A Historical Application Profiler for Use by Parallel Schedulers

Richard Gibbons
gibbons@cs.ubc.ca

University of British Columbia
201-2366 Main Mall, Vancouver, B.C., Canada V6T 1Z4

Abstract. Scheduling algorithms that use application and system knowledge have been shown to be more effective at scheduling parallel jobs on a multiprocessor than algorithms that do not. This paper focuses on obtaining such information for use by a scheduler in a network of workstations environment.
The log files from three parallel systems are examined to determine both how to categorize parallel jobs for storage in a job database and what job information would be useful to a scheduler. A Historical Profiler is proposed that stores information about programs and users, and manipulates this information to provide schedulers with execution time predictions. Several preemptive and non-preemptive versions of the FCFS, EASY and Least Work First scheduling algorithms are compared to evaluate the utility of the profiler. It is found that both preemption and the use of application execution time predictions obtained from the Historical Profiler lead to improved performance.

1 Introduction

Many theoretical and modeling-based studies indicate that knowledge of the characteristics of parallel applications can improve the performance of scheduling algorithms [MEB90, PD89, GST91, MEB91, Wu93, PS95, AS97, BG96, PS96]. However, much less research has focused on practical ways of obtaining such application knowledge. Sevcik [Sev94] proposes simply having the user provide estimates of application characteristics. However, this is inconvenient for users and the accuracy of data is not assured. As a result, more technical solutions may be warranted.

Kumar [Kum88] proposes a tool that measures application parallelism by inserting statements into application code. However, the tool he proposes requires special versions of applications to be created and run to determine application characteristics. This is a great inconvenience for users.

Another approach is to measure application characteristics at run time. Dusseau, Arpaci and Culler [DAC96] use this method with their *implicit scheduling* technique for distributed time-shared workloads. Local schedulers use the communication and synchronization events implicit in parallel applications to estimate load imbalances. The local schedulers are able to determine from this

data when to schedule parallel applications so that multiple processes of a job have a high probability of being scheduled simultaneously.

Nguyen, Vaswani and Zahorjan [NVZ96b, NVZ96a] use a combination of code instrumentation and hardware monitors to determine run time characteristics of iterative applications. By varying the processor allocations over several iterations of a loop, the scheduler can determine application characteristics. Although Nguyen, et. al. show the performance of schedulers using this method to be very good, thus far, this strategy requires that the application programmer instrument his code.

An alternative method of determining application characteristics is to keep a historical database containing data on every job that has been run on the system. The database could then deduce the resource usage of future jobs from the past usage. Despite results of workload characterization studies that seem to indicate that this approach holds some potential [PBK91, FN95, Hot96, HSO96], up to now, nobody has implemented this strategy. This paper addresses this issue by creating a Historical Profiler.

Through the examination of several production parallel systems, Section 2 justifies the use of and provides insight into an appropriate design of a Historical Profiler. Section 3 describes the features of the Historical Profiler, and discusses the issues associated with implementing such a profiler in a network of workstations (NOW) environment. Next, Section 4 proposes several scheduling algorithms and the workload to use to evaluate the performance of the Historical Profiler. Section 5 analyses the results of experiments discussed in the previous section. It notes that backfilling, preemption, and knowledge of application characteristics generally lead to reduced mean response times in the algorithms examined. Furthermore, it finds that in many cases, the performance of algorithms using the Historical Profiler is relatively close to the performance of the same algorithms using perfect information. Finally, Section 6 summarizes the findings and discusses future research.

2 Workload Characterization

Previous results [MEB90, Sev89, PD89, GST91, MEB91, Sev94] have indicated that knowledge of job characteristics can improve the performance of parallel schedulers. However, these results do not necessarily imply that such knowledge can be derived from the historical resource usage of applications, and they do not provide any indication of effective ways of classifying jobs to obtain this knowledge. To address these issues, it is necessary to examine the workload on production parallel systems. This analysis is used to guide the design of the Historical Profiler.

In order for the historical information to be of use in scheduling, it is necessary to first show that the duration of jobs can be predicted more accurately when historical information is used than when it is not. To determine if the job durations are predictable, we will use the coefficient of variation [1], or CV,

[1] The coefficient of variation is the ratio of the standard deviation to the mean.

of the service time distribution. Jobs can be classified into categories based on attributes such as the executable name or selected queue. It is desirable to find the attributes of jobs that lead to low runtime CVs for these categories. If the runtime CV of a given category is lower than the system-wide runtime CV, it implies that the historical knowledge may be used to provide more accurate estimates of runtimes than estimates that do not use historical knowledge. The lower the CV, the more accurate the estimates are likely to be. We will focus on choosing what attributes to use to classify jobs.

There have been only a few detailed workload characterization studies of parallel systems [CS85, PBK91]. Furthermore, the studies that do exist do not focus on different ways of classifying jobs, with the exception of Feitelson and Nitzberg's [FN95] analysis of the workload of the 128-node NASA Ames iPSC/860 hypercube. Feitelson and Nitzberg classify jobs by name, user, and number of processors, and discover that in the majority of cases where applications were run more than once, the coefficient of variation, or CV, of runtimes is less than one. Meanwhile, the overall systemwide CV is 3.56 during the day, and 2.11 during the night. This implies that in this system, historical knowledge could be used to predict future resource usage. Furthermore, it means that using name, user, and number of processors is a reasonable way of categorizing jobs.

