Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling

David Talby Dror G. Feitelson
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Running jobs on the IBM SP2, as in most distributed memory parallel system in the market today, is done by giving each job a subset of the available processors for its exclusive use. Scheduling jobs in FCFS order suffers from severe fragmentation that leads to utilization loss. This led Argonne National Lab, where the first large SP1 was installed, to develop the EASY scheduler, which has since then been adopted by many other SP2 sites. This scheduler uses aggressive backfilling: Small jobs may be moved way ahead in the queue, while large jobs may suffer an unbounded delay. A more conservative backfilling strategy, which retains the predictability feature of FCFS, seems to equalize EASY's performance on average workloads. None of the above schedulers support prioritization — allowing the administrators or the users themselves to schedule a job with a high or low priority, subject to accounting considerations such as paying for priority, preferred groups, quotas, emergencies and so on.

This paper presents a scheduler that supports both user and administrative priorities. The scheduler gives each waiting job a slack, which determines how long it may have to wait before running: 'Important' and 'heavy' jobs will have little slack in comparison with others. When a new job is submitted, all possible schedules are priced according to utilization and priority considerations and as long as no job is delayed beyond its slack, the cheapest schedule is chosen. This gives the scheduler the flexibility to effectively backfill, and preserves the bounded delay advantage of FCFS and conservative backfilling over EASY. Experimental results show that the priority scheduler reduces the average wait time by about 15% relative to EASY in an equal priorities scenario, and is responsive to differential priorities as well.

1 Introduction

1.1 The Problem: Utilization

Most currently available distributed memory supercomputers require users to submit the number of processors required for a job they wish to run. When the requested number of processors becomes available, the job is executed, and the processors are dedicated to it until it terminates or is killed. This scheme is called variable partitioning [3]. Usually the partitions of processors are allocated on a FCFS basis, where interactive jobs are submitted directly and batch jobs are submitted via a queuing system such as NQS. This approach results in severe fragmentation, because processors which cannot fulfill the demands of the next job in the queue must remain idle until more processors are freed. FCFS based schedulers show a typical system utilization of 50-80% [6, 10, 12, 16].

Two solutions have been proposed to this problem, but both suffer from practical limitations. This first is dynamic partitioning [15], in which jobs may gain or lose some of their processors dynamically during their lifetime. Jobs may also be halted in favor of other jobs, and renewed later, possibly on a different set of processors. This approach proves to be an improvement over variable partitioning, but it requires complex operating system software adjustments and the development of some means of programmer intervention, in cases where the number of processors a job has changes. Dynamic partitioning has never been implemented in a production system.

The second solution to the efficiency shortcoming of variable partitioning is gang scheduling [4], but the only efficient and widely used implementation of it so far was on the CM-5 connection machine. Other implementations are too coarse grained for real interactive support, and do not enjoy much use.

A far simpler approach is to use a non-FCFS policy for queuing the waiting jobs on a variable partitioning system [7]. This requires minimal change of current production systems, but may still lead to considerable utilization improvements. For example, consider the scenario in which several small jobs are running, and a job that requires the entire system is the first in the queue. A FCFS scheduler will reserve freed processors until the entire system is available, keeping processors idle until the last running job has terminated [11, 1]. A non-FCFS scheduler could execute small jobs from the back of the queue until the last job finishes, therefore improving the total utilization of the system. Such an approach is called backfilling: Idle partitions of processors can be filled by small jobs from the back of the queue. Clearly, the system should be cautiously designed not to starve large jobs.

The first large installation of the IBM SP1 was in Argonne National Lab, initially using FCFS scheduling. The lab developed a new scheduler that was based on an aggressive backfilling strategy and was part of EASY (Extensible Argonne Scheduling sYstem) [14]. The system has since then been integrated into IBM's LoadLeveler for the SP2 [17]. The EASY scheduler requires users to submit an estimation of the job's runtime along with it. Users are encouraged to submit accurate estimations: A low estimation may lead to killing the job before it terminates, and a high estimation may lead to a long wait time and possibly to excessive CPU quota loss. Using the users' estimations, the EASY scheduler backfills a jobs (moves it ahead in the queue) if it doesn't delay the first job in the queue.

A recent project [5] has shown two drawbacks of EASY, which are in essence two faces of the same problems. First, jobs may suffer an unbounded delay, and second, the scheduler is unpredictable and cannot commit to users about when their jobs will run. It was shown that a more conservative backfilling strategy, in which a job is used for backfilling only if it doesn't delay any other job in the queue, produces the same performance as EASY on average workloads, while guaranteeing the user the job's latest possible start time upon its submission. The conservative backfilling eliminates the above two problems of EASY while maintaining the same level of performance.

1.2 The Problem: Priorities

Supercomputers are typically used by several groups and projects at once — academic, business oriented, military and so on. The administrators may wish to give each of them a different priority, or enforce a CPU quota on groups, projects or users. Different users within a system may also wish to prioritize themselves: Although a small academic project has a lower priority than an major military one, the college project may have a nearby deadline and wish to temporarily increase its priority, even at some cost. Administrators can 'charge' users for using a higher priority by requiring more CPU quota for such a job, or by other more explicit mechanisms.

This functionality does not exist in the current schedulers available for the SP2 or similar systems [14]. In fact, SP2 doesn't even monitor the users' CPU quota use online, and it is checked periodically. The new scheduler to be presented will support both user and the so-called political priorities [13], which give a preference to selected groups or projects. Moreover, the priority mechanism will also be used to improve the chance that jobs will not wait substantially longer than the average wait time. If a job initially has to wait more than the system's average wait time its priority will increase, and if it has been initially 'lucky', its priority will decrease. This will give unlucky jobs an advantage in future backfilling operations, if there are any.

When users or projects exceed their quota for any system resource, the usual counter-action is to entirely block their access to the system. A softer policy would be to allow them to run jobs as long as they are not interrupting any other legitimate job. This policy seems preferable, and we implemented it by allowing the assignment of an 'absolute zero' priority to such jobs.

It is desirable that inserting priorities into the system will not deteriorate its performance, but this may be an impossible goal. Consider, for example, a scenario in which a high priority job that needs few processors is the first in the queue and a low priority jobs that needs all processors is the second. We would wish to run the second job first, since it may increase utilization (smaller jobs that come later could run side by side with the first job). However, this would be in contrast with the priority requirement. The scheduler should obviously have some method of weighing the importance of each requirement in such a case in order to make the best decision.

The priority scheduler presented here resolves the above situation by a pricing mechanism, which prices each possible schedule and then chooses the 'cheapest' one according to weighed costs of utilization, time, priorities and fairness. The scheduler performs better than EASY or conservative backfilling in an equal priorities scenario, and even when differential priorities are used, the low priority jobs wait less than in EASY. The weighed importance given to different considerations (for example, utilization versus priorities) can be set by the administrators. The resulting mathematical model seems simpler than other popular priority schedulers [2, 18, 9].

The following sections include a description of EASY, conservative backfilling, and finally the priority scheduler. Afterwards, performance results from simulations based on real production workloads are presented and discussed.

