A Historical Application Profiler for Use by Parallel Schedulers

by

Richard Gibbons

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

A Historical Application Profiler for Use by Parallel Schedulers

Richard Gibbons

Master of Science

Graduate Department of Computer Science

University of Toronto

1997

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 thesis 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 the best way of categorizing parallel jobs for storage in a job database and the job information that 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 estimates. 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 estimates obtained from the Historical Profiler lead to improved performance.

Acknowledgements

The log files containing historical job information for several systems that are discussed in this thesis were obtained from production NOW and multiprocessor sites. University1, a site that has chosen to remain anonymous, supplied the log files for their network of workstations. Kim Johnson was kind enough to provide me with the NASA Lewis NOW log files. Steve Hotovy allowed me to access the Cornell Theory Center IBM SP2 log files. I appreciate these people's willingness to provide me with the data upon which much of the research in this thesis is based.

Platform Computing deserves credit for providing LSF for use in the POW project, and for discussing methods of implementing scheduling algorithms on top of LSF.

I would also like to thank Eric Parsons, who created much of the Job and System Information Cache. His efforts made the implementation of the algorithms discussed in this thesis much easier. Eric also provided valuable feedback on many of the ideas presented in this thesis. These comments were appreciated.

Ken Sevcik, my advisor, provided many helpful comments and criticism throughout the creation of the thesis. His input greatly enhanced the quality of this work. I also appreciate his understanding of my priorities when writing this thesis, and his extreme efforts to support those priorities.

Thanks also go to Songnian Zhou, my second reader, for his interest in this work and his constructive comments.

Finally, I would like to thank NSERC for providing the funding that enabled me to do this research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Studies have indicated that existing networks of workstations (NOWs) are not being used at full capacity. Douglis and Ousterhout's examination of the Sprite operating system's process migration facilities [DO91] finds that even during the daytime on weekdays, over two-thirds of workstations are idle. Similarly, Mutka and Livny's study of the usage of 13 to 20 workstations over 9 months [ML91, Mut92] discovers that even at the busiest times, 55% of capacity is available. More recent analysis by Arpaci, et al. of a cluster of 53 workstations [ADV$^+$95] finds that the total number of idle workstations remained roughly constant, such that more than 60% of workstations were available at any given moment.

Such findings encourage viewing a network of workstations as a single system with excess capacity, rather than as multiple distinct machines intended for individual use. This idea immediately leads one to consider methods of improving the performance and functionality of such a system. One such method would be to use the excess capacity of NOWs to run parallel jobs. This, in effect, is treating the NOW as a parallel machine.

Previously, most parallel programming was done on parallel processors. However NOWs are becoming increasingly popular as a substitute. Anderson, Culler and Patterson attribute this trend to several factors [ACP95]. NOWs offer a significantly better price-performance ratio than a massively parallel processor (MPP). They do not have the one to two year lag that exists between the release of a new processor and its use in an MPP. Furthermore, since workstations generally already include an operating system, NOWs do not suffer to the same extent from the complications and high cost of creating an entirely new operating system. Additionally, with the large excess capacity of NOWs, existing NOWs can be used as a parallel machine with little additional cost.

The study by Arpaci, et al. [ADV+95] confirms the viability of running parallel and sequential workloads simultaneously on NOWs. Using simulations based on traces and benchmarks, they demonstrate that it is possible to provide acceptable service to interactive users while still allowing parallel jobs to have good performance. The technique in this study is to allow the coscheduling of parallel jobs using processors that have been idle for at least three minutes. To ensure that interactive users are not inconvenienced, parallel threads are migrated as soon as interactive use of a workstation resumes, and a daily limit is set on how many times parallel jobs can run on a given machine. This technique relies on the facts that, in general, a constant number of machines are idle at any time, and 95% of idle time is spent in intervals longer than 10 minutes, even though half of the idle periods are less than three minutes long.

These findings support the argument that doing parallel computation on NOWs is both feasible and desirable. However, for parallel applications to be commonly run on networks of workstations, operating system support for these applications in this environment is required. The support needed ranges from infrastructure to help parallel applications run on several machines and communicate with each other, to software to make the development of programs in this environment easier. This thesis will focus on one particular area that requires support, parallel job scheduling.

Much research has focused on parallel job scheduling in multiprocessors, and the area is still open to inquiry. It seems natural that many of the scheduling results that apply to multiprocessor systems may also apply to scheduling parallel jobs in a NOW environment. However, on closer examination, this may not be so self-evident, because the job mixes found in the two environments may be very different. Because machines in the NOW environment may be less reliable than the processors in a multiprocessor and communication may be slower, users may choose to run different types of jobs in the two environments. This is even more likely to be true if the users have access to both a multiprocessor and a NOW acting as parallel machine. Despite these misgivings, it is still worthwhile considering results obtained in multiprocessor environments.

Previous research in these multiprocessor environments has indicated that knowledge of application characteristics can improve the performance of parallel system schedulers [MEB90, PD89, GST91, MEB91, Wu93, AS97, PS96]. However, very few practical methods of determining application characteristics for use by a scheduler have been suggested or implemented [DAC96, NVZ96b, NVZ96a].

1.1 Objectives

The overall goal of this thesis is to determine a reasonable way for scheduling algorithms for parallel jobs executing on a NOW to achieve better performance by using knowledge of the historical resource usage of individual applications. This will be done by analysis of parallel job patterns on production networks of workstations, and by implementing and analyzing scheduling algorithms that can take advantage of this workload characterization. The resulting system will be tested and evaluated on a sixteen node network of workstations.

Chapter 2 will discuss previous results that impact this work. It presents both earlier workload characterization studies, and analysis of the characteristics of parallel applications. It then describes some parallel scheduling algorithms that use application knowledge. This chapter is intended to provide a broad overview of relevant research preceding this thesis.

The parallel workloads of one multiprocessor machine and two networks of workstations are examined in chapter 3. The goal of this analysis is to identify commonalities among the workloads in the three production sites. This examination contributes to the determination of features of the workload that are both predictable and useful in improving scheduling effectiveness. The primary focus will be methods of classifying jobs so that the coefficients of variation for wall clock execution time and processor time of each class will be relatively small.

Chapter 4 will use the results of Chapter 3 when specifying the implementation of a "database", called the historical profiler, that contains information about the historical resource usage of applications. The purpose of this chapter is to present the issues associated with creating a Historical Profiler in a NOW environment. To this end, it will present several features that a profiler should have to be useful to a scheduler. Methods of storing the data that consider the tradeoff between storage space and amount of detail are proposed. Finally, this chapter presents algorithms for transforming raw execution time data about jobs into approximate execution time functions more easily used by the scheduler. The execution time function for an application indicates its expected execution time as a function of the number of processors it is allocated.

Chapter 5 contains a discussion of the issues associated with implementing schedulers that use the Historical Profiler. This chapter serves a dual purpose. First, it introduces the algorithms to be used to test the performance of the Historical Profiler. Second, it indicates the difficulties encountered in designing several parallel schedulers in a NOW environment. Several preemptive and non-preemptive variants of three scheduling algorithms are examined, First Come First Serve (FCFS), Least Estimated Work First (LEWF), and Lifka's EASY [Lif95].

These scheduling algorithms are used to test the performance of the Historical Profiler. Chapter 6 evaluates the performance of these algorithms in scheduling parallel applications to run on a NOW, using information from the Historical Profiler. It compares this performance to the performance of the algorithms when perfect knowledge of execution times of the applications is available. The workload used to evaluate the algorithms is presented, followed by analysis of the results of the experiments. The goal is to determine how well these scheduling algorithms perform when using the profiler, relative to the ideal.

Chapter 2

Context and Previous Work

This chapter presents results preceding the work in this thesis. Section 2.1 discusses load balancing on networks of workstations and the Load Sharing Facility, LSF [LSF96]. Section 2.2 presents previous workload characterization studies and techniques. Section 2.3 briefly summarizes methods for characterizing parallel applications. Section 2.4 discusses the EASY scheduler, followed by Section 2.5 which presents other scheduling algorithms that could potentially use workload information, if it were available. This section motivates Section 2.6, which describes methods that scheduling algorithms currently use to obtain workload information.