To confirm that these results hold for other systems, we examine the Cornell Theory Center (CTC) IBM SP2 and network of workstations (NOW) sites at NASA Lewis and an anonymous university (which we shall refer to as University1).

Hotovy, et al. [Hot96, HSO96] have already done much analysis of the workload on the CTC IBM SP2. They have examined it in terms of utilization and user-node time. They find that the average utilization is only 60%[2]. They also find that half the jobs are serial, but these jobs account for only 8.6% of the user-node time. Hotovy, et. al. go on to examine the relationship between the number of processors and the job duration. They determine that sequential jobs have the longest job duration. The duration decreases for jobs using 2 to 16 processors, and then increases again for jobs of higher parallelism.

Hotovy, et. al. do not do analysis of the CVs of applications as was done by Feitelson and Nitzberg. However, they have made the log files available so that we can determine the CVs ourselves. Unfortunately, neither the job name nor the path name is available in these files. Nevertheless, it is still possible to calculate the system-wide CVs and the CVs when jobs are classified by user, degree of parallelism, and queue. The wall clock times and processor times are presented in Table 1. The CV for several categories is calculated using a weighted mean. Categories are only included in the mean CV if they include at least five jobs[3]. All other categories are given a weighting in proportion to the number of jobs in the category.

From the results in Table 1, it is evident that, in general, classifying jobs by

[2] Their definition of utilization is the percentage of processors allocated to active jobs.

[3] Categories have to include at least five jobs to be included in the results so that categories that include few jobs do not make the mean CV artificially low.

Table 1. Cornell Theory Center: Wall Clock and Processor Time Means and Coefficients of Variation

	Wall Clock Time		Processor Time	
	Mean	CV	Mean	CV
System-wide	6314	5.5	84603	4.3
By user	6315	3.9	84894	2.9
By parallelism	6202	4.6	76516	2.8
15 min queue	714	18.9	1870	3.7
3 hour queue	3690	10.0	25286	3.6
6 hour queue	5781	1.7	85880	3.0
12 hour queue	17935	3.8	280292	1.7
18 hour queue	30072	1.5	377261	1.7

user, parallelism, or queue leads to lower CVs than the system-wide CV. This is an expected result. For instance, it seems likely that a user would initiate jobs with similar characteristics, or that jobs with similar parallelism would run for similar amounts of time. The lower CV for the queues, too, is expected, since a user's selection of a queue to which to submit a job provides a prediction of the job's duration. Furthermore, in this system, jobs are killed when their duration exceeds the limit associated with the queue. This enforcement of a maximum duration can lead to lower CVs.

The only entries in Table 1 where the CV for a category is greater than the system-wide CV is for the wall-clock time for jobs in the short 15 minute and 3 hour queues. This could be because these are the primary queues used for development, testing, and debugging, while the longer queues are used for production jobs.

University1 (which shall remain anonymous) does parallel computing research using a network of workstations running LSF, the Load Sharing Facility distributed by Platform Computing. The system has 85 users using IBM RS/6000 and DEC Alpha workstations. There is a parallel queue for parallel jobs, but some parallel jobs are run in the short, regular, normal and long queues. For each job, these files contain information about the wall clock execution time, the job name, the user id, the number of processors used, and the queue used[4]. Unfortunately, although the log files cover 440 days and 16,000 jobs, only 90 of the jobs are parallel and all these jobs are submitted by only two users. Thus, it is unwise to make generalizations from these log files, but it is possible to identify trends.

The NASA Lewis network of workstations is used by 25 users for simulations, analysis and code development. It, like the University1 site, also runs LSF. The system consists of 60 SUN, HP, SGI, and IBM RS/6000 workstations running primarily over an Ethernet network, but also over FDDI and ATM networks. Most of the parallel jobs are submitted to the regular queue, although there are

[4] The next release of LSF is expected to contain additional information, including processor time and memory usage.

a few PVM jobs that are submitted to a separate PVM queue. The log files for NASA Lewis contain data for 3,682 jobs over a period of 152 days. Of these jobs, 395, or 11%, are parallel.

Table 2. University1 and NASA Lewis: Wall Clock Time Means and Coefficients of Variation

	University1		NASA Lewis	
	Mean	CV	Mean	CV
System-wide	242	4.1	35115	3.9
By user	232	4.1	35049	2.5
By queue	146	3.8	35839	2.8
By parallelism	209	3.6	35282	2.9
By executable	64	0.6	42124	2.0
By exec., user, parallel.	60	0.6	46490	1.5

Table 2 shows the wall clock mean times and CVs for the parallel jobs classified in different ways. In all the cases, the CVs for the various categories are lower than the CV of the entire system. The categories used by Feitelson and Nitzberg, executable, user, and degree of parallelism, proves to be the most effective categorization. In this case, the CVs for University1 and NASA Lewis are 0.6 and 1.5, versus system-wide CVs of 4.1 and 3.9 respectively.

This workload analysis not only supports the hypothesis that historical information may be used to provide accurate predictions of job duration. It also suggests how to categorize jobs to ensure that the predictions are accurate. Feitelson and Nitzberg's results and our results for the NOW systems suggest that classifying jobs by executable, user, and degree of parallelism leads to low CVs for the categories. Since the executable name was unavailable in the log files from CTC, these files cannot verify this result, although the lower CV when classifying by user or parallelism supports it. The CTC site files suggest an alternative classification might be by queue, but the NOW files show that a classification by executable, user, and degree of parallelism is more effective. Thus, the latter classification will be used.