2 Requirements

2.1 Formalities

A job that a user submits will be denoted as $j =< n, t, up, pp, t_0 >$. n is the required number of processors, t is the amount of runtime the user estimates the job requires, up and pp are the job's user and political priority (which will be further discussed in section 2.3), and t_0 is the arrival time of the job to the queue. A full representation of a job in the system will be $j =< n, t, p, t_0, t_e, s, s_0 >$. t_e is the time in which the job is scheduled to start executing (it may change several times before the job actually runs), p is the weighed priority of the job, s is the job's current slack, and s_0 is the job's initial slack. Slack will be measured in units of time. The EASY and conservative schedulers obviously ignore priorities and slack, and simply require a job's n, t and t_0.

A schedule will be represented by a profile of running and waiting jobs. For each point in time, the profile should include information about which jobs are assigned to which processors at that time. Usually we'll refer to time slots (intervals) and not individual points. Note that a profile can be implemented as an associative array of time slots and a list of scheduled jobs for each slot; its size is linear in the total number of running and waiting jobs.

Special points in time are ST — start time, or the earliest time slot; CT — current time; and ET — end time, or the last time slot. The constants PROCS (number of processors) and AWT (average wait time) of the system are assumed to be given. The idleness of time slot ts is defined as the unused CPU time in it, which is the product of its duration and the number of its free processors.

Graphically, jobs will be represented as a rectangle in a processors/time space, and profiles will be represented as a grid of jobs over such a space (Fig. 1). The schedule in the figure represents the situation when $J1$ terminates, $J2$ and $J3$ start running, and $J4$ waits.

A scheduler is basically an event driven program that supports these actions:

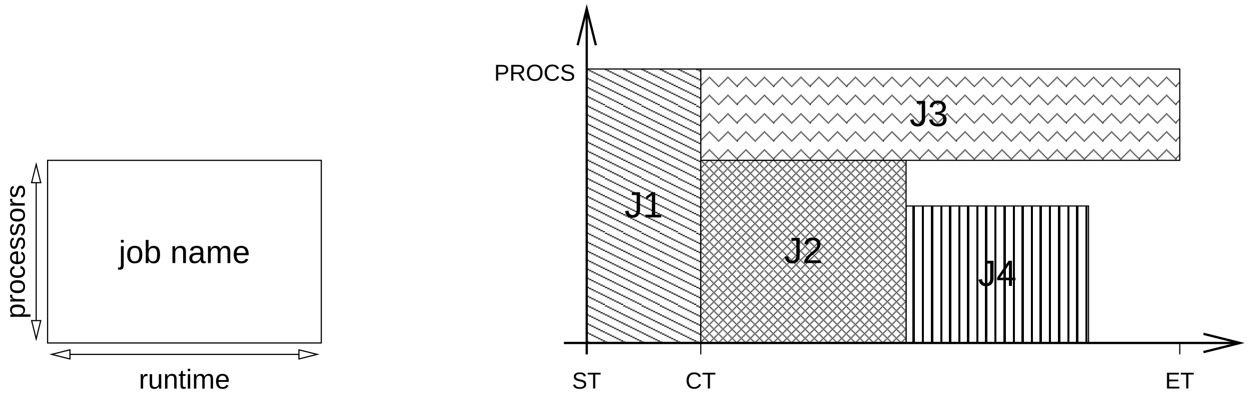

Figure 1: *Graphical representation of a job and a schedule.*

Insert(j) — A user requests to execute a new job.

Remove(j) — A user cancels a request, or a running job finishes running before its expected runtime $j.t$.

Tick() — Every time the current time reaches the beginning of a time slot ts, new jobs should be executed and jobs that should have ended may be killed.

2.2 Basic Requirements

The following describes the environment in which the scheduler works:

I. Users submit jobs (as described above) dynamically. The scheduler should insure the highest possible CPU utilization, subject to priority and fairness.

II. One job may be assigned to a processor at any time.

III. A job must be given all the n processors it requests at once.

IV. No preemption: A running job can't be stopped, and it's impossible to change the processors assigned to it.

V. Jobs are independent of one another, and don't create other jobs.

VI. There exists an accounting system that encourages users to give accurate time estimates for their jobs.

The following are the basic requirements from the scheduler:

VII. There is no starvation.

VIII. If a job runs more than the time it asked for $j.t$, it may be killed if continuing to run it would delay other jobs. This, however, causes a utilization penalty.

IX. When a job is submitted the user is given an upper bound on its start time.

2.3 Priority Requirements

There are three stakeholders in determining the priority of a new job:

UP — User priority. The user can give the job a priority in the range [0,1].

PP — Political or Administrator priority. Every user, group and project have such a priority in the range [0,1] which represents its relative 'importance'. If the user exceeds his/her quota for any resource, assign $PP = -\infty$.

SP — Scheduler priority. The scheduler may raise or lower a job's priority if its initial start time causes it to wait considerably above or below average.

The user and political priorities are given when insert(j) is called. The scheduler priority is also in the range [0,1], and will be discussed later. We assume that the priorities and quotas system is backed up by an accounting system that encourages users not to request a higher priority than they need.

The requirements from the scheduler that stem from priorities/quotas are:

X. Guarantee bounded delay: Although a job j may be rescheduled (pushed down the queue) many times, it is illegal to delay j by more than $j.s$ seconds during its entire lifetime.

XI. Maximize utilization: A schedule in which the total idleness (over all time slots) is smaller is preferable.

XII. Prefer jobs that arrived earlier. Specifically, If two waiting jobs j_1 and j_2 are identical but $j_1.t_0 < j_2.t_0$, then j_1 should run earlier.

XIII. Prefer jobs with higher priority. Specifically, If two waiting jobs are identical except $j_1.p > j_2.p$, then j_1 should run before j_2.

XIV. Fair share of delays: If one of two identical jobs j_1, j_2 must be rescheduled (their t_e increased because, for example, a higher priority job has arrived), and $j_1.s > j_2.s$, then j_1 should be delayed.

XV. Requirements XI, ..., XIV may conflict, therefore they can be given global weights $0 \leq \alpha_u, \alpha_t, \alpha_p, \alpha_f \leq 1$ which correspond to the weighed importance of the utilization, time, priority and fairness requirements.

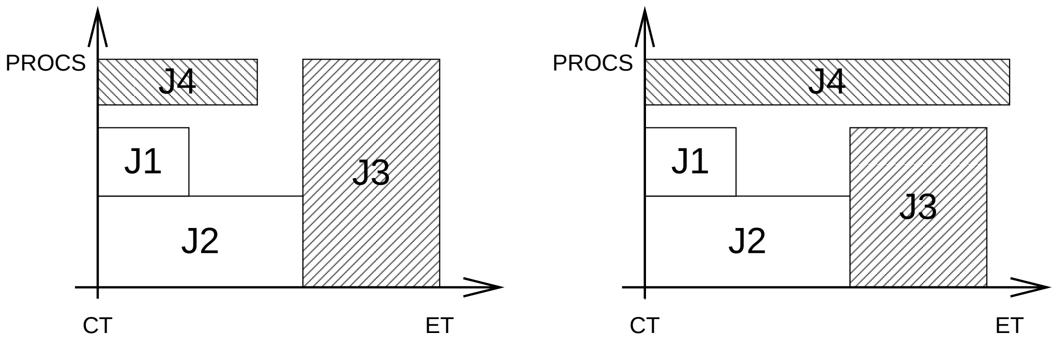

Figure 2: *EASY backfilling with criterion 1 (left) and criterion 2 (right). J1 and J2 are running, and J3 is first in queue. If J4 is submitted now it can be backfilled and start running immediately.*

Requirement X is a private case of requirement IX for slack based schedulers, and it specifies the role of slack as a guard against starvation. This requirement was added explicitly in order to stress its affect on the other priority requirements.