2.1 Load Balancing on Networks of Workstations

Load balancing on NOWs has been investigated for over a decade. Several NOW modeling and simulation studies [ELZ86, MTS90] support the hypothesis that relatively simple load balancing algorithms can result in large improvements in the average execution times of jobs. A simulation using job traces from production VAX machines [Zho88] confirms these findings. Zhou finds that under a moderate load of approximately 60% processor utilization, mean response times improve by 30-60%, and that even with light loads, the performance of every host improves. A more recent UNIX study by Harchol-Balter and Downey [HBD96] analyses the lifetime distributions of processes and proposes a policy for preemptive migration. Using trace-driven simulations, they show that their policy reduces mean delays by 35-50% compared to non-preemptive migration where jobs can only be migrated before they have started.

Many implementations and algorithms for load balancing have been proposed. Early efforts include Leblang and Chase's implementation of a parallel make on a network of workstations [LC87] and Theimer's techniques for finding hosts for remote execution [TL89]. A more recent study by Kunz using an artificial workload [Kun91] examines the use of different host workload descriptors to determine where to schedule processes. He discovers that using the node with the minimal number of tasks in the run queue leads to the most efficient dispatching of jobs.

Implementations of distributed operating systems, such as V [Che88] and Sprite [DO91], provide support for load sharing. However, subsequent research, on systems such as the Condor [LLM88] and Utopia [ZZWD93], focuses on implementing load sharing in a layer on top of the workstation operating system. In particular, LSF version 2.2, a commercial version of Utopia, is the load sharing system used in this thesis.

LSF, Load Sharing Facility [LSF96], provides transparent interactive and batch load balancing on a heterogeneous NOW. LSF is implemented on top of UNIX and a shared file system. It dynamically provides resource information, including number of processors, relative speed, and physical memory available on each host, and load information such as processor load, available memory/swap space, and I/O and paging activity. LSF automatically selects the hosts on which to run a job, based on the resource requirements of the job, and the current load conditions. Restrictions and resource limits may be specified for queues, for users, or for hosts. Accounting systems record the resource usage of all jobs on the system in log files.

Minimal support for parallel jobs is provided by LSF through a generic interface. Parallel applications that use packages such as PVM or MPI can be started by LSF's batch interface using shell scripts provided by LSF or written by the application programmers. An Application Programming Interface (API) provides programmers with access to most of LSF's features, including the remote execution of threads. This thesis will focus on enhancing LSF's batch execution features using the LSF API and LSF accounting mechanisms.

2.2 Workload Characterization

The primary goal of resource allocation algorithms, including scheduling algorithms, is to provide the workload on the system with effective access to system resources. The meaning of "effective" depends on the resource being allocated, the workload, and the system. As a result, when creating a scheduler, it is useful to examine the workload that will be scheduled.

Many workload characterization studies of parallel and sequential applications executing on different platforms exist. However, there are none that deal specifically with NOWs running parallel production workloads. This could be because running parallel jobs on NOWs is a new approach, so little data is available. However, despite this lack of data, there are other workload characterization studies that are of interest.

An early study by Agrawala, Mohr and Byrant [AMB76] on a two processor Univac workload presents several techniques used frequently in workload characterization studies. They discuss different job models and the use of clustering theory in the classification of jobs.

A different approach is used in a uniprocessor modeling study by Devarakonda and Iyer [DI89]. This study uses cluster analysis and state-transition modeling of trace files from a UNIX system to create a job model. This model is able to predict execution time with a high correlation to the actual execution time. The errors are relatively small, with 80% of jobs falling within half a standard deviation of the predicted value. The log files also have the interesting property that 21% of executables account for 92% of jobs. Any program with the same path and name is considered the same executable; if two programs are in different directories but have the same name, they will still be considered different executables[1]. These results indicate that, in sequential systems at least, it should be possible for a scheduler to use historical data to accurately predict applications' execution time.

Since the presumption of this thesis is that the NOW is being used as a parallel

[1]For the work in this thesis, a slightly different method is used. Executables are classified only by name, not by path. As a result, any two programs with the same name but possibly in different directories are still considered the same executable.

machine, multiprocessor workload characterizations are more relevant to this research than uniprocessor studies. Several studies of multiprocessor machines exist. Pasquale, Bittel, and Kraiman present a workload characterization of a production Cray X-MP based on two months of data from 1989 [PBK91]. Their clustering analysis shows that 88% of the jobs accounted for less than 2% of the processor time. However, 2% of jobs used 86% of the processor time, 77% of the memory space-time product, and 21% of the I/O time. They analyse the arrival rate of jobs, and find a common pattern of workload increasing from 8:00am to noon, remaining roughly constant until 5:00pm, and then decreasing until 9:00pm.

An earlier study of job arrival patterns in a multiprocessor was done by Calzarossa and Serazzi [CS85]. They analyse 14 one-day periods using polynomial-fitting techniques to derive functions for the arrival rate based on the time of day and cluster analysis to find groups of applications that have similar arrival behaviour. They verify their findings with a second set of data from a different month.

The most complete characterization of a multiprocessor system, an iPSC/860 hypercube, is presented by Feitelson and Nitzberg [FN95]. This study categorizes jobs by degree of parallelism and analyses each category in terms of the number of jobs, average processor time, cumulative processor time, and type of jobs. Furthermore, they analyse job submission rate, average job length, and job interarrival times in terms of time of day. They conclude by profiling both user activity in terms of number of jobs submitted and applications used, and applications in terms of the number of times each application is run and the variance in execution length. Some of their interesting results include:

1. A small number of large jobs consume most of the resources.
2. Despite the fact that system-wide job run times and interarrival times have a high coefficient of variation, the job run times of multiple executions of the same application on the same number of nodes tend to have a coefficient of variation less than one.

The first finding is also supported by Pasquale, et al.'s study [PBK91]. These results indicate that while the overall distribution of job execution times in the system is hyperexponential, it should still be possible to predict the execution times of individual applications.

The most recent studies of multiprocessors are Hotovy's examinations of a production IBM SP2 system [Hot96, HSO96]. These are the only workload characterization studies that note that the system utilization increases over a nine month period. He observes an increase in both the weekly backlog and median wait times. He examines the number of jobs and the processor time by the number of processors requested, and observes that most jobs requested a number of nodes that is a power of two. Hotovy's results differ from those of Feitelson, et al. and Pasquale, et al. in that the average job duration does not clearly increase for an increased number of processors. Instead, the duration is the highest for sequential jobs, then decreases for up to sixteen processors. The duration then increases for up to 32 processors but levels off for jobs requiring more processors. Preliminary analysis of more recent log files of the same site by Parsons [Par97] appears to confirm this increasing trend for small numbers of processors, followed by a leveling off for jobs using more processors. This system is examined in more detail in Section 3.1.

Each of these studies has focused on specific systems. However, it is also worthwhile exploring the methodology of workload characterization. Calzarossa and Serazzi [CS93] do this when they examine the common features of workload characterization studies. They present a five step methodology for the analysis of workloads and the construction of artificial workloads.

1. Formulation of the characterization level and basic components.
2. Choice of instrumentation.
3. Collection of data.
4. Analysis of data using partitions, parameter distributions, sampling, static characterizations, dynamic characterizations, and the construction of models.
5. Determining the validity of the model on other data.

They also suggest parameters on which to focus for various studies. Unfortunately, they do not address the workload characterizations of NOWs; they primarily emphasize network issues. For parallel applications, they suggest a focus on many characteristics of such applications, including those presented in the next section.

2.3 Parallel Application Characterization

Workload characterization is important for resource allocation, but it is also worthwhile to investigate the system at a finer granularity, the individual applications. The analysis of characteristics of parallel applications is valuable since systems software can be written to optimize the performance of such applications. However, there are many different ways that applications can be analysed. This section will present some of those methods.

One technique is to measure characteristics of applications using simulations while varying the size of data sets or various application parameters. There are many studies of this type. Woo, et al.'s analysis of the SPLASH-2 programs [WOT+95], 12 benchmark programs for shared address multiprocessors, takes this approach. Using execution-driven simulations, it analyses the programs in terms of speedup, load balancing, working sets, communication to computation ratio, and spatial locality. Cypher, et al. [CHKM93] do a similar study on eight parallel scientific applications running on two different multiprocessor architectures. They analyse the applications in terms of memory, I/O, and processing requirements, communication to computation ratio, message traffic, scaling of problem size and scaling of the number of processors. A further study of this type is done by Nguyen, Vaswani and Zahorjan [NVZ96a]. They examine several applications in the Perfect [CKPK90] and Splash-2 [WOT+95] benchmark suites running on a KSR-2 in terms of speedup and sources of slowdown. The major problem with such studies is that it is difficult to tell whether the applications studied really are representative of applications in production systems.

