ***PRI-B-WA-FA*** is 506.2 hours and the makespan of ***FCFS-B*** is 560.46 hours.
- Except ***PRI-B-FA***, all other adaptive approaches yield higher accumulated job efficiency (see Figure 7-7) than standard gang scheduling and its variants because we take application information (in this case, the speedup curves) into consideration. For instance, in ***Workload2***, the accumulated job efficiency of ***PRI-B-WA*** is 71.41% and the accumulated job efficiency of ***PRI-B*** is 64.99%.
- The reason why ***PRI-B-FA*** performs worst with respect to the effective utilization of the machine, the makespan, and the accumulated job efficiency mainly is that the fragmentation reduction (***FA***) is so limited in our adaptive approach. For example, the general procedure related to fragmentation in our adaptive scheduling algorithm is: first do workload adaptation (e.g. shrinking or expanding job sizes, which is Step 5); then do backfilling or EASY backfilling (Step 7); at last do fragmentation reduction (Step 8). Therefore, firstly, after workload adaptation and backfilling, there is not too much fragmentation left for the ***FA*** step. Secondly, since ***FA*** only expands new moldable and new medium malleable job sizes, there is little flexibility left. At last, there is no possibility to shrink and expand the sizes of currently running malleable jobs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(e.g. without ***WA*** – workload adaptation) in this approach (***PRI-B-FA***), long malleable jobs will stay with their optimal size (which is not the maximum size they can have) along the entire execution. This means there is no size adaptation for long malleable jobs at all in ***PRI-B-FA***.

7.6 SUMMARY AND DISCUSSION

The above experimental results can be summarized and discussed as following:

- Adaptive SCOJO scheduling delivers much better results than standard gang scheduling for almost all performance metrics measured like average response time, average bounded slowdown and accumulated job efficiency, even with a lower multiprogramming level.
- As regards another main performance metric – the effective utilization of the machine –, the simulated workloads are very heavy, i.e., the total number of jobs is very large, the inter-arrival times of the jobs are very short, and the job sizes and job runtimes of long jobs are very large especially for ***Workload2***. Therefore, a high efficient utilization of the machine (i.e. around 90% to 92%) is provided by almost all approaches. Fragmentation is typically less than 0.5% for all approaches and the rest is overhead.
- By considering real application characteristics like speedup curves in adaptive resource allocation, the adaptive SCOJO scheduling provides a great increase in overall productive usage of the machine (with respect to the accumulated job efficiency) compared with standard gang scheduling.
- Adaptive SCOJO scheduling performs best in most cases for a multiprogramming level of 1, though the average bounded slowdown for ***Workload2*** is best for a multiprogramming level of 4. This demonstrates our initial claims that the adaptive SCOJO scheduling can work well with a lower multiprogramming level.
- Adaptive SCOJO scheduling works well with realistic job mixes that consist of many moldable, some rigid, and some malleable jobs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Adaptive SCOJO scheduling with workload adaptation (i.e. ***PRI-B-WA-FA***, ***PRI-B-WA-FA***, and ***PRI-B-WA***) provides the best results. This demonstrates that the benefit gained for the adaptive resource allocation mainly comes from the workload adaptation (***WA***).
- Fragmentation adaptation (***FA***) by itself does not perform well because the fragmentation adaptation is very limited in our approach. A better approach to fragmentation is the potential future work for this thesis.
- Priorities play a very important role in adaptive SCOJO scheduling and also in variants of standard gang scheduling and deliver much better results than the first-come, first-served policy (***FCFS***).
- Backfilling or EASY backfilling can greatly improve the overall job performance by giving benefits for short jobs and medium jobs. In our test cases, the performance difference between them is little.
- Short jobs and moldable jobs perform much better than jobs with other runtime classes and other flexibilities. This indicates that we might be able to further improve the overall job performance by giving additional benefits to medium and long jobs via a more aggressive aging scheme to priorities.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