A priority scheduler has to support four conflicting goals, as requirement XV summarizes. Moreover, it is required that it would be possible to give different weights to each of these goals in order to give the administrators a stake in determining which services are more important to them.

3 The EASY and Conservative Schedulers

3.1 EASY Backfilling

The EASY approach to backfilling is an aggressive strategy: Jobs are allowed to skip ahead in the queue provided that they do not delay the first job in the queue [14, 17]. Influence on other jobs is not checked, and the result is that although there is no starvation, the wait time of a job is unbounded, and therefore the algorithm cannot commit as to when a job will run. The EASY scheduler supports requirements I through VIII (most of which simply introduce the scheduler's environment).

When a new job is submitted, its requirements are checked against the currently running jobs and the first job in the queue. The time in which the first job in the queue is going to run is called the shadow time; the idle nodes after the first queued job starts running are called extra nodes. The new job is backfilled (passes the first job in the queue and starts running immediately) if one of the following two conditions hold (Fig. 2):

1. It requires no more than the currently free nodes, and will terminate by the shadow time.

2. It requires no more than the minimum of the currently free nodes and the extra nodes.

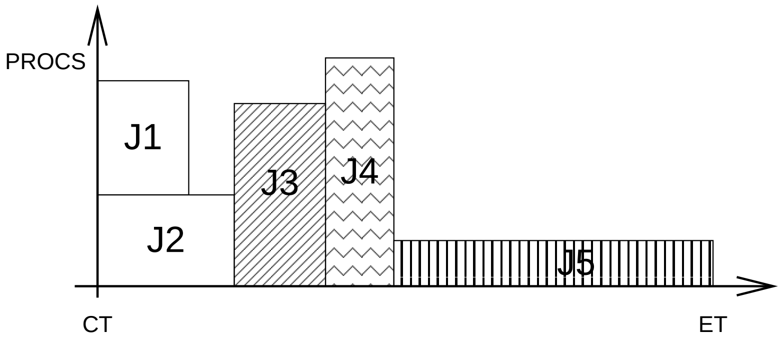

Figure 3: *Example profile with 3 queued jobs.*

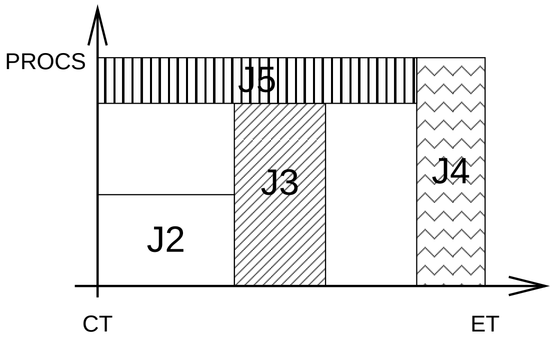

Figure 4: *Results of EASY backfilling after* $J1$ *terminates; compare with Fig. 3.*

When a job terminates (remove(j) is called), the algorithm loops over all waiting jobs in ascending order of arrival, and backfills all jobs that comply to one of the above criteria. When tick() is called (a new time slot begins, meaning that new jobs should start now, old job should terminate now, or both), the scheduler kills jobs that haven't terminated yet but were supposed to do so, and starts running jobs whose start time is the current time.

The EASY scheduler only checks that the backfilled job does not delay the first job in the queue, hence other jobs may be delayed. This leads to the fact that the waiting time of a job in an EASY queue is unbounded. Consider the profile depicted in Fig. 3, in which $J1$ and $J2$ are running and the queue is $J3$, $J4$ and $J5$ in this order.

Now assume that $J1$ terminates, and all waiting jobs are tested to see whether they could be used for backfilling. In this case $J5$ is suitable: It doesn't delay the first job in the queue, which is $J3$. However, it does delay $J4$, and the resulting profile is depicted in Fig. 4.

Two things should be noted here. The first is that since the runtime of $J5$ is in principle unbounded, then $J4$ may have to wait an unbounded time. Moreover, $J4$ may be delayed further by other jobs in a similar fashion. The second is that the backfilling nevertheless improved performance in this case: The total turnaround time ($ET - CT$) of the profile has decreased.

Although the EASY scheduler cannot make execution guarantees, it does not cause star-

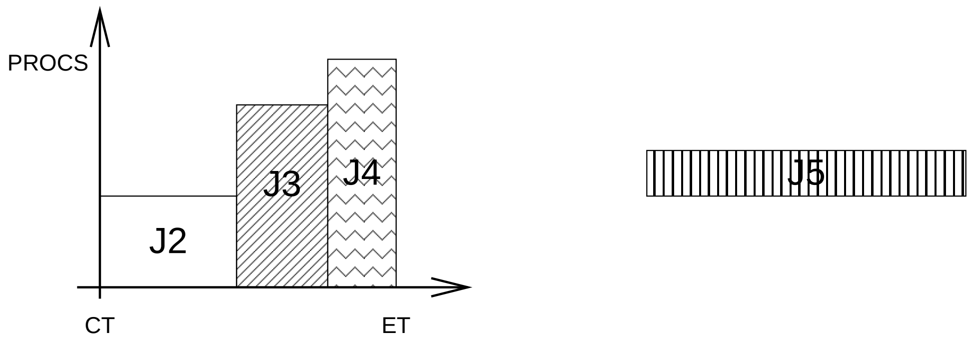

Figure 5: *Under conservative backfilling, if J5 is submitted into this profile it cannot start at CT (because it would delay J4). It would have to start at the current ET (compare with Fig. 4). Note that the execution guarantee to J4 comes at the expense of improved utilization and performance.*

vation. Since the termination time of the currently running jobs is finite, and the first job cannot be delayed, then it obviously must start running in finite time. When this happens, the second job in the queue will become first, and although it may have been delayed many times in the past, those delays will stop accumulating as soon as it becomes first. Therefore, the second job will eventually run as well, and the same argument can be used for the rest of the jobs in the queue.

3.2 Conservative Backfilling

The conservative backfilling algorithm takes a different approach to backfilling. It complies with requirements I through IX. This algorithm is usually the vanilla version assumed in the literature [4, 8], although it seems not to be used. The algorithm maintains a profile similar to that of EASY. When a new job arrives, the profile is scanned to find the first future time slot in which there are enough free processors to run the new job. The scanning then continues to see if the processors will be available until the new job terminates. If so, the job is assigned to this time slot. In the worst case, the job will be assigned to the last time slot (ET, in which all processors are free), and therefore the running time of this algorithm is linear in the size of the profile. The difference from EASY is that a job is backfilled only if doesn't delay any job in the queue, instead of looking only at the first job.