A similar technique that determines how system characteristics affect performance is to execute representative applications on different configurations of a system. Lantz, Nowicki and Theimer conducted an early study of this type examining a client/server system [LNT85]. Using graphics and text benchmarks, they found that the bandwidth of the network had only a small role in determining the performance of the system. More important factors were the speed of the machines involved and the transport protocol.

All the studies presented thus far focus on actual applications. However, much theoretical work has addressed more general models of parallel applications. Sevcik discusses the many different levels and types of theoretical characterizations [Sev89]. Low level characterizations such as data dependency graphs and task precedence graphs exist. The former deals with operations on data that are required to be sequential, while the latter decomposes an application into dependencies between purely sequential tasks. However, in practice, such low level characterizations can be both difficult to obtain for large applications and difficult to use.

Higher level characterizations have been proposed, and some of the definitions of terms used in higher level parallel application characterization appear in Table 2.1. Amdahl [Amd67] established an influential approach to characterizing parallel applications by identifying a limit on the speedup of an application with p processors, $S(p)$, based on the fraction f of the application that is intrinsically sequential:

$$S(p) \leq \frac{1}{f + (1-f)/p} \tag{2.1}$$

The execution time function, $T(p)$, is a popular way of characterizing an application in terms of the number of processors it uses. $T(p)$ is the execution time with p processors. This characterization has the benefit that it can generally be measured easily by running the application several times with different numbers of processors.

Various methods of defining the parallelism of an application in terms of a single number have been proposed, including the minimum and maximum parallelism. Eager, Zahorjan, and Lazowska investigate the use of average parallelism, A, and the tradeoff between speedup and efficiency for varying numbers of processors [EZL89]. The speedup and efficiency functions are defined in terms of the execution time function:

$$S(p) = \frac{T(1)}{T(p)} \tag{2.2}$$

$$E(p) = \frac{S(p)}{p} \tag{2.3}$$

Eager, et al. note that speedup and efficiency are inversely related to each other, so both measurements cannot simultaneously be low for any allocation of processors. They

Table 2.1: Application Characteristics

Characteristic	Definition
Execution Time Function $T(p)$	A function that for any p is equal to the duration of the application executing on p processors.
Speedup Function $S(p)$	The ratio of the execution time for the application running on one processor to the execution time of the application on p processors.
Efficiency Function $E(p)$	The ratio of the speedup for the application using p processors to p.
Average Parallelism	The average number of busy processors during the execution of an application if an unlimited number are available.
Maximum Parallelism	The maximum number of busy processors during the execution of an application if an unlimited number are available.
Minimum Parallelism	The minimum number of busy processors during the execution of an application if an unlimited number are available.
Processor Working Set (pws)	The number of processors that maximizes the product of the speedup and efficiency functions.
$\omega(p)$	The ratio of the time required for an application to execute on p processors to the time required on an infinite number of processors.

also note that the average parallelism can approximate the knee of the graph of speedup versus execution time.

Ghosal, Serazzi and Tripathi [GST91] expand on this work by actually finding the knee of the speedup versus execution time graph. The processor working set (pws) is defined as the value of p that minimizes the ratio of the execution time to the efficiency, $\frac{T(p)}{E(p)}$, or, correspondingly, maximizes the product of the speedup and efficiency. Conceptually, the pws attempts to indicate with a single parameter the threshold at which the gain in the speedup of an application is worth the marginal cost of using an additional processor.

Methods of characterizing the parallelism of applications are also a popular research topic. Majumdar, Eager and Bunt [MEB91] propose a function that provides an indication of the variability in parallelism:

$$\omega(A) = \frac{A}{S(A)} = \frac{T(A)}{T(\infty)} \tag{2.4}$$

This function is the execution time attainable by a job with A processors when processor sharing, using time-slicing, is employed.

A more detailed way of characterizing the parallelism of an application is in terms of the parallelism profile [Sev89]. The parallelism profile of an application is a graph of the maximum number of processors that an application can use at various times during its execution.

One problem with many of these characterizations is that they rely on knowledge of the execution time function. It is possible to measure the execution time function, but to get a smooth curve from incomplete data, a model of the function is required. Several functions have been proposed as models, and at least two representations are used several times in the literature.

Dowdy [Dow88] proposes an execution signature of the following form to characterize an application by execution rate $\mu(p)$:

$$\mu(p) = \frac{p}{C_1 p + C_2} \tag{2.5}$$

The variables C_1 and C_2 are constants specific to the application. From this expression, the execution time function may be derived:

$$T(p) = \frac{1}{\mu(p)} = \frac{C_1 p + C_2}{p} = C_1 + \frac{C_2}{p} \tag{2.6}$$

The main alternative to Dowdy's model is Sevcik's model [Sev94]. Sevcik's more general proposed model for the execution time function is based on four parameters. The work of an application is represented as W. A function $\phi(p)$ represents the degree to which work is not evenly spread among processors. The increase in work per processor due to parallel processing is α, while the communication delays and other delays that increase with p are represented by the product βp. Sevcik's formulation is:

$$T(p) = \phi(p)\frac{W}{p} + \alpha + \beta p \tag{2.7}$$

Dowdy's formulation (equation 2.6) is an instance of Sevcik's formula, where $W = C_2$, $\phi(p) = 1$, $\alpha = C_1$ and $\beta = 0$. Wu [Wu93] characterizes applications in terms of both Dowdy's and Sevcik's forms, and finds applications where Dowdy's form has large errors. The problem arises because Dowdy's execution signature is a non-decreasing function in p. However, for any parallel application, there is a point where the congestion and communication overhead become so great that assigning an additional processor to the application slows it down rather than speeding it up. Sevcik's function does not have the same disadvantage, since it has the βp term to deal with this situation.

2.4 The EASY Scheduler

When investigating scheduling, it is worthwhile examining popular current algorithms in order to judge both the performance of the algorithms and the requirements of users. Users are starting to demand the availability of certain features in scheduling software [RSLS95]. One response to this demand has been Lifka's creation of the Extensible Argonne Scheduling sYstem (EASY). EASY [Lif95] was designed with the help of users of the IBM SP system on which it was originally intended to run. Since that time, it has become increasingly popular and has been adopted by other sites [SCZH96].

EASY was designed according to users' goals of fairness, simplicity, predictability, exclusive access to nodes, and support for different job types. To satisfy the goal of simplicity, there are twelve UNIX-like commands with intuitive functions. Yet, despite

Table 2.2: Scheduling Terms

Characteristic	Definition
static scheduling	The number of processors assigned to a job cannot change after that job has begun execution.
dynamic scheduling	The number of processors assigned to a job can change after a job has begun execution.
preempt	Temporarily stop the execution of a job or a thread.
resume	Continue the execution of a preempted job or thread.
migrate	Move a job or thread that was running on one or more processors to continue execution on a different processor or set of processors.
adaptive	The number of processors assigned to a job when it initially begins execution is determined by the scheduler.
non-adaptive	The number of processors that must be assigned to a job is specified by the user, not the scheduler.
run to completion	Once started, a job executes to completion with no preemption or migration.
coscheduling	All the threads of a job are run simultaneously on different processors.
space-sharing	Jobs have exclusive access to the processors assigned to them.

this simplicity, EASY is able to meet all the other goals. To ensure that the users' exclusive access to nodes does not compromise the fairness of access, an accounting system is included. Each user has a limited quota of processor-minutes. After that quota has been exceeded, the user is not allowed to submit any more jobs.

Fairness and predictability are implemented using a static scheduling scheme based on FCFS with backfilling. When a job is submitted, the user must specify the duration of the job. Any job that fails to complete within the specified duration is killed. The FCFS with backfilling algorithm means that jobs are run in strict first come first serve order, except when a job can be run on available processors without delaying the start of any other job submitted before it. This algorithm ensures that, at any time, users can see the current schedule of every job in the system, and know that a submitted job will start execution no later than the time reflected in the current schedule.

The main drawback of this scheme is that it is far from optimal for either response time or efficiency. In an effort to remedy this situation somewhat, a new implementation of EASY is planned that allows executing jobs access to available nodes [SCZH96]. If a running job could use more processors than it currently has, it can request access to idle nodes as long as the request will not delay other jobs.