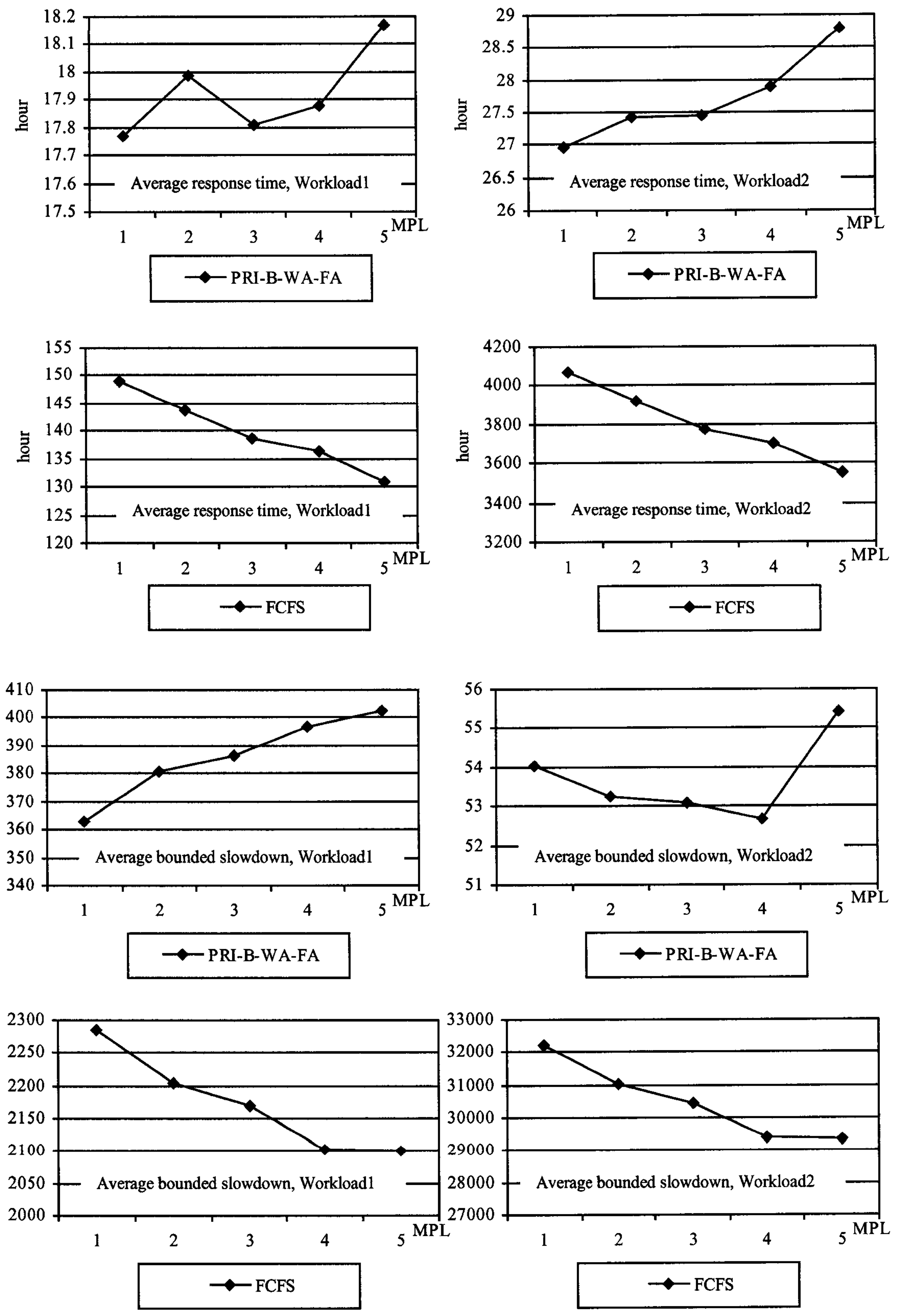


Figure 7-1. Comparison (of FCFS and PRI-B-WA-FA) **on varying multiprogramming levels**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