3 The Historical Profiler

The first step in the design of the Historical Profiler is defining the information that the Historical Profiler should provide to the scheduler. In the literature [MEB90, Dow88, PD89, GST91, Wu93, Sev94, PS96] scheduling algorithms that use the execution time of jobs have been frequently examined. Therefore, the Historical Profiler will provide a method of obtaining an estimate of the time a job will take to execute, with an indication of the uncertainty in the estimate.

3.1 Environment

The experimental platform consists of a network of workstations environment, with 16 IBM RS/6000 UNIX workstations communicating over Ethernet, Fast Ethernet, and ATM networks. The Ethernet network will be the primary network used for all experimentation.

The development of a Historical Profiler and scheduling algorithms for parallel jobs in this environment requires system support in several areas such as job management, host management, and the remote execution of parallel jobs. The commercial Load Sharing Facility (LSF) [LSF96, ZZWD93] supports many of these requirements. LSF does job, host, and queue management, supports access to all the job information required, and allows an external scheduler to control jobs to specify when and on which processors each job will be started. It also allows access to system information through the LSF Application Programming Interface (API).

Unfortunately, every call to the LSF API requires crossing address spaces. For efficiency, another layer is required, the Job and System Information Cache (JSIC). The JSIC, developed by Parsons [Par97] with help from the author, stores the data required by the profiler and the scheduler in the same address space. The information in the JSIC is periodically updated by polling LSF.

The use of these two layers has several advantages. First, they reduce the development time required for a Historical Profiler. Second, LSF is commercial software for load balancing on a distributed system, so it is fault tolerant, and it helps to ensure that the profiler and schedulers are fault tolerant. Furthermore, since future versions of LSF are likely to include in the log files information about jobs' processor times and memory usage, it will be relatively easy to add these features to our software. Finally, and most importantly, LSF is used at many production sites on many different platforms. Since our software is developed on top of LSF, our software should be easy to install and test on production sites that use LSF.

3.2 Interface

The Historical Profiler is an object with a public method for estimating the execution time of a job. Both the inputs and outputs are shown in Figure 1. The inputs to the method indicate the desired accuracy of the estimate and the job for which the profiler is making the estimate. The outputs are the estimate and an indication of the uncertainty in the estimate.

Jobs are identified by five attributes. The first three are the executable name, the user who initiated the job, and the number of processors. The final two are the wall clock time used by the job so far and the maximum memory usage of the job so far. The attained wall clock time is used in the predictions so that the longer the job has run, the longer the total execution time will be predicted to be. If the job has already run for ten minutes, the prediction for the total job duration will exclude the data for jobs that ran less than ten minutes. The inclusion of the memory size metric is based on the premise that the problem

getEstimate()
Inputs:

string executionCommand	string user	int numProcs
float attainedWallClock	float memSize	float confidenceDesired

Outputs:

float confidenceIntervalSize	float estimate

Fig. 1. The Interface to the Profiler Class

size has a positive correlation to the execution time of an executable, and the maximum memory used gives an indication of the problem size. This feature is included for future versions of LSF that provide memory usage information.

The remaining input is the percentage confidence C in the estimate that is desired. The method returns a mean estimate and a confidence interval such that the actual mean execution time for the job is C percent likely to lie within the interval.

3.3 Design

The Historical Profiler obtains all of its information from the accounting files from LSF. However, searching all the data in the log files for all historical executions of a single executable whenever a scheduler requests information would be extremely costly. To deal with this difficulty, the Historical Profiler has its own permanent repository to store data in a more appropriate format.

Figure 2 shows the structure of the Historical Profiler. At the bottom is LSF, which obtains data about the jobs from the log files. The Job and System Information Cache calls functions in the LSF API to read this data. It then converts LSF's data structures into its own data structures, and stores the information in the Historical Profiler repository.

When the scheduler at the top of Figure 2 requests information from the Historical Profiler, the profiler first asks the JSIC to update the information in the Historical Profiler Repository. Then the Historical Profiler reads the information from the Historical Profiler Repository, and transforms the data into the format requested by the scheduler.

The Historical Profiler Repository The design of the Historical Profiler Repository is intended to support the desired functionality of the profiler without requiring excessive storage space. The schedulers request information based on executable and user names. Therefore, these criteria are used to index entries in the repository. Every time a particular user runs a particular executable, the resource usage for that job is included in the appropriate repository entry. (If

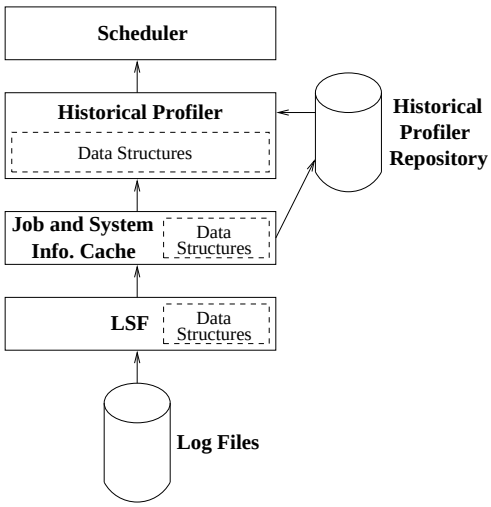

Fig. 2. High-level Design of the Historical Profiler

information about a particular executable-user pair is requested but is not in the repository, an entry that includes all the jobs ever run is used for the predictions instead.)