2.5 Selected Scheduling Results

Workload characterization results from Section 2.2 suggest that parallel application performance is predictable, while the success of EASY, described in Section 2.4, indicates that users are willing to specify limits on the run-time of applications. These results naturally lead to the question of whether knowledge of application characteristics can improve the performance of scheduling algorithms. The majority of results indicate that the answer is "yes".

Majumdar, Eager and Bunt's simulation [MEB90] of an artificial workload running on a shared memory multiprocessor compares the mean response time using FCFS and RR (round-robin) policies to SNPF and SCDF schedulers. SNPF allocates processes to the job with the smallest number of processors not yet allocated. Thus, jobs that demand few processors will, in effect, be scheduled before jobs that demand more processors.

SCDF, smallest cumulative demand first, allocates free processors to the job with the least cumulative demand, where the cumulative demand is the product of the number of processors the job uses and the length of the job. The study discovers that the greater the variability in demand, the more beneficial is the additional workload knowledge that SNPF and SCDF possess, and the better is the performance of algorithms that use preemption.

An alternative technique used in several other studies is to use instead knowledge of the processor working set for scheduling. Ghosal, et al. [GST91] use four benchmark applications running on a sixteen transputer machine to compare various static scheduling algorithms that use the pws. They suggest allocating to jobs a number of processors equal to their pws (the PWS rule) as a viable scheduling strategy, although they note that allocating the pws does not maximize average speedup. They do not compare in depth their algorithms using the processor working set to any other algorithms.

A synthetic workload-based simulation by Majumdar, Eager and Bunt [MEB91], however, does compare the performance of algorithms using the pws to algorithms using other application characteristics. This study focuses on the identification and use of appropriate application characteristics for scheduling. It finds that the PWS rule yields near optimal performance for many workloads. Majumdar, et al. also suggest other static scheduling algorithms that use knowledge of both A, the average parallelism of the application, and $\omega(A)$, and the execution time attained by a job using A processors when processor sharing is used among the active threads. Furthermore, they propose a form of dynamic scheduling, *program behaviour-based scheduling*, where the number of processors assigned to an application changes whenever the speedup or average parallelism of the application changes. Not surprisingly, this type of scheduling leads to lower average response times than static scheduling, where the number of processors cannot change after the application has started executing.

A study by Chiang, Mansharamani, and Vernon [CMV94] a few years later seems to contradict these results. It finds that algorithms that use average parallelism and the processor working set perform worse than alternative static policies that do not use such information. Using simulation and an artificial workload, Chiang, et al. discover that both adaptive static partitioning with a maximum limit on the number of processors allocated, ASP-max, and shortest demand first with a maximum limit on number of processors allocated, SDF-max, perform better than policies using A and pws. With ASP-max, a new job is allocated either all the idle processors in the system, or its maximum parallelism. When a job completes, its processors are allocated in a round robin fashion to jobs waiting in an FCFS queue. SDF-max schedules jobs in order of increasing demand with an upper limit on the number of processors assigned to a each job. Chiang, et al. also determine that EQ, a dynamic policy that assigns an equal number of processors to all the jobs in the system (up to maximum parallelism), outperforms all the other policies presented. They explain the contradictions to previous studies by citing differences in the policies and workloads examined. This result is verified by Parsons and Sevcik [PS95].

Sevcik is responsible for two studies examining the use of knowledge of application behaviour when scheduling. In the first [Sev89], he compares several static scheduling algorithms that use knowledge of average parallelism, system load, and minimum and maximum parallelism. His simulations lead to the conclusion that at low loads, knowledge of average parallelism is sufficient. However, at higher loads, schedulers that use additional information can improve mean response times.

In the second study, Sevcik [Sev94] examines several special cases of allocating P processors to N applications for which execution time functions (equation 2.7) are known. He derives optimal allocations to minimize average response time for one application on P processors, N applications on two processors, N identical applications on P processors, and two applications on P processors.

Several other researchers propose scheduling algorithms based on the execution signatures and speedup functions of applications. Park and Dowdy [PD89] note that execution signatures may be obtained experimentally, and may be used to calculate corresponding speedup functions. They observe that such a function may be used in scheduling to maximize the throughput of a system. They support this assertion by presenting an iterative method that can be used by a dynamic scheduler to determine the optimal allocation of processors to jobs.

Wu [Wu93] uses Sevcik's execution time function (equation 2.7) to analyse several parallel applications in a non-uniform memory access (NUMA) environment. He then shows that a static scheduling algorithm using information about the execution time function outperforms a dynamic scheduling scheme where the applications notify the system of how many processors they would like. He explains this result by the overheads associated with the dynamic policy.

Simulations by Anastasiadis and Sevcik [AS97] of three workloads continue this line of research. This study demonstrates that, at high loads, static algorithms that use workload knowledge can outperform an EQ policy with no overhead. At lower loads the EQ algorithm remains better because it is a dynamic scheduling strategy. They also show that SDF-max does not generally perform better than SDF as is claimed in the study by Chiang, Mansharamani, and Vernon [CMV94]. Rather, the maximum limit on the number of processors assigned to an application simply optimizes the performance of the algorithm for a particular workload and arrival rate.

Parsons and Sevcik [PS96] provide additional results about the performance of scheduling algorithms that use the execution time function. This research focuses on the benefits of application knowledge of applications with various memory requirements. This modeling study shows that, if there is no correlation between the execution time and memory size of jobs, a dynamic algorithm with knowledge of the execution time function only moderately outperforms an EQ algorithm. However, if memory requirement and execution time are correlated, the algorithm with knowledge outperforms the EQ algorithm by a large margin. Parsons and Sevcik propose several algorithms to take advantage of this result and find, using simulations, that at all loads, the algorithms using application knowledge are able to outperform EQ. They note that the jobs in the study have vastly different execution time functions, and that a workload of jobs with less diverse execution time functions would not benefit from the same performance improvements.

A similar study of dynamic schedulers based on Dowdy's execution signature (equation 2.5) is described by Brecht and Guha [BG96]. They find that algorithms that use application characteristics such as jobs' remaining work, efficiency, and processor working set are able to outperform a simple EQ algorithm. In particular, an algorithm that considers both the remaining work of jobs and their efficiency results in the most significant improvements in performance. Like Parsons and Sevcik [PS96], Brecht and Guha point out that the possible improvement over EQ is affected by the variability in execution times, the efficiency of jobs, and the system load.

2.6 Methods for Ascertaining Application Characteristics

It is clear from findings in Section 2.5 that knowledge of application characteristics can be used to improve the performance of a scheduler. This section will discuss methods that have been used to gather information about application characteristics.

One method of measuring application parallelism is a tool created by Kumar [Kum88]. His tool inserts statements into application code to determine *ideal parallelism*, which is the ultimate level of parallelism attainable by the code if it were to run with no overhead on an infinite number of processors. However, the tool as proposed has a major disadvantage. It requires special versions of an application to be created and run to determine the application characteristics. Such a procedure greatly inconveniences users of the system. A similar disadvantage exists with an idea Sevcik [Sev94] presents, of having the user provide estimates of execution characteristics.

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 communication and synchronization events that are 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 that multiple processes in a job have a high probability of being scheduled simultaneously. It does this without passing around any global information such as messages to synchronize clocks. Simulations for artificial workloads show that this method yields response times no more than 25% longer than for coscheduling algorithms (defined in Table 2.2), and for coarse-grained algorithms can be 25% better. This method has the advantage of not requiring extra effort from either the application programmer or the user to achieve acceptable scheduling.

Nguyen, Vaswani and Zahorjan [NVZ96b, NVZ96a] use a combination of code instrumentation and hardware monitors to determine run time characteristics of iterative applications. These studies assume that the behaviour of future iterations of loops will be similar to the behavior of past iterations of the same loop. By determining the execution times of the loop over several iterations with varying processor allocations, the scheduler is able to estimate the speedup characteristics of the application. Simulations of a variety of workloads indicate that a variant of the EQ scheduler that uses runtime information, ST-EQUI, is able to outperform an EQ scheduler that does not. Furthermore, the performance of ST-EQUI is almost comparable to that of a scheduler that uses perfect information about the application. The main disadvantage of this method is that, thus far, it 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. Despite results discussed in Section 2.2 that seem to indicate that this approach holds some potential, it has not been implemented by anyone up to now, but will be addressed in this thesis.