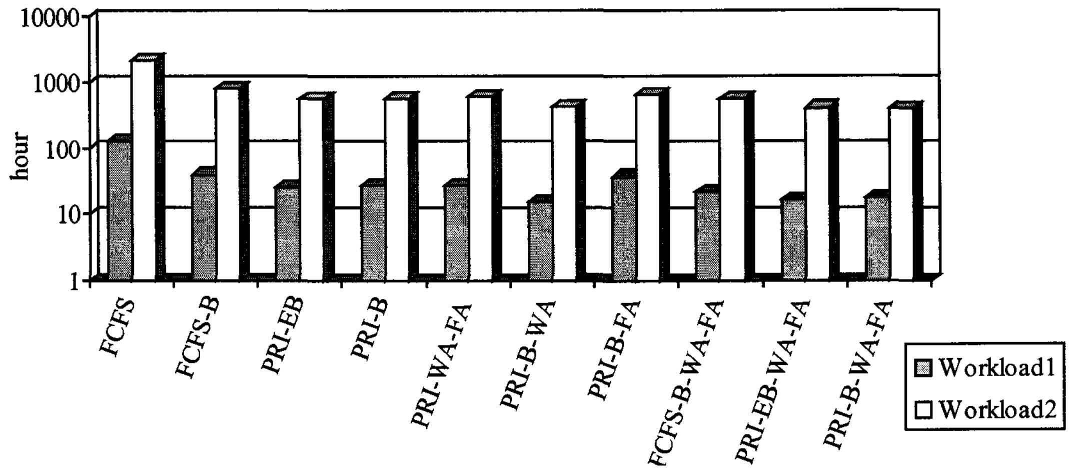


Figure 7-2. Comparison of all approaches on average response time

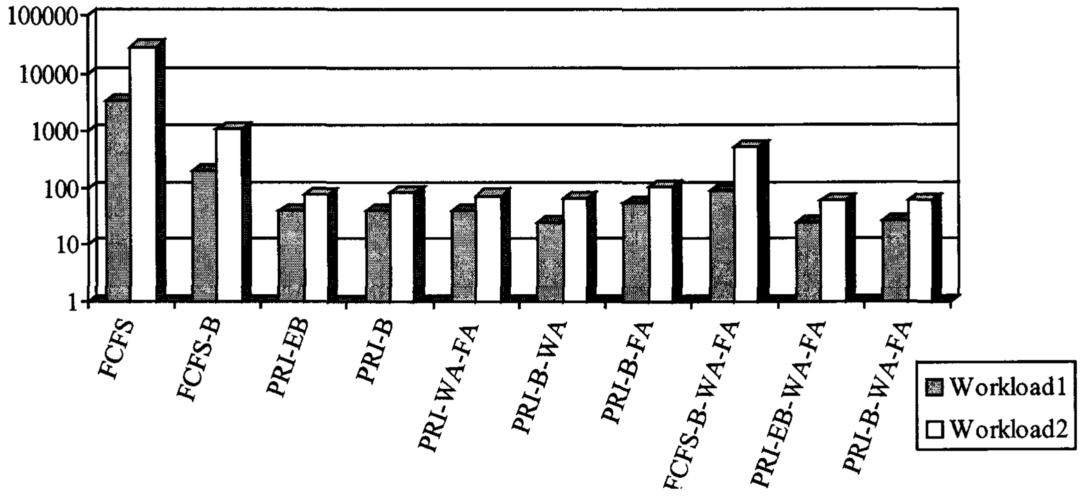


Figure 7-3. Comparison of all approaches on average bounded slowdown

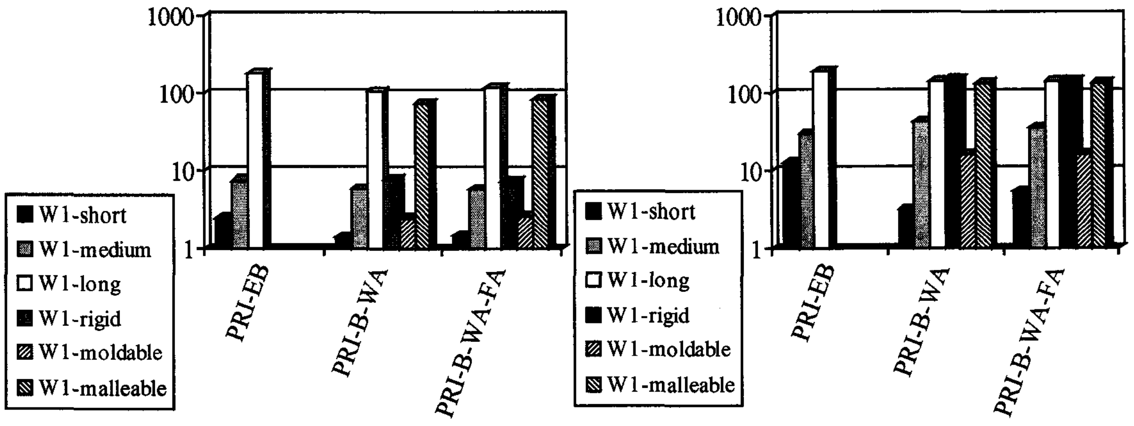


Figure 7-4. Average bounded slowdown for different job runtime classes and job types