The next issue is determining what information is provided in each repository entry. Including complete detailed information for each job would be useful, but could lead to excessive use of storage space and either slow access times or complicated data structures. Therefore, each repository entry consists of several bins, each of which includes information for multiple jobs. Since the data for calculating mean execution times and confidences is required, the data stored in the bin includes the number of jobs included in the bin, the sum of the execution times, and the sum of the squares of the execution times [5].

It is also necessary to select an appropriate number of bins to use. In this case, a three-dimensional array of bins indexed by execution time, memory usage, and processor allocations is appropriate. The estimates of the execution time can vary based on the attained execution time, memory usage, and number of processors allocated. As a result, each repository entry must have bins containing data for multiple execution time, memory usage, and processor allocation ranges.

Calculating Execution Times The repository supplies the data required by the Historical Profiler, but the profiler is required to manipulate this information into an execution time estimate usable by a scheduler. To do this, the profiler uses well-known statistical methods for estimating means and confidence intervals

[5] Confidence intervals can be derived if data for the mean, number of entries, and standard deviation is available, and the standard deviation can be derived from the quantities recorded.

based on a number of observations.

One complication arises when predicting the execution time of a job that is using a number of processors different from that the executable has used in the past. For instance, suppose a given executable has been run on two processors, four processors, and sixteen processors, but an estimate is desired for the executable running on eight processors.

This problem is dealt with by estimating the execution time function based on the available information. For any $p > 0$, the execution time function $T(p)$ represents the execution time of the job executing on p processors. Using weighted least squares [DS81], a standard method of approximating functions using several point estimates of varying accuracy, the Historical Profiler finds a quadratic approximation of $T(p)$. It then evaluates this function for the desired number of processors to find the appropriate execution time estimate [6].

4 Evaluation of the Historical Profiler

4.1 Algorithms

In order to evaluate the performance of the Historical Profiler, scheduling algorithms are required. Eight variants of three basic algorithms are used:

1. **First Come, First Serve (FCFS)**: Jobs are serviced in strict FCFS order.
2. **First Come, First Serve Fill (FCFS-fill)**: Jobs are generally serviced in FCFS order. However, if insufficient processors are available to service the next job in the queue, but a different job in the queue that requires fewer processors can run, that job will be run. Jobs are never preempted.
3. **EASY-kill**: Lifka's EASY algorithm [Lif95]. Estimates of each job's duration are used. If any job exceeds its estimate, it is killed. Jobs are serviced in FCFS order. However, if insufficient processors are available to service the next job in the queue, but a different job in the queue requiring fewer processors can run *and is guaranteed to finish without delaying any previously submitted job*, that job will be run. Jobs are never preempted.
4. **EASY-preemptive**: EASY-preemptive services jobs in the same order as EASY-kill. The only difference is that if a job exceeds its estimate, it is preempted and put at the end of the FCFS queue, just as if the job had been resubmitted. The estimate of the service time is unchanged.
5. **Least Estimated Work First (LEWF)**: Jobs are serviced is a strict least estimated work first order without preemption.
6. **Least Estimated Work First Fill (LEWF-fill)** : Jobs are serviced in a least estimated work first order without preemption. However, if insufficient processors are available to service the next job in the queue, but a different job in the queue requiring fewer processors can run, that job will be run.

[6] A more detailed description of the use of weighted least squares in this context is given discussed elsewhere [Gib97].

7. **Least Estimated Remaining Work First (LERWF)**: LEWF with the addition of preemption. Jobs are run in strict least estimated work first order. If a job is running, and another job arrives that is expected to complete in less time, the first job will be preempted to run the second.
8. **Least Estimated Remaining Work First Fill (LERWF-fill)**: LEWF-fill with the addition of preemption. Jobs run in least estimated work first order with preemption. Preemption occurs under the exact same conditions as LERWF: if a job is running, and another job arrives that is expected to complete in less time. Unlike LERWF, if insufficient processors are available to service the next job in the queue, but a different job in the queue requiring fewer processors can run, that job will be run.

Most of these algorithms require estimates of the execution times of jobs. For comparison purposes, the algorithms are tested using both the imperfect estimates provided by the profiler and perfectly accurate information obtained from the applications. The profiler estimate that is used is the greatest value in a 95% confidence interval for the mean execution time, a very conservative estimate typically much greater than the mean.

4.2 Workload

Thirteen synthetic applications are used to evaluate the performance of the algorithms. These applications execute for a period of time that depends on a pseudo-random total work parameter, W, Amdahl's fraction sequential parameter, D [Amd67], and the number of processors on which the application is running, p:

$$t = W(D + \frac{1 - D}{p}) \tag{1}$$

For any job, p, W, and D are determined as follows. The number of processors p is selected randomly between 2 and 16 inclusive according to a Uniform distribution. The value D is a value that is constant for any application. The fraction sequential parameters, D, for the synthetic applications vary between 0.1 and 0.001. W is derived on the NASA Lewis workload as follows. The first twelve executables of the workload have mean work proportional to the product of the number of processors and the mean execution time for the twelve most frequently run parallel executables. The thirteenth job represents the aggregation of all the other parallel executables that were run in the system (approximately 35% of the jobs). Similarly, the coefficients of variation for the applications are chosen to be the same as the measured coefficients of variation for the corresponding executables in the NASA Lewis workload.