Chapter 3

Workload Characterization

When designing parallel scheduling algorithms, it is useful to analyze the workloads on both multiprocessing systems and networks of workstations running parallel jobs. Such analysis provides an idea of the important characteristics of jobs running on such machines. For this thesis, in particular, such analysis is critical, since this work focuses on the creation of a Historical Profiler to be used by parallel job scheduling algorithms. Before such a profiler can be created, it is necessary to decide what information could be useful to a scheduler. An analysis of workloads lets us do this.

Ideally, this workload analysis would focus on the parallel jobs running on production networks of workstations. Unfortunately, there are several difficulties with this strategy. The first is that traces of production sites are generally not available. Either the site does not support detailed traces, or the traces contain confidential information. A further difficulty is that the use networks of workstations as multiprocessors is only now starting to become feasible with the development of high-speed networks and networking protocols. As a result, there are few sites that execute parallel jobs on networks of workstations in a production mode. Additionally, in some of cases where parallel jobs are run on networks of workstations, the jobs are actually run on a single multiprocessor workstation on the network.

To reduce these difficulties, this thesis will examine not only the parallel workloads on two production networks of workstations, but also the multiprocessor workload described by Hotovy, et al. [Hot96, HSO96]. The characterization of parallel applications in a multiprocessor environment may be relevant to networks of workstations used as parallel machines. In both cases, the goal is to run parallel jobs in an environment that has multiple processors and communication paths between the processors. However, it may be that, due to the slower communication in NOWs and the distributed memory of the systems, users run different jobs or different versions of the same jobs on NOWs than they run on multiprocessors. Despite these potential problems, we will consider multiprocessor workload characterizations relevant to a NOW environment.

Three sites will be examined. The multiprocessor site, the Cornell Theory Center IBM SP2, is examined in Section 3.1. The two network of workstations sites are the anonymous University1 research site, described in Section 3.2, and the NASA Lewis site, addressed in Section 3.3. In each case, the focus will be on the parallel jobs that are running on the systems, and the sequential jobs will be ignored. A brief description of each system will be presented, and the log files will be analyzed. This analysis will be limited to aspects of the workload that are potentially useful to a parallel job scheduler, and that can easily be stored in a database. Section 3.4 summarizes the relevant conclusions that may be drawn from the workload analysis.

3.1 Cornell Theory Center IBM SP2

The Cornell Theory Center IBM SP2 consists of 512 processing nodes, each of which has from 128 MB to 2048 MB of memory, and 2 GB of disk capacity for swap space. There are two types of nodes, *wide* and *thin* nodes. Wide nodes have four times the cache and four times the bandwidth between the cache and memory relative to thin nodes. During the 169 days of log files, the number of nodes available for batch jobs varied from 200 to 430. There were a total of 54,042 batch jobs, of which 25,831, or 48%, were parallel.

The system has five queues based on the maximum execution time of jobs: 15-minute, 3-hour, 6-hour, 12-hour, and 18-hour. The 15-minute queue has the highest priority, while the rest of the queues have equal priority. If a job exceeds the maximum duration allowed in a given queue, the job is killed. LoadLeveler, the scheduler used in this system, supports a space sharing run to completion policy. To ensure that jobs eventually receive service, it increases the priority of the jobs that have waited the longest to receive service. Users are allowed to have a maximum of two jobs simultaneously executing, although more are permitted to be in the queues.

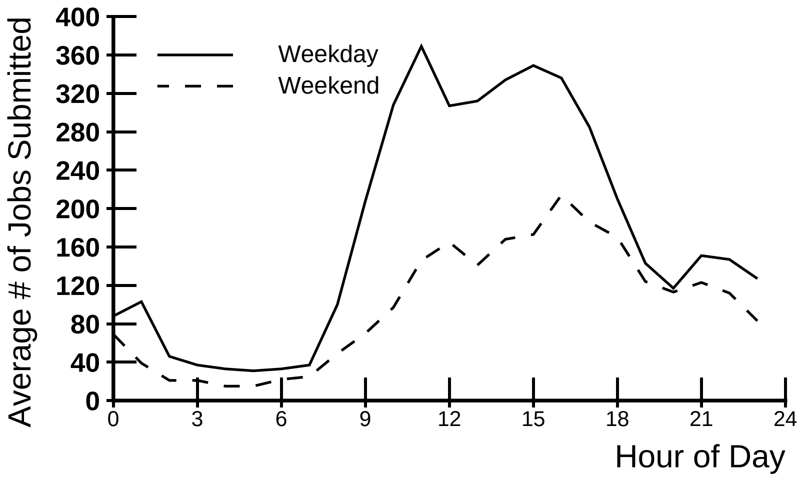

Figure 3.1: Cornell Theory Center: Parallel Job Submission Rate vs. Time of Day

Hotovy, et al. [Hot96, HSO96] analyzed several features of this workload, which are presented briefly here. They noted that over the six-month time period, the use of the machine, measured by user-node time, increased, but the number of jobs declined. This difference was caused by a large (250%) increase in the duration of jobs and an increase in the average number of processors used by each job. Throughout the entire time period, the system had an average utilization of only 60%, where the utilization is the percentage of processors allocated to active jobs. During most of the period, the number of queued jobs increased, as did the average wait time. In terms of parallelism, over half of the jobs were serial, but these jobs accounted for only 8.6% of the user-node time. 55% of parallel jobs requested a number of nodes that was a power of two. Hotovy did not find a clear relationship between the number of processors and the job duration. He found that sequential jobs had the longest job duration. The duration decreased for jobs using 2 to 16 processors, and then increased again for jobs of higher parallelism.

Several aspects of this workload were not analyzed by Hotovy, but are useful for scheduling, in particular, the number of jobs submitted by time of day. This information is relevant, because it has been shown [Sev94, PS96] that static schedulers should reduce the number of processors allocated per job when the load increases. Thus, knowledge of when the load is likely to increase in the future is useful. Figure 3.1 shows this, for

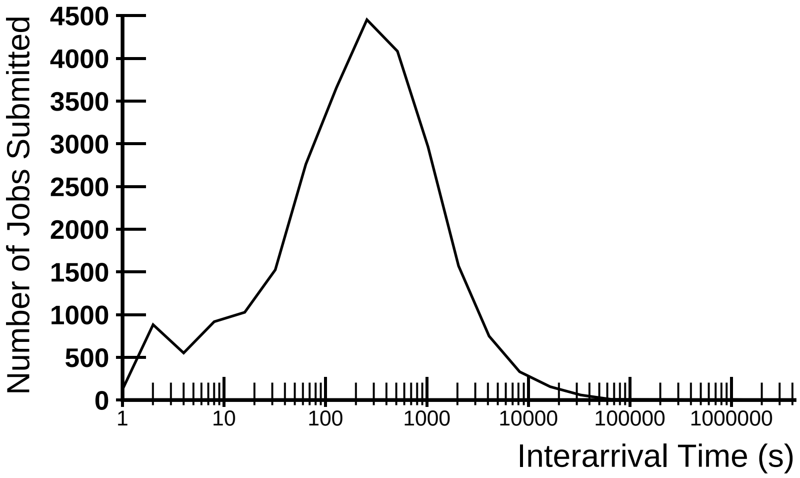

Figure 3.2: Cornell Theory Center: Number of Parallel Jobs vs. Interarrival Time (adapted from Hotovy, et al.'s paper [HSO96])

both weekdays and weekends. This graph is extremely similar to the ones presented by Feitelson and Nitzberg [FN95] and by Pasquale, et al. [PBK91], with a peak at noon, and a low at 8:00 am. A graph of the number of jobs dispatched by time of day looks very similar, but is smoother, as would be expected. The interarrival time graph for the Cornell Theory Center site, in Figure 3.2, based on work by Hotovy, et al. [HSO96], has a shape extremely similar to that of a corresponding graph presented by Feitelson and Nitzberg [FN95].

Hotovy performs no analysis of the variance of job run times for the system, individual jobs, or queues. However, such data is useful, since it provides an indication of the possible performance improvements attainable by the scheduling algorithm. It also shows which statistics are useful to store in a database for use by a scheduler. This analysis of the variance of run times is presented in Table 3.1. The relatively high system-wide coefficient of variation[1] for both the wall-clock time and the processor time is noteworthy. The higher the value, the more differences there are among the jobs and the more potential benefit there is from a scheduler that distinguishes among jobs. If the scheduler can classify jobs into categories that have lower coefficients of variation, it should be able to to schedule

[1]The coefficient of variation, or C.V. is the ratio of the standard deviation to the mean of the distribution.