The insert(j) operation determines the start time of j, and this time can only be improved later, if jobs that were ahead of j in the queue are removed. The start time initially assigned to j is therefore an upper bound on its actual start time.

There are several alternatives to implementing remove(j) — the preferable option is to compress the current schedule. This can be done easily, for example, by removing and reinserting to the profile all the jobs scheduled for future time slots in the order in which they are currently scheduled. This can move small jobs way ahead of their originally assigned start time, and will never schedule a job for a later time than it originally was in (since in the worst cast, it will be reinserted back to its previous time slot). The complexity of this approach is $O(n^2)$ — for each job, call insert which runs in linear time.

3.3 Comparison

Conservative backfilling was tested against EASY using both a workload model and production workloads, using bounded slowdown as the performance metric [5]. The results show consistently that the two algorithms are practically identical in low and average workloads. Conservative Backfilling showed a slightly better performance in very high workloads, which are not practical. The conclusion is that the aggressive backfilling strategy used by EASY is not preferable, and that the execution guarantees made by the conservative scheduler make it preferable.

Further tests examined how the performance of the conservative scheduler is affected by the fact that users' estimates of runtimes are generally very poor. The results indicated that a certain degree of inaccuracy improved the algorithm: Overestimation of runtimes seems to give the scheduler more flexibility to backfill. This led to the idea that an improvement of conservative backfilling should explicitly incorporate this inaccuracy into the scheduler, by using the notion of slack. This was the basis for the priority scheduler.

4 The Priority Scheduler

4.1 Outline

Apart from supporting priorities, the priority scheduler is an extension of conservative backfilling. It differs from it by its backfilling strategy, and, of course, by supporting priorities and adhering to all I through XV requirements. The scheduler assigns each waiting job some slack, which measures the maximal amount of time that the job may be delayed beyond its initially assigned start time. While EASY allows backfilling a job if it doesn't delay the first job in the queue, and conservative backfilling allows backfilling a job if it doesn't delay any job in the queue, the priority scheduler allows backfilling a job if it doesn't delay any job in the queue by more than that job's slack. When a job is delayed or sped up its slack changes accordingly. This way the scheduler enjoys more flexibility than conservative scheduling, but still retains the execution guarantee requirement IX, because the initial start time of a job plus its initial slack is an upper bound on its actual start time.

The priority backfilling algorithm maintains a profile similar to that of conservative backfilling. When a new job is inserted, any other jobs may be rescheduled in order to optimize the overall utilization, subject to constraints of no preemption, execution guarantees and priorities. The algorithm gives a price to every possible new schedule, and chooses the cheapest one. The price of a schedule is the sum of prices of jobs, and the price of each job is the product of its delay by the number of processors it uses.

Consider the profile in Fig. 6, in which $J1$ and $J2$ haven't started running yet. Assume

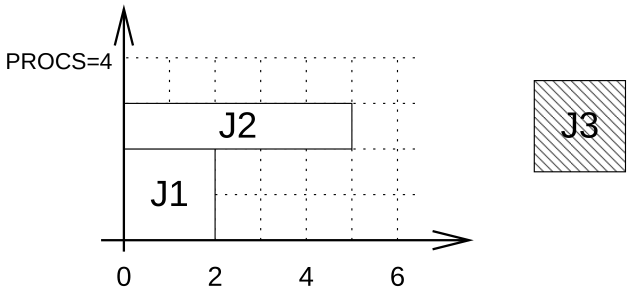

Figure 6: *Example profile for explanation of cost considerations.*

that $J3$ arrives. $J3$ requires four processor seconds in total. There are three possible schedules to consider: We can schedule $J3$ at $t = 2$. This will cost us two seconds times two processors (four processor seconds, or p.s.). We can also schedule $J3$ at $t = 0$ and move $J2$ to $t = 2$. This way we save the four p.s. for $J3$, but we have to pay two for $J2$ (one processor times two seconds), so the total cost of this option is two. A third option is to schedule $J3$ at $t = 0$ and move $J1$ to $t = 2$. The cost of this schedule is four. The 'cheapest' alternative is the second one, so it will be chosen, and utilization is improved. Note that we don't consider obviously redundant schedules such as running $J3$ at $t = 5$.

But, it is possible that at the current time, $J2$ has already been delayed for an hour and pushed twenty times down the queue, while $J1$ is a new job. In order to prevent the starvation of $J2$ we may decide that one of the two other alternatives is preferable. The mechanism to prevent starvation and ensure bounded delay (requirements VII and IX) is slack: Whenever a job is submitted, it is assigned a slack that depends on its priority and system parameters. Whenever a job is delayed by x seconds, its slack decreases by x. When a job is moved up by x seconds, its slack increases by x. A job cannot be delayed in a manner that would cause its slack to be negative. We therefore have an upper bound on the start time of j, which is $j.t_e + j.s$.

Another issue that has to be addressed is priorities: Perhaps the priorities of $J1$ and $J2$ are much higher than $J3$'s. In such a case, we would want to schedule $J3$ at $t = 2$. Another possible case may be a high priority of $J2$, but a low priority of $J1$ (relative to that of $J3$). In this case, it may be best to schedule $J3$ at $t = 0$ and $J1$ at $t = 2$.

A fourth and final consideration are the priority weights as defined in requirement XV. The algorithm can be tuned so that some of the previous considerations will be more important than others are. For example, we may decide that utilization is much more important than priorities (assign α_u close to 1 and α_p close to 0), and in this case we may decide to delay $J2$ in favor of $J3$ although $J2$ has a higher priority.

4.2 The Algorithm

To define the algorithm we need to define the three operations insert, remove, and tick.

insert(j) : Loop over all possible schedules (conceptually) and give a price to every possible schedule. The price of scheduling j at $j.t_e$ and delaying $j_1, \ldots, j_k$ by $t_1, \ldots, t_k$ seconds is infinity if it causes one of the j_i to achieve a negative slack (if $j_i.s - t_i < 0$), and otherwise it is:

$$price = (j.t_e - CT)^{\alpha_t} \times j.n^{\alpha_u} + \sum_{i=1}^{k} cost(j_i, t_i, j)$$

The cost function determines the cost of moving j_i by t_i seconds in favor of j, and will be discussed later. Cost will be negative if t_i is negative (i.e. if the job is being moved up rather than being delayed). Note that there is always at least one schedule of finite price: Do what conservative backfilling would have done. Once we know what the cheapest schedule is, we adjust the slacks of the rescheduled jobs (for all $i = 1..k$, $j_i.s = j_i.s - t_i$), and start running jobs that should start now.

Remove(j) : Remove j from the profile, and then loop over all possible schedules, and find the cheapest one, exactly as in insert. Here we expect a negative price — a profit.

Tick() : We can choose to kill jobs that were supposed to terminate but didn't, as done in EASY and conservative backfilling. A better possibility is described in Section 4.7.

The following two sections describe how the priority and slack of a new job are determined, and the $cost(j_i, t_i, j)$ function that prices reschedules of jobs. Section 4.5 addresses the complexity problem: Since there is an exponential number of possible schedules of k jobs (which is $k!$), it is not practical to check each of them in a naive manner. The last section summarizes the algorithm and its parameters.