The actual work W required for a given job in Equation 1 is randomly determined based on the mean and variance associated with the corresponding executable. If the coefficient of variation is less than one, an Erlang distribution of work is assumed, if equal, an Exponential distribution, and if greater, a Hyperexponential distribution.

The test consists of 200 jobs submitted with interarrival times chosen from an exponential distribution with a mean of 150 seconds. The next executable to run is randomly determined. The probability of a given executable being the next job submitted is the proportion of the number of runs for this executable in the NASA Lewis workload.

The initial state of the Historical Profiler can affect the predictions significantly. In this case, the Historical Profiler is seeded with twenty-five random executions of each of the thirteen executables.

Table 3. Parameters Used in the Experiments

Number of Executables	13
Number of Jobs	200
Interarrival Time Distribution	Exponential
Interarrival Time Mean	150 s
Profiler estimate	greatest value in a 95% confidence interval
Number of Processors Distribution	Uniform
Minimum number of processors	2
Maximum number of processors	16
Initial Seeding of Profiler	25 random executions of each executable

In order to ensure a fair test of the algorithms, a single sequence of pseudo-random numbers is used. Thus, in a single experiment, every algorithm must handle the exact same jobs submitted at the exact same times, and the repository contains the exact same information.

5 Results

This section discusses the results of the experiments in which different schedulers are used to schedule a particular test workload. The primary criteria used to judge the performance of the algorithms is mean response times. The benefits of preemption, filling, and knowledge are examined. Table 4 classifies the algorithms based on these three criteria.

The results of the experiments are summarized in Table 5. In this table, the average response time is calculated as the average time from when a job is submitted until it finishes. The wait time is the average time from when a job is submitted until it first begins executing. The utilization is the average number of processors assigned to executing jobs during the test.

5.1 The Relative Performance of the Non-Preemptive Schedulers

To judge the relative performance of the algorithms, we compare the performance of the simplest non-preemptive versions of the algorithms using perfect service

Table 4. Classification of Algorithms

Accuracy of Knowledge	Non-Preemptive		Preemptive	
	Non-Filling	Filling	Non-Filling	Filling
None	FCFS	FCFS-fill		
Imperfect	EASY-kill-pro LEWF-pro	LEWF-fill-pro	EASY-pre-pro LERWF-pro	LERWF-fill-pro
Perfect	EASY-act[1] LEWF-act	LEWF-fill-act	LERWF-act	LERWF-fill-act

[1]Note: EASY-act is actually EASY-kill-act and EASY-pre-act. These two algorithms function in exactly the same way if the job durations are predicted accurately, since no jobs are killed or preempted.

Table 5. Performance of Scheduling Algorithms ("pro" means use of the profiler, "act" means use of actual service times)

Algorithm	Preempted (Killed)	Number Preemptions	Mean Response (s)	Mean Wait (s)	Elapsed Time (s)	Util. (%)
FCFS	0	0	6480	6251	43986	64.6
FCFS-fill	0	0	2284	2056	37609	75.0
EASY-act	0	0	2361	2138	36684	74.7
EASY-pre-pro	24	43	2429	1864	40130	69.9
LEWF-act	0	0	1031	804	41760	67.3
LEWF-pro	0	0	2965	2738	39116	71.6
LEWF-fill-act	0	0	926	697	39271	72.0
LEWF-fill-pro	0	0	1259	1031	38234	74.0
LERWF-act	31	124	908	469	44456	63.7
LERWF-pro	25	121	2404	1565	48017	61.5
LERWF-fill-act	34	135	855	267	42684	67.3
LERWF-fill-pro	38	163	1678	389	45441	65.8
EASY-kill-pro	(24)	0	∞[2]	602	29781	58.9

[2]Note: This time is not given because 24 long-running jobs were killed. If this fact is ignored, the mean response time is 763.

time knowledge, FCFS, EASY-act and LEWF-act. Figure 3 shows the relative performance of these three algorithms. As would be expected, FCFS has the worst performance, EASY-act the middle, and LEWF-act the best. This is the expected result, since FCFS makes no attempts to change the ordering of jobs to decrease the mean response times. EASY-act does change the order somewhat, by the addition of filling, but not as much as LEWF.

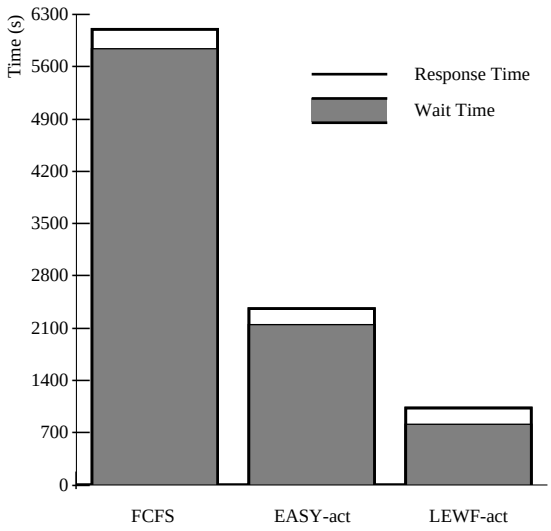

Fig. 3. The Non-Preemptive Schedulers' Mean Response Times

5.2 The Value of Filling

Filling is the first addition to the basic algorithms which will be examined. Figure 4 compares the mean response times for filling and non-filling algorithms. From these results, it is clear that in general, filling improves the mean response times. It is intuitive that FCFS would be improved by the addition of filling, so the results are not surprising. The average response time is reduced by almost two-thirds, while the total time required decreases by 17%. Filling means that jobs that could not be started if strict ordering were used can be started earlier. Thus, the wait times decrease, as is evident from the test.