Table 3.1: Cornell Theory Center: Wall clock time and Processor time Averages and Coefficients of Variation

Type	Wall Avg.	Wall C.V.	Processor Avg.	Processor C.V.
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
$5 <$ Num. Jobs < 20	6626	1.5	198099	1.9
Num Jobs ≥ 20	6198	4.7	75329	2.8

the entire system more effectively.

One possible classification is to divide the jobs based on the user initiating the jobs, with the premise that each user tends to initiate jobs with similar characteristics. Table 3.1 shows this category. Each number for the classifications by user is calculated to only include users who ran more than five jobs. Each user is given a weighting in the averages and coefficients of variation proportional to the number of jobs that user ran. The results indicate that overall there is less variance in both wall clock and processor time when jobs are classified this way. Alternatively, jobs can be classified based on their parallelism. Using a weighted average as with user, this classification, too, results in a lower coefficients of variation. It might be expected that a reasonable classification would be by queue, since different queues, by definition, are supposed to contain different type of jobs. This intuition is validated by Table 3.1, where most of the queues have lower coefficients of variation than the system, especially the longer queues. The two exceptions are the wall clock time for the 15 minute and 3 hour queues. This result could be due to users using these queues for development, testing, and debugging, while the longer queues were used exclusively for production jobs.

Perhaps the most natural way of classifying jobs is by the executable that is run, since it seems likely that multiple executions of the same executable should perform similarly. Unfortunately, the Cornell Theory Center log files do not contain any means to directly identify the executable that was used, so this hypothesis cannot easily be tested. However, close analysis of the number of jobs by number of processors requested shows some interesting clustering. As Hotovy notes [Hot96], requesting a power of two number of processors seems popular. Similarly, requesting multiples of 10 and 25 seems popular. In contrast, many of the processor allocations that are not powers of two, or multiples of 10 or 25, have not been requested at all. This situation makes it particularly noticeable when an allocation is popular, but does not obviously fall into one of the three popular categories. It seems reasonable that, in many of these cases, a single executable that requires a specific number of processors is being run. Isolating these cases can give a hint about the variance of executables.

Table 3.1 shows the results of dividing the jobs into two sets. All the jobs with the same degree of parallelism are placed in the same category. For each category, if the number of job executions in that category is greater than five and less than 20, it is put in one set. If the number of job executions is greater than or equal to 20, it is put in the other set. The weighted averages and weighted average coefficients of variation of both sets are in Table 3.1. As is expected, the C.V. of the first set is relatively small, supporting the hypothesis that individual executables have low coefficients of variation. Further evidence that supports this hypothesis is that the users who ran the most jobs (presumably using the SP2 for many different tasks with different executables) tended to have a higher C.V. than users who ran fewer jobs (presumably using system only occasionally, with a single executable.) Thus, although the results are by no means conclusive, there is some evidence that individual executables have a lower coefficient of variation than the overall system.

3.2 University1 Network of Workstations

University1 (which shall remain anonymous) does parallel computing research using a network of workstations running LSF. The system has 85 users using IBM RS/6000

Table 3.2: Wall Clock Average and Coefficient of Variation for the University1 and NASA Lewis

Type	Univ1 Avg.	Univ1 C.V.	NASA Avg.	NASA C.V.
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 executable, user, parallel.	60	0.6	46490	1.5

and DEC Alpha workstations. There are short, normal, and long queues for each type of machine, as well as a single queue for all parallel jobs. The log files for this site cover 440 days and 16,000 jobs. The default LSF scheduling algorithms of round-robin, FCFS, and fairshare are used to schedule the system. Fairshare schedules jobs based on queue priority, user id, how much processor time the user has used recently, and the amount of time the job has been in the queue. Each user is allocated by the system administrator a number of shares of the system, and the scheduler gives the user processor time proportional to the number of shares. The scheduler uses an FCFS policy. However, if a user has used a small proportion of their share of processing time relative to the other users, their jobs are given high priority and skip over the other jobs in the queues.

Unfortunately, there are several difficulties with the analysis of data from this site. Much of the resource information acquired by LSF and recorded in the log files is inaccurate for parallel jobs. The only data that are accurate are wall clock times, job names, user ids, numbers of jobs, numbers of processors used, and queues used. In addition, only 90 of the 16,000 jobs are parallel, less than 1%, and these jobs are submitted by only two users to two queues. Thus, it is impossible to make generalizations from these log files, although it is possible to identify some trends.

Among the parallel jobs, the maximum degree of parallelism is seven, while the average is 4.6. As indicated by Table 3.2, University1 has a high system-wide coefficient of variation for wall-clock execution time, but the weighted average C.V. of the sets of jobs partitioned by parallelism is lower, similar to the results from the Cornell Theory Center. The weighted average C.V. by queue is also slightly lower. The results also indicate that classifying jobs by the name of the executable leads to a very small coefficient of variation. This is due to the fact that the very long running executables were only run a few times, but pushed up the average execution time and the C.V. of the entire system. A classification by executable, user, and parallelism is also effective, with a C.V. of only 0.6. This classification seems reasonable since different users may use the same executables in different ways, and the executables are likely to perform differently based on the number of processors.

A graph of the number of submissions by time of day has roughly the same shape as that of the Cornell Theory Center SP2, but is based on too few jobs to support any strong conclusions.

3.3 NASA Lewis Network of Workstations

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. The queues are configured for short, regular, long, and night jobs. Additional queues exist for parallel jobs that use PVM, although the majority of parallel jobs are submitted to the regular queue. The default LSF scheduling algorithms described in Section 3.2 are used. The log files for NASA Lewis contain data for 3,682 jobs over a period of 152 days.

NASA Lewis data has a higher proportion of parallel jobs than University1, with 395 parallel jobs, about 11% of all jobs. Ten users submitted all the parallel jobs run on the system. The maximum degree of parallelism is 23, while the average is 6.1. Thus, despite the high ratio of users to workstations, the parallelism is relatively low. This could be due to slow communications and the overhead of distributed memory; jobs actually slow down when they are run on too many processors. Nevertheless, analysis of the parallel jobs on the NASA Lewis system is more significant than the analysis of University1's parallel workload due to the higher proportion of parallel jobs. Unfortunately, the limitations of

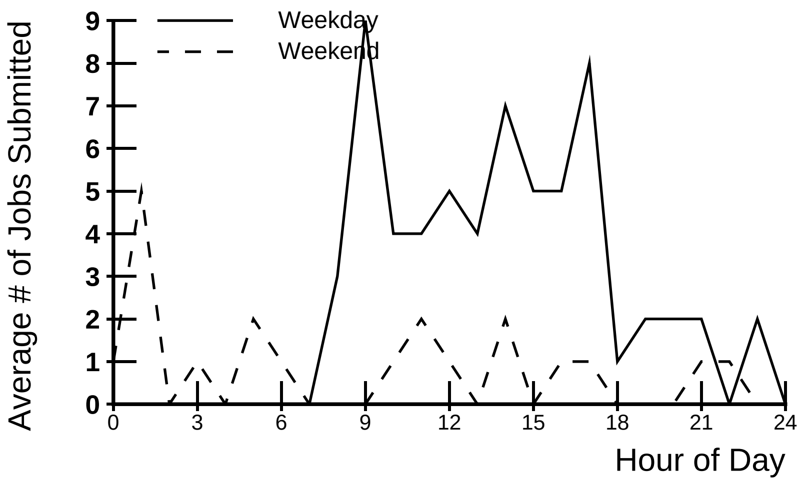

Figure 3.3: NASA Lewis: Parallel Job Submission Rate vs. Time of Day

LSF's log-file records for parallel jobs, mentioned in the previous section for University1, constrain the analysis of the workload on this site, too.

Figure 3.3, which shows the parallel job submission rate versus the time of day, has a similar shape to the Cornell Theory Center's corresponding Figure 3.1. The associated interarrival time graph, in Figure 3.4, is extremely jagged due to the low number of jobs, which makes it difficult to compare to the corresponding multiprocessor graph in Figure 3.2. However, the jobs appear to have a broader distribution of arrival times; there are several jobs which have very short interarrival times and several that have very long interarrival times. This is expected, considering the small number of jobs distributed over a large interval.