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

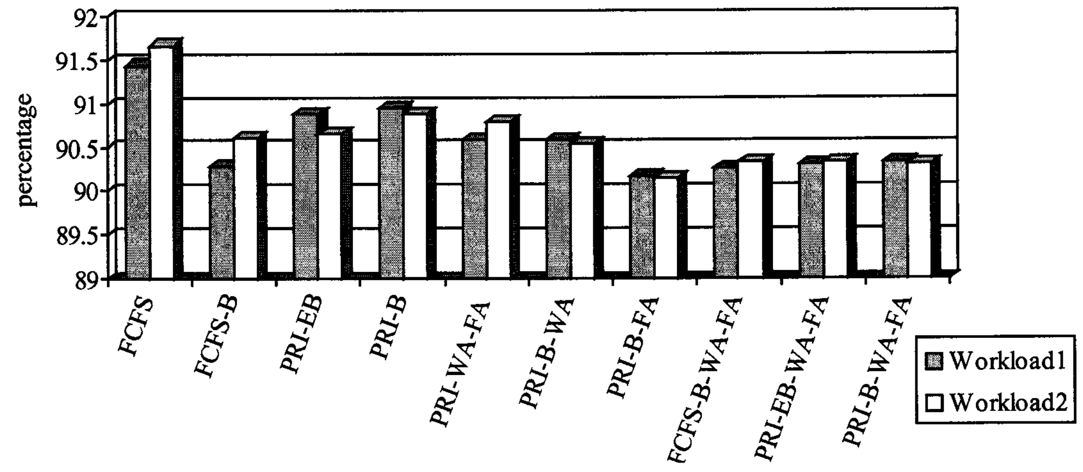


Figure 7-5. Comparison of all approaches on effective utilization of machine

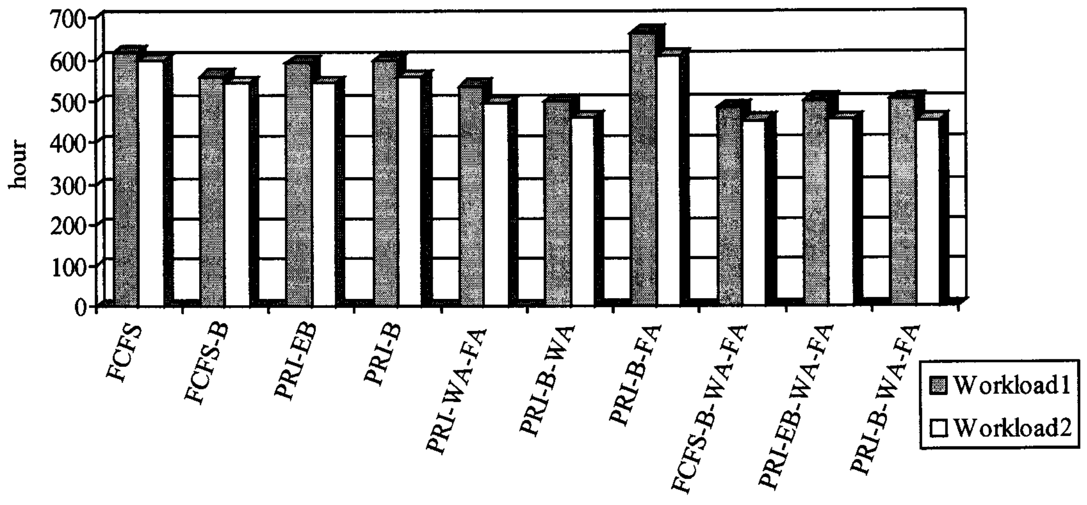


Figure 7-6. Comparison of all approaches on makespan

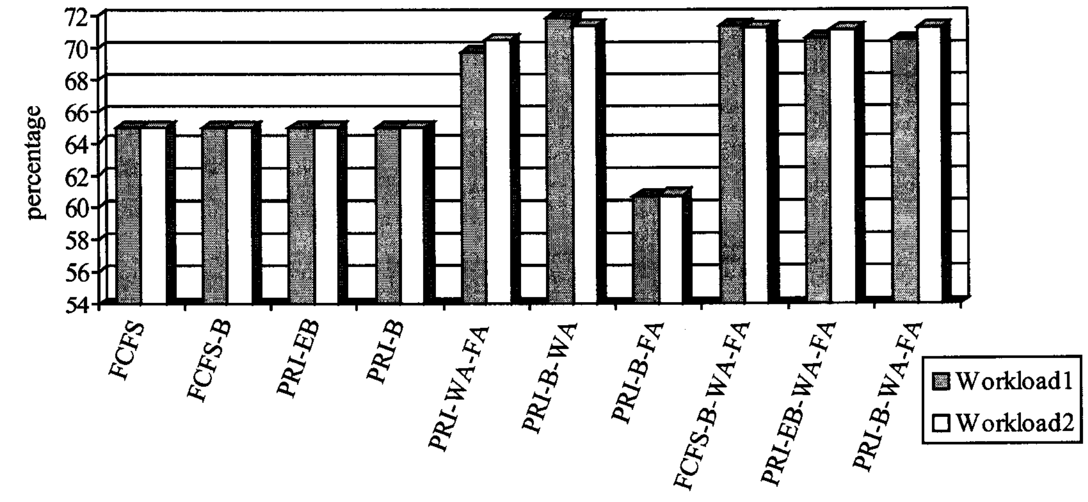


Figure 7-7. Comparison of all approaches on accumulated job efficiency

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CONCLUSION

We have presented a new approach – Adaptive Space-time Sharing with SCOJO, which incorporates adaptive resource allocation into gang scheduling. It also applies other standard job scheduling techniques like backfilling; and it considers realistic job mixes of rigid, moldable, and malleable jobs. Our approach adjusts job sizes to adapt to workload changes and reduces fragmentation based on a clear model. Moreover, the relevant context-switching overhead and adaptation cost are considered.

The experimental results show that our approach can deliver significantly better average response times and average bounded slowdowns than standard gang scheduling. The performance gained mainly comes from workload adaptation; fragmentation adaptation contributes little. Moreover, our approach works well with standard backfilling; and EASY backfilling does not yield much improvement. Most importantly, our approach performs well even with a lower multiprogramming level. This suggests that gang scheduling may not be needed at all to avoid context-switching overhead and memory pressure. The mere space sharing (the multiprogramming level equals to 1) in combination with adaptive resource allocation may even provide the best result.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

[Chiang01] S.-H. Chiang and M. K. Vernon. “Characteristics of a large Shared Memory Production Workload”. Procs. JSSP, 2001.

[Chiang94] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. “Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies”. In SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 33-44, May 1994.