Since the variants of LEWF order the queues in an attempt to minimize the mean response time, it was not clear that changing this ordering by using filling would lead to improved response times. Filling could result in a number of short jobs being delayed for a long time by a long job that was filled. However, the results indicate that this does not happen. In all cases, filling reduces the mean response times. The improvements in the wait times for jobs requiring few processors outweighs the effects of short jobs being delayed by long-running filled jobs. This is particularly true for the variants of LEWF that use the profiler.

5.3 The Value of Preemption

There are five pairs of algorithms that differ only in that one is preemptive and the other is not, EASY-kill-pro and EASY-pre-pro, LEWF-act and LERWF-act,

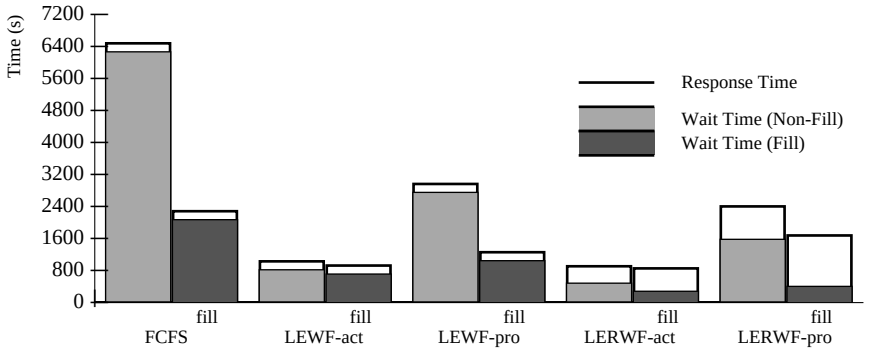

Fig. 4. The Impact of Filling on Mean Response Times

LEWF-pro and LERWF-pro, LEWF-fill-act and LERWF-fill-act, and LEWF-fill-pro and LERWF-fill-pro[7]. The relative performances of the final four pairs of algorithms are shown in Figure 5.

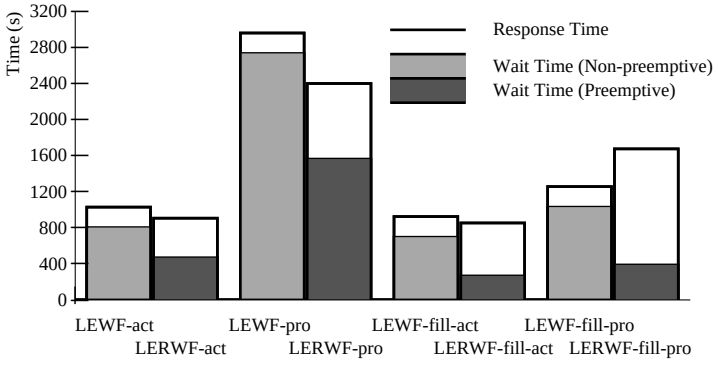

Fig. 5. The Impact of Preemption on Mean Response Times

Comparing the performance of the non-preemptive EASY scheduler using the profiler to the preemptive version of the same scheduler is meaningless. EASY-kill-pro disposes of all the jobs quickly, but only because it kills 24 of the longest

[7] Another comparison might be made between the preemptive version and non-preemptive versions of EASY that use perfect data. However, such a comparison is uninteresting because no jobs are preempted, and so the algorithms lead to the exact same schedule and have the exact same performance (that of EASY-act).

running jobs when they exceed the estimates. It is unlikely that users would prefer a scheduler that kills jobs in such as haphazard manner whenever the scheduler's estimates of the execution time are inaccurate.

For three of the four variants of the LEWF algorithm, preemption improves the mean response times. The improvements are greater than the additional overhead of preemption, and stem from the fact that long jobs that are running can be preempted in order to run shorter jobs. In the non-preemptive version, if a long job starts running, subsequently arriving shorter jobs must wait until that long job finishes if there are insufficient available processors. Because preempted jobs require more wall-clock time from when they are first started to when they finish, the difference in the average response time and the average wait time is higher for LERWF-act than for LEWF-act, but this increase is still less than the improvement in average wait time.

LEWF-fill-pro and LERWF-fill-pro are the only pair of algorithms in which the performance of the preemptive algorithm is worse than that of the non-preemptive algorithm. The poor response time is due to two factors. The first is the overhead of preemption. This by itself is not enough to make the preemptive algorithm worse, since the other variants of LEWF are able to overcome this overhead. The second factor is the limitations of the current heuristic for assigning processors in LERWF-fill. Whenever a job that begins execution has a choice of processors on which to run, the scheduler attempts to avoid assigning the processors of the next preempted job in the pending queue. This policy is an attempt to ensure that the processors required by the next preempted job will be available when it is ready to run. Unfortunately, this tends to lead to a scenario where all the jobs that require few processors are assigned the same processors, so that several jobs of this type cannot run concurrently. Jobs are started relatively quickly, but after they are suspended, it takes a long time before they are resumed. This hypothesis is supported by the large difference in the average response time and average wait time for LERWF-fill pro, relative to the other algorithms. Improving the heuristics for the LERWF-fill algorithms could reduce the mean response times.