Table 3.2 contains the average wall clock times and coefficients of variation for the NASA Lewis site. Similar to both the Cornell Theory Center site and the University1 site, this site has a relatively high system-wide C.V. of 3.9. Like both of the other sites, classifying the jobs into various categories results in a lower weighted average C.V. for the categories. Classifying jobs by user, queue, and parallelism resulted in a small decrease in the C.V., while classifying by executable was more effective. A classification by executable, user, and number of processors proved most effective, with a weighted average C.V. of 1.5. This result indicates that the wall clock execution time of an executable run

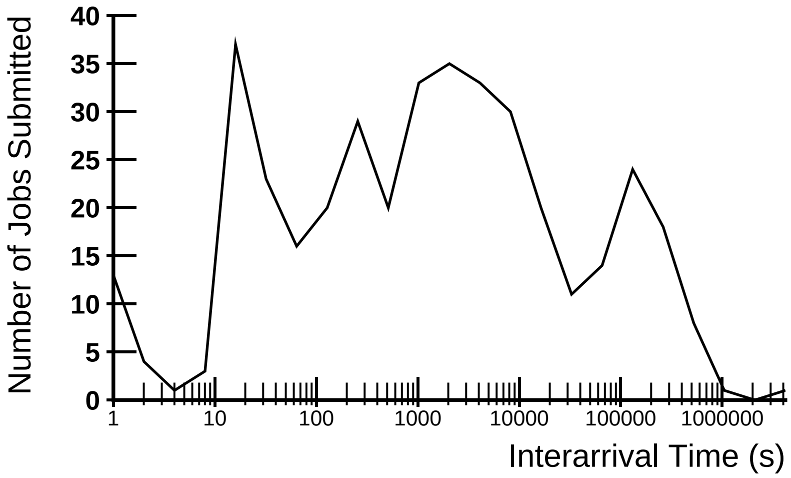

Figure 3.4: NASA Lewis: Number of Parallel Jobs vs. Interarrival Time

by a given user on a given number of processors tends to have low variability.

3.4 Conclusions

Both the results of the previous sections and the results of Feitelson and Nitzberg [FN95] and Pasquale, et al. [PBK91] indicate that, across many systems, the submission rate at a given time of day is predictable. All the graphs of the submission rate versus time of day, including Figure 3.1 and Figure 3.3, have a similar shape. Such a shape is expected, since it mirrors the work activity of a typical person during a typical day. Naturally, the usage of the system is expected to increase at 9:00 am when people generally start work. The graphs of job interarrival times were less consistent, but still reasonably similar.

Because of this extreme similarity, it would be useful putting such data in a database, but this would not help a lot. If a scheduler were to only consider the "average" workload, to plan for such a workload on an extraordinary day, such as a holiday, it could allocate resources in an inefficient manner. Thus, a more complicated historical model would be necessary. However, a simpler and more effective strategy might be one that takes into account the current load and extrapolates the future load using a static model of the changes in submission rate at different times of the day. Thus, although the results indicate that the systems have consistent submission rates based on time of day, this does not necessarily imply the best strategy for a scheduler would be to use complicated models derived from the historical data.

On the other hand, analysis of the log files does indicate that it is useful classifying jobs into categories, since categories can be found that have a lower average variability in wall clock execution time than the overall system. In particular, although categorizing jobs by queue, user, and parallelism are effective, it appears that the most effective categories are based on the executable being run, or a combination of the executable, user, and degree of parallelism. This result is intuitive, since the characteristics of executables should not change to a large degree during different executions, particularly if the jobs are started by the same user with the same number of processors. One reasonable case where this might *not* be true is where different jobs based on a single executable treat different problems sizes (e.g., different matrix sizes in a matrix factorization algorithm). In this case, it may be possible to classify a job not only by the executable, user, and number of processors, but also by the job's memory usage. Unfortunately, due to limitations in the data available, this hypothesis cannot be tested by examination of the log files we analyse.

The most important result of the analysis of these log files is that the use of historical information should be useful to a scheduler to predict the future behavior of jobs. According to these workloads, the most useful classification of jobs is by executable, user and degree of parallelism.

intro

Chapter 4

The Implementation of the Historical Profiler

Results of Chapter 3 indicate that classifying parallel jobs allows reasonable predictions of the wall-clock execution times of jobs. In particular, categorizing jobs according to the executable name, degree of parallelism, and user name leads to smaller coefficients of variation for each of the categories than the coefficient of variation for the entire system. These results naturally lead to the idea of predicting jobs' resource requirements based on historical data.

Such predictions would be extremely useful to scheduling algorithms. As discussed in Section 2.5, many results indicate that knowledge of application characteristics is beneficial when scheduling parallel jobs. In particular, Sevcik [Sev94] has shown methods of calculating the optimal schedule based on knowledge of the applications' execution time functions. As a result, a method that provides some indication of applications' characteristics is valuable.

Although researchers have used a variety of techniques to determine the application characteristics (see Section 2.6), in accordance with the results of Chapter 3, the approach discussed here will be using a "database" containing historical information about jobs, a *Historical Profiler*. A first step in designing such a component is defining the important features that should be supported. Analysis of the ways job information is used by the scheduling algorithms described in Sections 2.5 and 2.6 shows that three main features would be useful.

1. A method of obtaining an estimate of the time a job will take to execute, with some

indication of the possible error in the estimate.

2. A method of approximating the execution time function for a given job, including error ranges. This feature can be used by a scheduling algorithm to find a good processor allocation to jobs in order to maximize efficiency or minimize mean response time.

3. Hypothesis testing that allows schedulers to ask questions such as, "Based on historical data, can we be 95% confident that the mean execution time of job A at least n minutes longer than the mean execution time of job B?".

This chapter will discuss the implementation of a Historical Profiler that supports these three features in the NOW environment.

4.1 The 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. This system is used exclusively for parallel applications and systems research; none of the workstations is intended for general, daily interactive use.

The development of a Historical Profiler and scheduling algorithms for parallel jobs in this environment requires system support in several areas. First, there must be job management facilities accessible by the scheduler. The scheduler must be able to learn what jobs are in the system and their current status. Furthermore, it must be able to modify the status of jobs to take actions such as starting a given job on a particular set of processors. Similarly, there must be host management facilities accessible to the scheduler so that the scheduler can know which hosts are available. To obtain information for the Historical Profiler, accounting information about all batch jobs in the system must be available. Other features that are useful, but not strictly requiring system support, are queue management facilities and a user interface for submitting and modifying jobs.

The commercial Load Sharing Facility [LSF96, ZZWD93] from Platform Computing supports many of these requirements. LSF allows access to system information through the LSF Application Programming Interface (API). LSF supports access to all the job information required and allows an external scheduler to modify jobs to specify when and on which processors each job will be started. It also provides both host and queue management facilities that enable an external application to determine the status of each host and queue. Furthermore, using the account and event log files that LSF maintains, historical information about jobs and system events is available. LSF also provides a convenient graphical user interface for submitting jobs and monitoring the system.

One difficulty with the LSF API is that accessing information requires crossing address spaces. In addition, it does not provide a convenient way to store state information to be used by scheduling algorithms (e.g., the amount of time that the job has been suspended). These difficulties are handled by another layer, the Job and System Information Cache (JSIC). The JSIC, developed by Eric Parsons [Par97] with help from the author, stores job, queue, and host information in the scheduler's address space. To update this data, it periodically polls LSF using the LSF API. The JSIC also abstracts the LSF API interface into a form more easily used by schedulers and the Historical Profiler. For instance, it provides a function call for moving a job between queues, and another for specifying the hosts on which a job will run.

Using these two layers has several advantages and disadvantages. A major advantage is that they greatly simplify the development of scheduling algorithms and the application profiler. They provide the infrastructure required by all the scheduling algorithms, without requiring extensive design, programming, and testing. In addition, since LSF is commercial software, many of the difficulties with fault tolerance associated with a distributed environment are solved by LSF. For instance, if the machine running the scheduler should crash, LSF can automatically start the scheduler on another machine, transparently to the users of the system.

Perhaps the primary advantage is that LSF is used at production sites. As a result, it may be possible in the future to incorporate the algorithms developed into these sites for both improved performance and further research. Furthermore, LSF can be used on many different platforms. Thus, software developed on top of LSF in a portable language such as C++ is likely to be portable too. As a result, the Historical Profiler and the scheduling algorithms developed in this thesis should be easily portable to different platforms.