[Corbalan01] Julita Corbalan , Xavier Martorell , Jesus Labarta. “Improving Gang Scheduling through job performance analysis and malleability”. Proceedings of the 15th international conference on Supercomputing June 2001.

[Dutot01] Pierre-François Dutot, Denis Trystram. “Scheduling on hierarchical clusters using malleable tasks”. Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures July 2001.

[Feitelson97A] Feitelson D., “Job scheduling in multiprogrammed parallel systems”, Technical Report, IBM T.J. Watson Research Center, Second Revision, 1997.

[Feitelson97B] D. G. Feitelson, L. Rudolph, U. Schweigelshohn, K. Sevcik, and P. Wong, “Theory and practice in parallel job scheduling”. In Job Scheduling Strategies for Parallel Processing D. G. Feitelson and L. Rudolph (Eds.), pp. 1-34, Springer-Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.

[Feitelson97C] D. G. Feitelson, M. A. Jette, “Improved Utilization and Responsiveness with Gang Scheduling”. In Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science vol. 1291, pp. 238-261, Springer-Verlag, 1997.

[Feitelson96A] D.G.Feitelson and L.Rudolph. "Toward convergence in job schedulers for parallel supercomputers". In Job Scheduling Strategies for Parallel Processing, pp. 1-26, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[Feitelson96B] D. G. Feitelson, “Packing schemes for gang scheduling”. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), pp. 89-110, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[Feitelson95A] D. G. Feitelson and L. Rudolph, “Parallel job scheduling: issues and approaches”. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), pp. 1-18, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[Feitelson95B] D. G. Feitelson and B. Nitzberg. “Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860”. In Proceedings of the IPPS’95 Workshop on Job Scheduling Strategies for Parallel Processing, pages 337–360, Apr. 1995.

[Feitelson92] D. G. Feitelson and L. Rudolph. "Gang Scheduling Performance Benefits for Fine-Grained Synchronization". Journal of Parallel and Distributed Computing, 16(4): 306-318, December 1992.

[Feitelson90] D. G. Feitelson and L. Rudolph, “Distributed hierarchical control for parallel processing”. Computer 23(5), pp. 65-77, May 1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Frachtenberg03] Eitan Frachtenberg, G. Feitelson, Fabrizio Petrini, Juan Fernandez. "Flexible Coscheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources". Proc. Int. Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 2003.

[Ghosal91] Ghosal, D.; Serazzi, G.; Tripathi, S.K.; "The processor working set and its use in scheduling multiprocessor systems". Software Engineering, IEEE Transactions on, Volume: 17 Issue: 5, May 1991 Page(s): 443 –453.

[Gupta91] A. Gupta, A. Tucker, and S. Urushibara. "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications". ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, Vol. 19, pp. 120-132, Assoc. Comput. Mach., New York, 1991.

[Ishikawa99] Yutaka Ishikawa, Hiroshi Tezuka, Atsuhi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O'Carroll, and Hiroshi Harada. "RWC PC Cluster II and SCore Cluster System Software -- High Performance Linux Cluster". Proceedings of the 5th Annual Linux Expo, pages 55 -- 62, 1999.

[Kazi02] Kazi, Iffat H.; Lilja, David J. "Dynamically adapting to system load and program behavior in multiprogrammed multiprocessor systems". Concurrency and Computation: Practice and Experience Volume: 14, Issue: 12, October 2002, pp. 957 – 985.

[Kazi00] Kazi, I.H.; Lilja, D.J.; "A comprehensive dynamic processor allocation scheme for multiprogrammed multiprocessor systems". Parallel Processing 2000. Proceedings. Page(s):153–161.

[Leutenegger90] S. Leutenegger and M. Vernon. "The performance of multiprogrammed multiprocessor scheduling policies". ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, Vol. 18, pp. 226-236, Assoc. Comput. Mach., New York, 1990.

[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.

[Ludwig94] W.T. Ludwig and P. Tiwari. "Scheduling malleable and nonmalleable parallel tasks". In Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. ACM-SIAM, 1994.

