Scheduling on Timesharing Multiprocessors

Walter Lee*
walt@lcs.mit.edu
The Fugu Project

1 Introduction

Ousterhout introduced coscheduling as a key concept in scheduling in multiprocessor timesharing environments. [5] The CM5 recognized this concept when it implemented strict gang scheduling. CM5's idea of providing a job with a virtually dedicated machine is attractive for two reasons. From the protection standpoint, independent jobs are protected from each other via barriers in the time domain. From the performance standpoint, the lack of interference guarantees that a job does not incur any slowdown when it is running. In particular, a process in a job with fine-grained communication does not have to incur substantial overhead from trying to communicate with a descheduled process.

Since then, the protection issue has been decoupled from the performance issue, so that rigid partitioning is not a prerequisite for enjoying the performance benefit of coscheduling. Recent research in this area has focused on two fronts. One front searches for coscheduling algorithm which address the two shortcomings of primitive coscheduling algorithms: scalability [3] and poor machine utilization [2]. The other front studies mechanisms which can approximate coscheduling behavior in an environment where explicit coscheduling is expensive to implement [1] [6]. The collection of ongoing research may lead one to believe that this area of research is governed by a single commandment, "Thou shall coschedule." However, coscheduling is only a single thread in the web of complexity in multiprocessor scheduling. For one thing, some jobs may not require coscheduling to achieve good performance. Since coscheduling is often realized at the cost of poorer machine utilization, it may be more efficient to use a less restrictive scheduling requirement for some jobs. At a higher level, the performance bottom line tends to lose its place amidst all the hype on coscheduling.

This work studies scheduling in a multiprocessor timesharing environment in a more general framework than simply coscheduling. It first proposes a simple model with which one can evaluate scheduling algorithms and policies. Then it introduces the major scheduling concepts one needs to consider when building a good scheduler. Next, it proposes a scheduling policy based on these concepts. Finally, it briefly discusses the status and direction of present research.

*Walter Lee is advised by Anant Agarwal and are supported in part by ARPA contract N00014-94-1-09885, by NSF Experimental Systems grant MIP-9012773, and by an NSF Presidential Young Investigator Award to Prof. Agarwal. Address: Room 626, 545 Technology Square, Cambridge MA 02139.

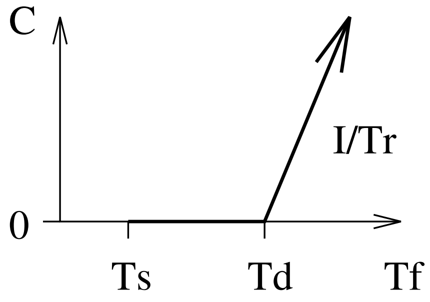

Figure 1: Cost vs finish time of a single job.

2 Model for scheduler evaluation

Schedulers which have been implemented in multiuser multiprocessor environments are surprisingly diverse. Samples include rigid space partitioning with no timesharing, processor scheduling, and timesharing with no space partitioning. This lack of uniformity necessitates identifying a universal performance metric which can be used to compare different mechanisms. An essential feature of this metric is that it must correctly capture the roles to be played by throughput and response time in scheduler evaluation.

To this end, we propose the following cost function: (See figure 1):

$C = \sum_j \max(0, T_{f\,j} - T_{d\,j}) * I_j / T_{r\,j}$

Each job is represented by five quantities. $T_{f\,s}$ is the start time; $T_{f\,j}$ is the finish time; $T_{d\,j}$ is the deadline time; I_j is the importance; and $T_{r\,j}$ is the run time given a fully dedicated machine. $T_{f\,s}$ is not in the function explicitly, but it is implicit in the value of $T_{d\,j}$. Each term in the cost function has a nonnegative value, and the goal of the scheduler is to minimize the cost.

The cost function uses $T_{d\,j}$ and I_j to directly capture the notions of deadline scheduling and externally assigned priorities, respectively. The way it captures the throughput versus response time tradeoff is more subtle and interesting. Consider the following simplified version of the function, where $I_j = 1$ and $T_{d\,j} = T_{s\,j}$:

$C = \sum_j (T_{f\,j} - T_{s\,j}) / T_{r\,j}$

This cost has a straightforward meaning: it measures the aggregate slowdown of all jobs within a system. Therefore, minimizing the function minimizes weighted response time. The equation does not maximize throughput explicitly; instead, throughput is directly proportional to machine utilization, which in turn is viewed as a means through which low response time can be achieved. This user-centric view of the world optimizes directly for what is important to the users. It argues against the common, system administrator approach which accentuates utilization over response time.