5.4 The Value of Knowledge

Three different levels of knowledge will be compared: no knowledge, imperfect knowledge, and perfect knowledge. These levels will be represented by FCFS, the algorithms using the profiler, and the algorithms using the actual execution times, respectively. Figure 6 shows the performance of the algorithms. Each set of three algorithms consists of one algorithm using no knowledge of the application, one using imperfect knowledge, and one using perfect knowledge.

Comparing the use of no knowledge to the use of any knowledge, even imperfect knowledge, shows clearly that knowledge is highly beneficial. Both FCFS and FCFS-fill are much worse than the LEWF and LERWF algorithms that use the same type of filling. In the case of FCFS, the poorest comparable algorithm is LEWF-pro, which still has a mean response time less than half that of FCFS, while the other algorithms only improve on this performance. The results are

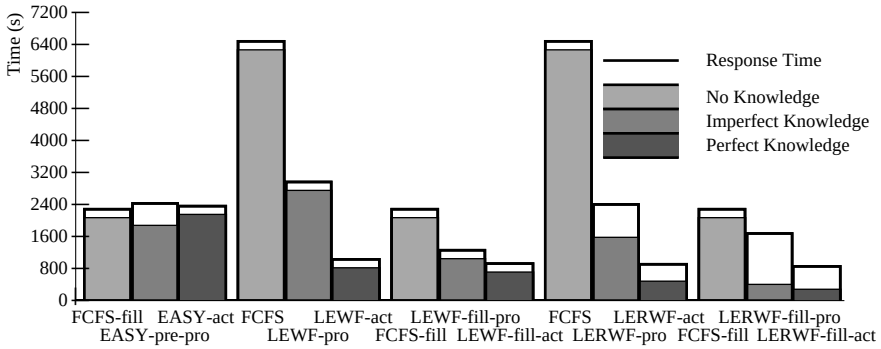

Fig. 6. The Improvements in Mean Response Time Due to Knowledge

similar for the filling variants. The only exceptions are the variants of EASY, where application knowledge does not improve performance. However, this is because EASY does not use application knowledge to ensure low mean response times, but rather only to make the schedule predictable. Both the perfect and imperfect knowledge variants of EASY have roughly the same mean response time.

Now we will examine the impact of the accuracy of knowledge by comparing the improvements in mean response times over FCFS and FCFS-fill that are attainable using imperfect knowledge to the improvements attainable using perfect knowledge. For the LEWF algorithm, LEWF-pro has mean response times 46% as long as FCFS, while LEWF-act has mean response times of only 16% as long. This difference is noticeable in Figure 6. For the filling version, the mean response time of LEWF-fill-act is approximately 41% of the mean response time of FCFS-fill, while the mean response time of LEWF-fill-pro is 55% of that of FCFS-fill. Thus, in the first case, the improvements due to knowledge are substantial, but not close to the results with perfect information. In the second, the difference is far less significant.

It is worthwhile examining the cause of this difference in performance between perfect and imperfect information. In general, the jobs are run in the "correct" executable order so that the executables with the least work are run before the ones with more. The problem arises in distinguishing between different jobs involving the same executable. In this case, the jobs requiring more processors are run first, since it is expected that the more processors available, the shorter the

job[8] (a result that may not necessarily be true in a real workload[9]). As a result, when the profiler is used for scheduling, several jobs with short execution times but requiring few processors are delayed until the end of the test, after longer jobs with more processors have finished. This difference has less of an impact on the filling versions of the algorithm, since short jobs requiring few processors are likely to be filled regardless of how high the execution time estimates are.

Finally, there are the preemptive algorithms. As is evident in Figure 6, LERWF-pro has an average response time equal to 37% of the average response time for FCFS, while LERWF-act has an average response time of 14% of that of FCFS. LERWF-fill-pro has a mean response time equal to 73% of that of FCFS-fill, while LERWF-fill-act has a mean response time of 37% of that of FCFS-fill. In the former case, the scheduler using imperfect information attains most the benefits possible due to the use of knowledge. The latter has a much larger difference. As mentioned previously, the poor performance of the LERWF-fill-pro algorithm stem from the heuristic for assigning processors.

Thus, in every case, the use of knowledge is beneficial. In addition, for all but the LERWF-fill-pro algorithm, the algorithms using the profiler achieve 75% of the possible improvements in mean response times that are attainable using perfect information.

6 Conclusions

Through the analysis of the workload on parallel processing sites, it was evident that historical job information could be used to improve the performance of a scheduler. The execution times of jobs categorized by executable name, user, and number of processors were relatively predictable.

To take advantage of these results, a Historical Profiler that uses historical job information to calculate execution time estimates was designed and implemented. Experiments to evaluate this profiler with variants of the FCFS, EASY, and LEWF scheduling algorithms led to the following main results:

1. Out of the three basic, non-preemptive algorithms, FCFS is the worst, EASY-act is better, and LEWF-act is the best.
2. Filling reduces the mean response times attainable for all disciplines.
3. Preemption reduces the mean response times attainable for most disciplines.

[8] This is caused by the jobs that were used to seed the profiler's repository. For any job, the amount of work was selected according to the workload distribution, and the number of processors was selected from a uniform distribution. The run time of that job was then calculated to be approximately proportional to the ratio of the work to the number of processors, leading to a negative correlation between the run time and the number of processors.