4.3 Calculating Priority and Slack

The priority $j.p$ of a job is composed of its user, political and scheduler priorities. The user and political priorities are given when the job is submitted, but the scheduler priority is not. The scheduler priority is a number in the range [0,1], and we wish it to be higher when the job's initial wait time is longer. This way the scheduler can ensure that if a job was initially scheduled very early compared to the average wait time, its priority can be lowered and its slack increased in favor of less fortunate jobs, so that more fairness could be achieved later.

At first, we assign $SP = \frac{1}{2}$ for all jobs. This means that we want all jobs to wait exactly the average wait time. Then, we calculate the job's priority:

$$j.p = \frac{j.UP + j.PP + j.SP}{3}$$

Afterwards, we calculate the job's initial slack:

$$j.s_0 = \begin{cases} (1 - j.p) \times SF \times AWT & \text{if } j.p > -\infty \\ \infty & \text{otherwise} \end{cases}$$

The constant SF is the slack factor of the system — it's another parameter of the algorithm. In section 4.1 we saw that giving jobs slack can improve utilization; however, giving jobs too much slack makes the upper bound on delays meaningless. The slack factor gives a way to express an opinion about this tradeoff. The case in which $j.p = -\infty$ occurs when the user exceeds his or her quota for one of the system's resources. The administrator then submits j with $j.PP = -\infty$ which causes the job to have infinite slack. This means that this job can suffer an unbounded delay, and will only run when it's not disturbing any other job. Once we have a priority and an initial slack for j, We can compute the price of each possible schedule. After deciding where it's best to execute the new job, its start time $j.t_e$ will be defined. Then, we recalculate:

$$j.SP = \min\left\{\frac{j.t_e - CT}{2 \times AWT}, 1\right\}$$

Note that $j.SP$ is zero if j doesn't wait at all, $\frac{1}{2}$ if it waits the average wait time, and 1 if it waits twice the AWT or more.

It may be wise to use several AWT's for several sizes of jobs: For example, the average wait time in a system may be two hours, but it is unreasonable that a ten seconds long job would have to wait that much. This option was not tested, and the overall average system wait time was used in all simulations.

Once we have the new $j.SP$, we recalculate $j.p$ and $j.s_0$ according to this new $j.SP$ value, and use the new priority and slack values from now on. This recalculation takes place only once — we do not reschedule j after recalculating its priority and initial slack. These new values will only have an effect in case of future backfilling attempts.

4.4 Calculating Cost

The cost of moving j_i by t_i seconds in favor of j depends on the utilization gain or loss that the move causes (requirement XI), the relative priorities of the two jobs (XIII), and the percentage of j_i's slack that was already used (XIV). The preference of earlier jobs (XII) is contained in the fact that when two schedules are equally priced, we'll choose the schedule with the least number of moved jobs. Since delaying the new jobs doesn't 'count' as a delay, older jobs have an advantage over the new one. By requirement XV all the above considerations are weighed by α_u, α_t, α_p, and α_f. To conclude we get:

$$cost(j_i, t_i, j) = \begin{cases} j_i.n^{\alpha_u} \times t_i^{\alpha_t} \times \left(\dfrac{j_i.p}{j.p}\right)^{\alpha_p} \times \left(\dfrac{j_i.s_0}{j_i.s}\right)^{\alpha_p \times \alpha_f} & \text{if } t_i \le j_i.s \\ \infty & \text{otherwise} \end{cases}$$

As an example that demonstrates how cost() behaves, consider the profile discussed in Section 4.1. $J1$ and $J2$ have been scheduled, and $J3$ arrives. There are three schedules to consider (Fig. 7): S1 in which $J3$ is scheduled at $t = 2$, S2 in which $J2$ moves to $t = 2$ and $J3$ is scheduled at $t = 0$, and S3 in which $J1$ moves to $t = 2$ and $J3$ is scheduled at $t = 0$.

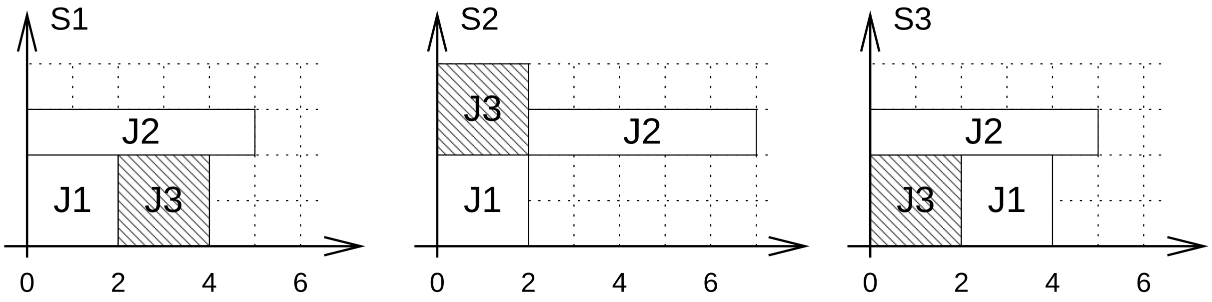

Figure 7: *The 3 possible schedules in the cost example.*

The following tables calculate the price of each schedule under different priorities. A slack reading 'x/y' means that the job's current slack is x, and its initial slack was y.

These results assume $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$:

J1	J2	J3	Price(S1)	Price(S2)	Price(S3)	Best
s=10/10 p=0.5	s=10/10 p=0.75	s=10/10 p=0.5	4	3	4	S2
s=10/10 p=0.15	s=10/10 p=0.9	s=10/10 p=0.3	4	6	2	S3
s=5/10 p=0.15	s=10/10 p=0.9	s=10/10 p=0.3	4	6	4	S1
s=1/10 p=0.15	s=1/10 p=0.9	s=10/10 p=0.3	4	∞	∞	S1

This is how the first scenario changes if $\alpha_u = \frac{1}{2}$:

s=10/10 p=0.5	s=10/10 p=0.75	s=10/10 p=0.5	2.828	3	2.828	S1

This is how the second scenario change if $\alpha_p = \frac{1}{2}$:

s=10/10 p=0.15	s=10/10 p=0.9	s=10/10 p=0.3	4	2.88	3.17	S1

The first scenario demonstrates that if all priorities are equal or even if $J2$'s priority is slightly higher, the utilization gain in delaying $J2$ by two seconds rules to delay it in favor or $J3$. The second scenario demonstrates what happens if $J2$'s priority is very high and $J1$'s priority is very low: $J1$ will be delayed in favor of $J3$, instead of $J2$. The third scenario demonstrates fairness: It is the same as the second except the fact $J1$ has already been delayed (its current slack is only half of its initial slack). Fairness leads to leaving $J1$ where it is, and scheduling $J3$ at a later time. The fourth scenario is just a reminder to the execution guarantees given by the priority scheduler: Schedules which delay jobs by more than their slack can never be chosen. The fifth and sixth scenarios demonstrate the influence of the importance factors on scheduling decisions: Both demonstrate how the scheduler's decision is changed due to the different weight it gives to each requirement.