3 Scheduling concepts

Three scheduling concepts need to be considered in order to build a good scheduler. They are shortest job first, intra-job fairness, and coscheduling.

Shortest job first is known to be optimal for minimizing response time in a uniprocessor system. Its behavior in a multiprocessor system is more complex, because it is possible to delay running short jobs to improve utilization and indirectly reduce the aggregate response time. However, given a response time driven cost function, this strategy cannot be ignored. In particular, this means giving a high priority to I/O bound jobs, and also attempting to predict the run time of compute bound jobs.

A scheduler is *intra-job fair* to a job if it allocates the same amount of time to each process within a job. Conventional parallel programs are written in such a way that the runtime of individual processes between synchronization has a small variance. If the synchronization pattern of these programs are course-grained as well, ensuring intra-job fairness may be sufficient to ensure good performance of these programs.

Coscheduling is the idea that processes belonging to the same job should be scheduled simultaneously across all the machines. It is a necessity for jobs with fine grained synchronization patterns. Note that coscheduling also ensures intra-job fairness.

4 Scheduling algorithm

Based on the above concepts, we propose the following scheduling algorithm. It will form the basis of the scheduler to be implemented on Fugu [4].

The scheduler is responsible for managing both processor allocation and scheduling of jobs. Processor allocation is performed when a job request enters the system, using the buddy based packing algorithm [2]. This algorithm assigns processors in predefined groups of powers of two. It is designed to enable scheduling to be performed so that processor resources are devoted to coscheduled jobs as much as possible.

For the purpose of scheduling, jobs are either classified as I/O bound or CPU bound. A multiprocess job is considered to be I/O bound if any of its process is I/O bound. CPU bound jobs are further divided into two groups based on whether they require gang scheduling. I/O bound jobs are assumed not to require gang scheduling.

Jobs are also prioritized. The set of I/O bound jobs has higher priority than the set of CPU bound jobs. I/O bound jobs are prioritized in order of their reciprocal run lengths, while CPU bound jobs are prioritized in order of their reciprocal accumulated run times.

A centralized agency is responsible for computing and distributing a schedule. This schedule consists of two components: a two dimensional matrix for scheduling gangs, with processor number on one axis and time slice number on the other; and a processor-indexed array of job lists for scheduling non-gangs. There is an upper limit on the number of time slices in a schedule. The schedule is computed by traversing the list of jobs in order of priority and fitting them into the schedule, space permitting. A gang job is fit into the schedule by picking a single time slice when it should be scheduled and filling in the appropriate boxes in the matrix. A non-gang job is fit into the schedule by appending it to the lists of the processors in which the job's processes reside.

Once a schedule is determined, each processor receives its portion of the schedule. For time slices filled with gang jobs, the local scheduler schedules those specified jobs. For time slices not reserved for gang jobs, the local scheduler runs jobs on the scheduled job list for their preallotted amount of time. A schedule is valid for the period during which the scheduling matrix is defined. When the matrix runs out, a new schedule is computed.

5 Status and continuation

The scheduling algorithm is still under examination and may be revised before it is actually implemented. Some implementation issues, such as the runtime identification of gang jobs, still need to be resolved. The algorithm will eventually be implemented and evaluated on Fugu.

[1] Andrea Dusseau, Remzi Arpaci, and David Culler. Effective distributed scheduling of parallel workloads. In *Third International Conference on Distributed Computing Systems*. Berkeley, May 1996.

[2] Dror Feitelson. Packing schemes for gang scheduling. In *IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 54–66. Hebrew University, April 1996.

[3] Dror Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–77, May 1990.

[4] Ken Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing translation and protection in a multiuser, multimodel multiprocessor. MIT/LCS/TM 503, MIT, October 1994.

[5] John Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30. Berkeley, October 1982.

[6] Pat Sobalvarro and William Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75. MIT, April 1995.