[9] If instead, the reverse were true and jobs with more processors ran longer than jobs with fewer processors, the profiler's predictions of job length would still be relatively accurate. However, with such a workload, the importance of preemption might increase and the importance of filling might decrease.

4. The heuristic that the preemptive disciplines use for assigning processors to jobs has a large impact on the mean response times attainable using those disciplines.
5. Schedulers that use application knowledge can attain lower mean response times than those that do not.
6. In many cases, schedulers that use the imperfect knowledge from the profiler can attain most of the possible knowledge-related improvements to mean response times.

There are several possible areas of future work. First, the Historical Profiler could be installed at a production site. This would conclusively show the benefits possible from from a Historical Profiler. Second, this work only discussed non-adaptive space sharing scheduling algorithms that did not permit the migration of jobs. This is a small subset of all scheduling algorithms; future research could evaluate the performance of the profiler with other types of scheduling disciplines. Third, many of the performance problems of the LERWF algorithms were attributed to the methods of assigning processors. An examination of the performance improvements attainable using different heuristics for assigning processors would be interesting. Finally, the addition of more data to the repository, or the use of the existing data in the repository in different ways, could lead to interesting findings.

Future work is required to address these issues, but the most important issue has been resolved. The results indicate that it is feasible to use information about previously run parallel jobs to predict the characteristics of future jobs. Furthermore, these predictions can improve the performance of schedulers substantially.

7 Acknowledgements

Thanks to Kim Johnson for the NASA Lewis log files, Steve Hotovy for the Cornell Theory Center log files, and the University1 site for its log files. Thanks to Platform Computing for providing LSF. Also, thanks to Ken Sevcik, Eric Parsons, and Songnian Zhou for their help in writing this paper.

References

[Amd67] G. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of the 1967 AFIPS Conference*, volume 30, AFIPS Press, pages 483–485, 1967.

[AS97] S.V. Anastasiadis and K.C. Sevcik. Parallel application scheduling on networks of workstations. *To appear in: Journal of Parallel and Distributed Computing*, June 1997.

[BG96] T.B. Brecht and K. Guha. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27(8):519–539, October 1996.

[CS85] M. Calzarossa and G. Serazzi. A characterization of the variation in time of workload arrival patterns. *IEEE Transactions on Computers*, C-34(2):156–162, February 1985.

[DAC96] A.C. Dusseau, R.H. Arpaci, and D.E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 1996.

[Dow88] L. W. Dowdy. On the partitioning of multiprocessor systems. Technical Report Technical Report 88-06, Vanderbilt University, March 1988.

[DS81] N.R. Draper and H. Smith. *Applied Regression Analysis, 2nd ed.* John Wiley and Sons, Toronto, 1981.

[FN95] D.G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/ 860. In *Proceedings of IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 215–227, April 1995.

[Gib97] R.B. Gibbons. A historical profiler for use by parallel schedulers. M. Sc. thesis, University of Toronto, Toronto, Ontario, Canada, 1997.

[GST91] D. Ghosal, G. Serazzi, and S. K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[Hot96] S. Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In *Proceedings of IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 15–22, April 1996.

[HSO96] S. Hotovy, D. Scheider, and T. O'Donnell. Analysis of the early workload on the Cornell Theory Center IBM SP2. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 272–273, May 1996.

[Kum88] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computing*, 37(9):1088–1098, September 1988.

[Lif95] D.A. Lifka. The ANL/IBM SP scheduling system. In *Proceedings of IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 187–191, April 1995.

[LSF96] *LSF Users's Guide*. Platform Computing Corporation, 5001 Yonge St, Suite 1401, North York, ONT, Canada M2N 6P6, 1996.

[MEB90] S. Majumdar, D.L. Eager, and R.B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 104–113, May 1990.

[MEB91] S. Majumdar, D.L. Eager, and R.B. Bunt. Characterization of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation*, 13(2):109–130, February 1991.

[NVZ96a] T.D. Nguyen, R. Vaswani, and J. Zahorjan. Parallel application characterization for multiprocessor scheduling policy design. In *Proceedings of IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 105–118, April 1996.

[NVZ96b] T.D. Nguyen, R. Vaswani, and J. Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In *Proceedings of IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 93–104, April 1996.

[Par97] E.W. Parsons. *Using Resource Requirements in Multiprogrammed Multiprocessor Scheduling*. Ph. D. thesis, University of Toronto, Toronto, Ontario, Canada, 1997.

[PBK91] J. Pasquale, B. Bittel, and D. Kraiman. A static and dynamic workload characterization study of the San Diego Supercomputer Center Cray X-MP. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 218–219, 1991.

[PD89] K.H. Park and L.W. Dowdy. Dynamic partitioning of multiprocessor systems. *International Journal of Parallel Programming*, 18(2):91–120, February 1989.

[PS95] E.W. Parsons and K.C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In *Proceedings of IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 76–88, April 1995.

[PS96] E.W. Parsons and K.C. Sevcik. Benefits of speedup knowledge in memory-constrained multiprocessor scheduling. *Performance Evaluation*, 27(8):253–272, October 1996.

[Sev89] K.C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[Sev94] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.

[Wu93] C.S. Wu. Processor scheduling in multiprogrammed shared memory numa multiprocessors. M. Sc. thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, October 1993.

[ZZWD93] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogenous distributed computer systems. *Software: Practice And Experience*, 23(12):1305–1336, December 1993.