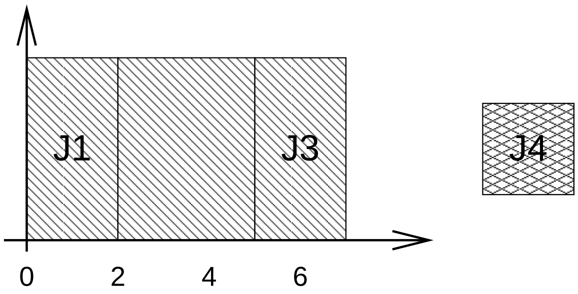

Figure 8: *Example schedule for complexity considerations.*

4.5 The Complexity Problem

Basically, the problem at hand is a scheduling problem of jobs of variable duration with no preemption, variable deadlines, a resource constraint on processors, and variable costs for delaying a job. Not surprisingly, this is a NP-hard problem.

The following exponential algorithm finds the optimal schedule inserting a job j into a profile: For each time slot ts from CT to ET, do the following: Delay all jobs from ts to the profile's end by $j.t$, and insert j as the only job scheduled for time ts. Then, compress the schedule in every possible way, and remember the cheapest schedule.

If k jobs were delayed, there may be $k!$ schedules to check, because every permutation of the delayed jobs defines an order in which the delayed jobs can be compressed, and each such permutation can create a different schedule with a different price. However, in many cases only few of these permutations will be worth checking. The example below will demonstrate these points.

After looping over all time slots, the algorithm chooses the cheapest schedule of all. Note that when trying to assign the new job to a time slot we don't have to try to change jobs that are scheduled for earlier slots, since we assume that the schedule was optimal before the new job was submitted.

Note that remove(j) takes exactly the same time insert(j) takes, since remove(j) simply deletes j from the profile and then inserts a dummy job of zero time and processors.

Consider the schedule in Fig. 8 to which J4 should be inserted. The above algorithm will iterate four times. Note that no real compression is done here, but still all the following schedules will be checked:

Iteration 1, J4 at t=0: (J1,J2,j3), (J1,J3,J2), (J2,J1,J3), (J2,J3,J1), (J3,J1,J2), (J3,J2,J1).

Iteration 2, J4 at t=2: (J2,J3), (J3,J2) (J1 stays at t=0 in both cases)

Iteration 3, J4 at t=5: One schedule to check in which J3 runs at t=6

Iteration 4, J4 at t=6: No delayed jobs.

As many profiles in practice can be as long as a hundred jobs, this algorithm is not practical. Also, note that in the first iteration, because $J1$, $J2$ and $J3$ capture all processors, we don't really need to traverse all permutations: We know that the most expensive jobs (for which $cost(j_i, 1, J4)$ is largest) should come first; or, most exactly, we know that the jobs with the largest $cost(j_i, 1, J4) \times j_i.t$ should come first. This is a good heuristic in general, since we expect the cheaper schedules to be the ones in which the more expensive jobs are delayed less.

4.6 Complexity Reduction Heuristics

Several heuristics can be offered for choosing which permutations are checked. These heuristics choose one permutation to consider in each iteration — this approach still requires $O(n^3)$ time to insert or remove a job.

Ascending Scheduled Time (AST) — Sort the delayed jobs by their scheduled time (before the delay), and check this permutation only. This heuristic tries the most to preserve the current ordering of the delayed jobs.

Ascending Arrival Time (AAT) — Sort the delayed jobs by their $j.t_0$, and check this permutation only. This benefits jobs that are already waiting longer.

Descending Utilization (DU) — Check the permutation that results from sorting the delayed jobs by descending $j_i.n \times j_i.t$. This heuristic acknowledges that jobs with higher utilization are likely to be delayed more because of their size, and therefore they should be scheduled first.

Descending Cost (DC) — Sort the delayed jobs by descending cost to delay the jobs for one second, and check this permutation only. This way 'expensive' jobs will be delayed less, and priorities and fair share issues will also be considered.

Descending Priority (DP) — Reschedule first the jobs whose priorities are highest. For the common case of equal priorities, a secondary sort by ascending arrival time was also used. This heuristic is expected to be more responsive to priorities.

4.7 Handling Unfinished Jobs

We have discussed the insert(j) and remove(j) operations in detail, but haven't touched tick() so far. In most implementations this is done trivially: Kill all jobs that were supposed to end by now but didn't, and start executing all jobs that are supposed to start now. A slight improvement is not to automatically kill jobs if they are not taking anyone's place. In both cases, A job that has been running for a day on 64 processors could be killed only because it needed ten minutes more than it declared it would. Unless a checkpoint system was used, which in many cases requires extensive programmer intervention, the job will be lost. This causes a utilization loss of one day times 64 processors, because nothing was actually done on them.

The solution is to provide accurate time estimates, of course, but most users are far from doing so. By using priority scheduling, we can quantify the utilization loss caused by killing a job, and can decide whether it would be best to give an unfinished job more time at the expense of others, or kill it. This way we can enhance utilization further, by letting more jobs get finished, and still satisfy all of the algorithm's requirements.

When tick() is called and it recognizes an unfinished job j, it calculates the price of killing it, which is $j.n^{\alpha_u} \times j.t^{\alpha_t}$. Then, it creates a new job with the same priority as j, same number of processors, and a duration of 10% of j's declared duration $j.t$. The minimal price for scheduling the new job at the current time is computed (there's no point in trying to schedule it later, since j can't be preempted). If this price is less than the price of killing j, then j gets an extension of 10% of its declared time, and the necessary changes in the original schedule are made. Otherwise, j is killed.

This simple to use method satisfies all the priority requirements, including execution guarantee, since if a job is supposed to start now and it can't be delayed any longer (zero slack), than the price of giving j the extension will be infinity. In other cases most jobs will get the extension, because the price of killing a job is usually very high.

This 'Tick Strategy' has not been tested within this project. The reason is that all production systems as for today kill jobs that do not terminate by their announced estimated runtime, or at the better case allow them to run unless they are delaying another job. Hence, there are no records of the runtime of jobs that continued running even though they delayed other jobs. This idea cannot be simulated.

4.8 Summary of Parameters

The priority scheduler can be altered by several parameters whose roles have been scattered in the previous sections. Before turning to the experimental result, we'll review each of them and their expected influence on the algorithm's behavior.

- According to requirement XV, The utilization, time, priority and fairness considerations can be given weights for their relative importance, in the form of $0 \leq \alpha_u, \alpha_t, \alpha_p, \alpha_f \leq 1$. See Section 4.4.

- The Slack Factor measures how many AWTs we assign as the initial slack of a job whose priority is zero. The SF determines how much flexibility we give the algorithm on the expense of a tight execution guarantee: A low SF will give users a tight bound on the start time of their jobs, but will give the algorithm little flexibility to backfill. A high SF will causes the opposite behavior. When SF=0 the priority scheduler collapses to conservative backfilling. See Section 4.3.

- One of several heuristics may be used to decide which permutations are priced at each iteration of the backfilling stage of the scheduling. Although some of the heuristics are clearly superior to others, the situation is not always clear: As the experimental results will demonstrate, different heuristics are preferable for different circumstances. See Section 4.6.