[Majumdar88] S. Majumdar, D. Eager and R.B.Bunt. "Scheduling in multiprogrammed parallel systems". Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (May 1988), pp. 104-113.

[McCann94] Cathy McCann, John Zahorjan. "Processor allocation policies for message-passing parallel computers". ACM SIGMETRICS Performance Evaluation Review, v.22 n.1, p.19-32, May 1994.

[McCann93] Cathy McCann, Raj Vaswani, John Zahorjan. "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors". ACM Transactions on Computer Systems (TOCS) May 1993 Volume 11 Issue 2.

[Moreira98] Jose E. Moreira, Waiman Chan, Liana L. Fong, Hubertus Franke, and Morris A. Jette. "An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments". Supercomputing'98, Nov. 1998

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Naik97] Vijay K. Naik, Sanjeev Setia, Mark S. Squillante. “Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems”. Journal of Parallel and Distributed Computing, pp. 28-47, Volume 46, 1997.

[Naik93] V.K.Naik, S.K.Setia and M.S.Squillante. "Performance analysis of job scheduling policies in parallel supercomputing environments". In Supercomputing'93, pp. 824-833, Nov 1993.

[Nguyen96] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. "Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling". In D. G. Feitelson and L. Rudolph, editor, Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing, volume 1162. Springer Verlag, 1996.

[Ousterhout82] Ousterhout, J.K. "Scheduling Techniques for Concurrent Systems". In Third International Conference on Distributed Computing Systems, pp. 22-30. 1982.

[Sevcik89] K.C.Sevcik. "Characterizations of parallelism in applications and their use in scheduling". In SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 171-180, May 1989.

[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.

[Sodan03] Angela C. Sodan. “Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling – A Survey”. Technical Report 03-006, May 2003.

[SodanHuang03] Angela C. Sodan, Xuemin Huang. “SCOJO – Share-Based Job Co-scheduling With Integrated Dynamic Resource Directory In Support Or Grid Scheduling”. Proc. Ann. Int. Symposium on High Performance Computing (HPCS), Sherbrooke, Canada, May 2003, pp. 213-221.

[Tucker89] A. Tucker, A. Gupta. “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors”. Proceedings of the twelfth ACM symposium on Operating systems principles, p.159-166, November 1989.

[Turek92] John Turek, Joel L. Wolf, Krishna R. Pattipati, Philip S. Yu. “Scheduling parallelizable tasks: putting it all on the shelf”. ACM SIGMETRICS Performance Evaluation Review, Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems June 1992, Volume 20 Issue 1.

[Wolski99] Rich Wolski, Neil Spring, and Jim Hayes. “The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing”. Journal of Future Generation Computing Systems, Volume 15, Numbers 5-6, pp. 757-768, October, 1999.

[Yue98] Yue, K.K.; Lilja, D.J.; “Dynamic processor allocation with the Solaris operating system”. Parallel Processing Symposium, 1998. 1998 IPPS/SPDP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proceedings of the First Merged International...and Symposium on Parallel and Distributed Processing 1998, 30 Mar-3 Apr 1998 Page(s): 392 –397.

[Zhang01] Y. Zhang, H. Franke, J. E. Moreira, A. Sivasubramaniam. “An Intergrated Approach to Parallel Scheduling Using Gang Scheduling, Backfilling and Migration”. Proc. JSSPP, 2001.

[Zhang00] Y. Zhang, H. Franke, J. E. Moreira, A. Sivasubramaniam, “Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques”, IPDPS2000, Cancun, Mexico, May 2000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA AUCTORIS

NAME:	Xuemin Huang
PLACE OF BIRTH:	Lanzhou, Gansu Province, China
YEAR OF BIRTH:	1971
EDUCATION:	The Third High School, Lanzhou 1985-1988
	Hohai University, Nanjing, Jiangsu Province, China 1988-1992 B.Sc.
	University of Windsor, Windsor, Ontario 2000-2001 B.Sc.
	University of Windsor, Windsor, Ontario 2002-2004 M.Sc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.