The algorithm requires two other system parameters, which are the total number of processors PROCS and the system's average wait time AWT. The AWT is usually known for every production system or can be derived using existing logs and maintenance statistics. Using a false AWT in order to change the algorithm's performance has the exact same affect as that of changing the slack factor: In fact, using a slack factor of n means that the algorithm should consider the system slack time to be $n \times$AWT. Therefore it is advisable to use a realistic AWT and adjust the SF according to the administrators' desires.

Every job is submitted along with a requested number of processors, expected runt user priority and political priority. It is assumed that users have an incentive to give correct runtime estimates, and that high priorities are not used when unnecessary. However, the algorithm does not rely on correct estimates in any way.

5 Experimental Results

5.1 The Simulator

A simulator for testing the conservative and priority schedulers was written in C++ using the Borland C++ Builder environment. The simulator implements all aspects of both algorithms. Jobs that exceeded their declared runtime were immediately killed.

The logs used for the simulations were the Swedish Royal Institute of Technology (KTH) files from September 1996 to August 1997. The average number of jobs per month is 2357 and the average wait time of the system with 128 processors, under the conservative scheduler (which is nearly the same as the average wait time obtained by EASY scheduling) is 2401 seconds. The logs contain, for each job, the number of processors and both the estimated and actual runtimes of the job. Hence the tests do not require a model of the human ability to estimate runtimes, and use actual numbers.

Using a Pentium 200Mhz with 64MB of memory, the conservative simulator took several seconds to complete a month's schedule, and the priority scheduler took about a minute to do the same job.

5.2 Equal Priorities

The following simulations tested the performance of the priority scheduler against those of the conservative scheduler, assuming that all jobs have equal user and political priorities and an unlimited quota. The following parameters were used for the priority scheduler: $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$, SF = 3, AWT = 2401, Heuristic = AST.

The results in Table 1 show an average reduction of 16.5% in wait time over conservative scheduling. This result implies that priority scheduling is useful as is, even without using the option to assign priorities. The overall average was calculated correctly — not as the

Month	Average Wait Time		Improvement
	Conservative	Priority	
September	440.6	440.6	0.0%
October	3062.7	2504.3	18.2%
November	4887.0	3774.6	22.8%
December	2157.2	1904.2	11.7%
January '97	3302.3	2531.9	23.3%
February	2624.3	2373.9	9.5%
March	2644.8	2196.9	16.9%
April	2220.3	1905.2	14.2%
May	2093.6	1779.2	15.0%
June	1443.2	1282.0	11.2%
July	399.9	354.1	11.5%
August	1941.5	1670.8	13.9%
Average	**2401.44**	**2004.46**	**16.5%**

Table 1: *Simulation results of priority backfilling compared to conservative backfilling.*

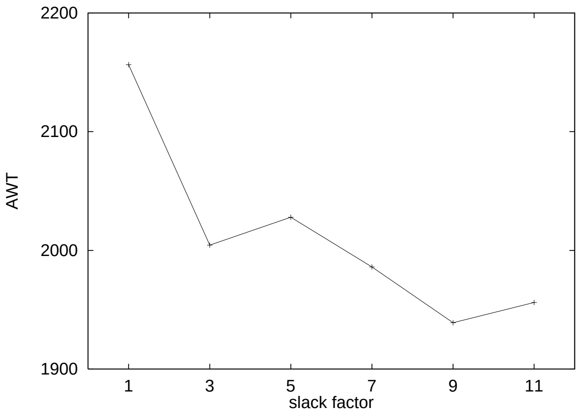

Figure 9: *Dependence of AWT on the slack factor (SF).*

average of the rows above it, but as the quotient of the total wait time by the total number of jobs throughout the year.

The same simulation using larger slack factors modestly improved the results (19.25% average improvement for SF=9, see Fig. 9). However, giving too much slack to jobs increases the extent to which jobs can be starved — for example, a slack factor of nine means that a job can be delayed up to nine times the average wait time. Therefore a small slack factor, which still gives significant improvements, seems like the best tradeoff.

The average wait time used for all months was the same — the yearly average. If the average wait time used is adjusted for each month, the performance of the priority scheduler is improved by a few more percents. A good estimation of the average wait time is crucial:

Using a very small number causes the algorithm to collapse to conservative scheduling (too little slack implies an inability to delay jobs), and using a very large number increases the chance of (bounded) starvation.

During the simulation, all jobs were assigned a user priority of zero and a political priority of zero. Giving all jobs equal but higher user or political priorities degraded the algorithm's performance, probably because it dims the effect of the scheduler priority (which raises the priority of jobs that initially have to wait longer). Except for September '96 in which only 85 jobs were executed, the algorithm performs well on a variety of workloads. Logs of other months included between 1924 and 4060 jobs.

The ascending scheduled time heuristic used to obtain the above results is the one that gives the best performance. Ascending arrival time gave an average improvement of 13% over conservative scheduling, descending priority gave an average improvement of 11.7%, descending cost gave 9.2% and descending utilization gave 8.1%. Notably all heuristics improve conservative scheduling (and, hence, EASY as well).

5.3 Differential Priorities

To test how well the algorithm schedules jobs with a higher priority then others, the following mechanism was used. For each of the twelve monthly logs, each fifth job was given a user priority of 1.0 and a political priority of 1.0, and all other jobs were given zero for both the user and political priorities. This simulates a scenario in which a user or a group have a considerably higher priority than others, and they submit jobs uniformly. It was implicitly assumed that the 'fifth jobs' distribute identically compared to other jobs regarding wanted time, used time, processors and so on. The results, summed on the table at the bottom of the page, were received using the same parameters as in the previous section: $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$, SF = 3, AWT = 2401, Heuristic = AST

The results (in Table 2) show that the priority change decreased the average wait time of the preferred jobs by 2.5% but increased the wait time of the other jobs by 3.9%, compared to the best results achieved with equal priorities. In total, assigning different priorities had, as expected, a cost — the average wait time of the entire system rose by 11.1%, from 2004.5 to 2226.2 seconds. This is still an improvement over conservative scheduling and EASY, and the decision of whether to support priorities or maximize system performance is in the administrators' hands. In any case, the priority scheduler outperforms conservative scheduling and EASY.

The above simulations, as mentioned above, gave a random group of 20% of the jobs a user plus political priority of two against zero to the other jobs. Other simulations using a smaller gap exhibited a smaller, but still positive, difference in the average wait time between the groups. Fig. 10 shows the average wait time of the entire system as a function of the gap. With the exception of the maximal gap, the overall AWT was very close to the optimum (which is obtained with zero gap, e.g. in equal priorities). This means that in most cases the prioritization does not lead to a general degradation of service.

Other simulations tested the effect of increasing the importance of priority in contrast

Month	Equal Priorities	Unequal Priorities		
		All Jobs	High Pri Jobs	Low Pri Jobs
September	440.6	440.6	0.0	550.8
October	2504.3	2783.7	2295.6	2906.0
November	3774.6	4797.7	4460.2	1882.3
December	1904.2	2026.0	1814.8	2078.8
January '97	2531.9	2777.4	2603.8	2820.8
February	2373.9	2728.7	2369.2	2818.7
March	2196.9	2220.9	1687.2	2354.6
April	1905.2	2056.2	1797.4	2120.9
May	1779.2	1889.3	1491.9	1988.7
June	1282.0	1374.7	1323.3	1387.6
July	354.1	372.6	320.2	385.8
August	1670.8	1847.1	1817.5	1854.5
Average	**2004.46**	**2226.17**	**1955.28**	**2294**

Table 2: *Simulation results of priority backfilling with differential priorities.*

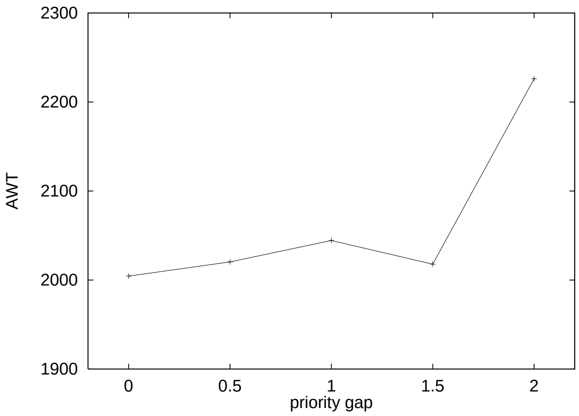

Figure 10: *Dependence of the overall AWT on the priority gap.*

with time, utilization and fairness, for example by assigning $\alpha_p = 1$, $\alpha_u = \alpha_t = \alpha_f = 0.2$. These tests indicated a slight increase in the wait time gap between the preferred and the regular groups, but also exhibited a considerable degradation of the average wait time of both groups. In several cases, the average wait time of the preferred group was worse than that of the $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$ setting. It seems that a high α_u and α_t are crucial to the effectiveness of the scheduler.

The results that were achieved using the descending scheduled time heuristic were actually only the second best. Descending arrival time performed marginally better. Descending priority is also very close, hinting that this performance level is probably the best that can be expected. The results of all five heuristics are summarized in Table 3. They were all

	Ascending Sched Time	Ascending Arrival Time	Descending Cost	Descending Utilization	Descending Priority
Equal Pri AWT	2004.5	2088.5	2179.9	2206.0	2120.0
Unequal Pri AWT	2226.2	2223.3	2250.1	2279.8	2228.5
Performance Loss[1]	11.1%	10.9%	12.3%	13.7%	11.2%
High Pri Jobs AWT	1955.3	1952.6	1988.1	2048.9	1962.3
Low Pri Jobs AWT	2294.0	2291.1	2315.7	2337.6	2295.1
Gap[2]	338.7	338.5	327.6	288.7	332.7

[1] The ratio of increase in the total average wait time due to the differential priorities. This increase is measured relatively to the AST result for equal priorities, which was the best one, and not relatively to the heuristic's own equal priorities performance.

[2] The average difference between the low priority group and the high priority group's average wait times.

Table 3: *Simulation results comparing various complexity reduction heuristics.*

tested using the full set of logs. All numbers are yearly averages, in seconds.

6 Conclusions

The many production installations of EASY around the world prove that backfilling is advantageous over FCFS allocation of processors to jobs. The ability to backfill increases the overall system performance by being more responsive to short jobs while preventing the starvation of long batch jobs. However, while EASY takes an approach of aggressive backfilling, a more conservative approach exhibits the same performance and retains the predictability of the FCFS scheduler.

Further simulations on the conservative scheduler pointed out that inaccurate estimates of runtimes, namely overestimation, improve the algorithm's performance. This led to the idea of slack based backfilling: If the estimated runtime of a job is $j.t$, we should be able to delay it by $j.t + j.s$, where the job's slack $j.s$ determines how flexible we let the backfilling be. Such an approach also means that, in contrast with EASY or conservative backfilling, we would have to consider many alternative legal backfilling combinations each time the profile changes.

Both of these properties combine nicely with the support in priorities: Low priority jobs could be assigned more slack, meaning that we allow delaying them more than important jobs; Priorities can also be taken into account when deciding between the many available schedules. As the priority scheduler was developed it was clear that besides the added functionality of prioritization, it would not be a surprise if the algorithm would improve the existing backfilling algorithms even in systems which don't use priorities.

Indeed, the experimental simulations show that the priority scheduler is better than both EASY and conservative backfilling on the tested production systems. The scheduler encapsulates both extra flexibility, required to improve the conservative scheduler, and sufficient execution guarantees, required to assure that long jobs do not starve and affect the entire system's performance.

The differential priorities simulations show that while the high priority jobs indeed exhibited a lower wait time than in the equal priorities case, the inequality in priority distribution has a cost in terms of the overall system performance: The low priority jobs had to wait more than before. However, This is intuitive and expected, and it should be stressed that the simulations show that the low priority jobs scheduled by the priority scheduler waited on average less than in EASY.

The scheduler includes a relatively high number of parameters, and it should be adjusted to the system it's installed on. Two delicate parameters seem to be the slack factor and the utilization and time importance weights; setting any of them to a very low value causes the scheduler to collapse to the performance of EASY and conservative backfilling, and also decreases the gap between high and low priority jobs. Using a very low average wait time parameter led to similar mediocre results. This leads to the conclusion that while the algorithm supports priorities and high predictability, it is crucial to give a considerable weight to utilization in pricing profiles, because this affects the performance of both high and low priority jobs.

Although backfilling was originally developed for the SP2, and was so far tested using workload traces from SP2 sites only, it is applicable to any other system using variable partitioning. This includes most distributed memory parallel systems in the market today.

Acknowledgements

This research was supported by the Ministry of Science and Technology. Thanks to Lars Malinowsky of KTH for his help with the workload traces. This trace is now available from the parallel workloads archive at http://www.cs.huji.ac.il/labs/parallel/workload/.

References

[1] D. Das Sharma and D. K. Pradhan, "*Job scheduling in mesh multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.

[2] D. H. J. Epema, "*An analysis of decay-usage scheduling in multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 74–85, May 1995.

[3] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[4] D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[5] D. G. Feitelson and A. Mu'alem Weil, "*Utilization and predictability in scheduling the IBM SP2 with backfilling*". In 12th *Intl. Parallel Processing Symp.*, pp. 542–546, Apr 1998.

[6] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[7] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. 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. Lect. Notes Comput. Sci. vol. 1291.

[8] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[9] G. J. Henry, "*The fair share scheduler*". *AT&T Bell Labs Tech. J.* **63(8, part 2)**, pp. 1845–1857, Oct 1984.

[10] S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[11] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual.* Order number 312261-002, May 1992.

[12] P. Krueger, T-H. Lai, and V. A. Dixit-Radiya, "*Job scheduling is more important than processor allocation for hypercube computers*". *IEEE Trans. Parallel & Distributed Syst.* **5(5)**, pp. 488–497, May 1994.

[13] R. N. Lagerstrom and S. K. Gipp, "*PScheD: political scheduling on the CRAY T3E*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 117–138, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[14] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[15] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[16] P. Messina, "*The Concurrent Supercomputing Consortium: year 1*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 9–16, Feb 1993.

[17] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 41–47, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[18] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 1–11, USENIX, Nov 1994.