However, there are several disadvantages to using the LSF and JSIC layers. First, source code for LSF is unavailable since the software is commercial, so LSF cannot be modified or patched. Second, this choice tends to restrict any software developed to use only features supported by LSF. For instance, since LSF does not record in the account files the processor time for parallel jobs[1], this information is not easily accessible to the historical profiler. Third, LSF was not originally intended for this type of use, but rather has its own algorithms for determining where and when to run jobs. As a result, any scheduling algorithm built on top of LSF is required to configure the system in such a way that LSF will "decide" to schedule only those jobs that the scheduler wants to run. Despite these disadvantages, the use of the LSF and JSIC layers is justifiable, since a great deal of effort would be required to build infrastructure that is already provided by LSF.

Limitations in the current version of LSF affect the profiler directly in several ways. Information about resource usage for parallel jobs is not reliable. As a result, the most natural measure of the "work" done by an executable, the processor time, cannot be used in this environment. Similarly, determining the problem size by the amount of memory used is also not feasible. For this version of LSF, as a substitute for using processor time to measure the work of a job, the wall clock execution time will be used. The system, including the interface, will be designed so that in future releases when better information is available, it can be easily used in estimates.

4.2 The Interface

The Historical Profiler is a C++ object-oriented design consisting of two object classes, the Profiler and the ExecutionTime classes. A Profiler object stores and retrieves job historical data. ExecutionTime objects, which store information about a specific job, are returned by the Profiler object in response to queries about a job. An ExecutionTime object is an approximate execution time function for a job, with associated error ranges. It ensures that a scheduler trying to determine processor allocations does not incur the

[1]This problem is expected to be remedied in a future release of LSF.

overhead of calling the Profiler more than once.

The public methods of the Profiler class are shown in Figure 4.1. The `updateProfiler()` method has to be called by the scheduler periodically to ensure that the Profiler has the most up-to-date information available. The three features of (i) obtaining an estimate of execution time, (ii) obtaining an approximation of the execution time function, and (iii) hypothesis testing are provided using the `getEstimate()`, `getExecTimeFunction()`, and `compareJobEstimates()` methods, respectively. All of these methods identify jobs by the name of the executable and the user who ran the executable. In Chapter 3, it was found that classifying jobs by both the executable name and user name led to relatively low coefficients of variations. Thus, this is a reasonable way of identifying jobs.

The profiler uses two other pieces of information for all its estimates: the attained wall clock time, `attainedWallClock`, and the maximum memory usage, `memSize`. The attained wall clock time is the length of time that the job under consideration has been running, while the maximum memory usage is the maximum amount of memory that it has used thus far in its execution. 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 size has a positive correlation to the execution time of an executable, and the maximum memory used gives an indication of the problem size.

The `getEstimate()` method predicts the execution time for the specified job that uses the specified number of processors, `numProcs`. When the method is called, the `confidenceDesired` input must be a value between zero and one, specifying the percentage confidence desired for the confidence interval. The estimated mean total execution time is returned in `estimate`, while `confidenceIntervalSize` is set to a value equal to half the width of the confidence interval.

The `getExecTimeFunction()` method returns an ExecutionTime object for the specified executable name, user, maximum memory size, and wall clock execution time combination. This object may be used to determine execution time confidence intervals for

updateProfiler()

Inputs:

None

Outputs:

None

getEstimate()

Inputs:

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

Outputs:

Boolean status	float estimate	float confidenceIntervalSize

getExecTimeFunction()

Inputs:

string executionCommand	string user	float memSize
float attainedWallClock		

Outputs:

ExecutionTime ExecutionTimeFunction

compareJobEstimates()

Inputs:

float difference	float confidence	string executionCommandA
string executionCommandB	string userA	string userB
int numProcsA	int numProcsB	float attainedWallClockA
float attainedWallClockB	float memSizeA	float memSizeB

Outputs:

boolean truthValue

Figure 4.1: The Interface to the Profiler Class

estimateTime()

Inputs:

float numProcs	float confidence

Outputs:

float estimate	float error

Figure 4.2: The Interface to the ExecutionTime Class

different processor allocations.

The `compareJobEstimates()` method does hypothesis testing, comparing the projected execution times of two jobs, A and B. Both jobs are identified by executable name, user, maximum memory size, wall clock execution time, and number of processors, as with `getEstimate()`. This method returns a boolean indicating whether with confidence `confidence`, the mean total execution time of job A is greater than or equal to `difference` times the mean total execution time of job B.

The ExecutionTime interface, shown in Figure 4.2, consists of a single method, `estimateTime()`. When this method is called with the specified number of processors `numProcs`, it will return an estimate for the mean total wall clock execution time for the job. If `confidence` is specified to be between zero and one, then `error` will be set to half of the width of a confidence interval based on the value `confidence`.

4.3 The Design

4.3.1 Overview

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 about that executable would be extremely costly. To deal with this difficulty, the application profiler has its own permanent repository to store data in a more appropriate format.

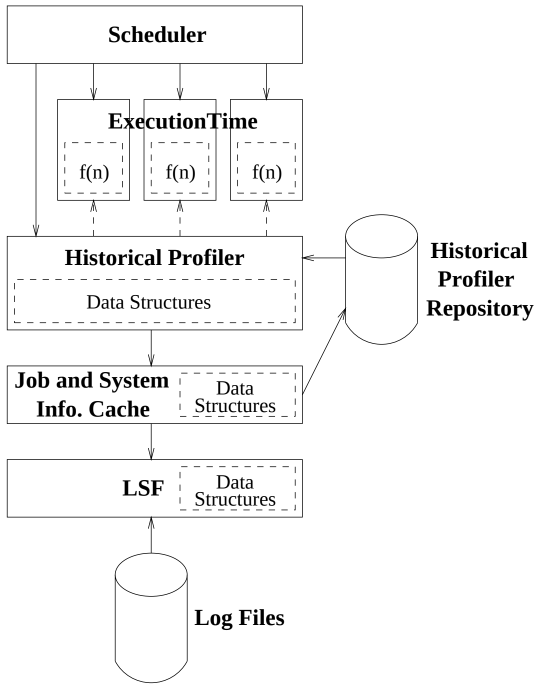

Figure 4.3: High-level Design of the Historical Profiler

Figure 4.3 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 4.3 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. If the scheduler requests a single estimate or the testing of a hypothesis, the result is returned to the scheduler. If, instead, the scheduler requests the execution time function, the Historical Profiler creates an ExecutionTime object for the specified job. Subsequently, the scheduler can request, from the ExecutionTime object, the evaluation of the job's execution time function $T(p)$, where p is the number of processors assigned to the job.

4.3.2 Historical Profiler Repository Design

The Historical Profiler Repository contains all the data used by the Historical Profiler for predicting job execution times. Thus, the design of the repository depends primarily on the data required by the profiler. However, a secondary issue of some importance is minimizing the permanent storage space required for the repository. These two criteria are the primary factors affecting the design of the repository.

As noted in Section 4.2, schedulers request information based on executable name and user name. Thus, since information is always requested based on these criteria, it is natural to index jobs by the combination of executable and user names. Of course, this means that if a user has not run a particular executable, or an executable has never been run before, the profiler will be unable to find information in the repository. To remedy this deficiency, the profiler also stores data for each executable name irrespective of user, and for the entire system, irrespective of both executable and user. As a result, the profiler always has data available for predicting execution times, though if the executable has never been run before, the estimate may have a high error.

Obtaining an approximation of the execution time function for a job requires estimates of the job's execution time for several different processor allocations. In addition, since the interface allows the execution time estimates to vary depending on the current execution time and the memory usage, this data must also be available in the repository. Furthermore, error calculations require knowledge of the number of data points. These factors lead to the development of the structures shown in Figure 4.4.

The repository consists of ExecEntry structures, one for each executable-user pair. Each ExecEntry has an `index`, a `date` when this entry was last modified, a `numExecuted` integer specifying the number of jobs included in this structure, and a three-dimensional RunEntry array, `runTimeTotal`. The `RunEntry` array contains the data used to estimate run times. Its dimensions are based on the three ways the scheduler can further define a particular job: by the number of processors, execution time so far, and memory usage. Although memory usage is unavailable in this version of LSF, it is included here for use with future versions.

To find information about a particular executable, user, number of processors, exe-

RunEntry Structure {

float timeTotal

float timeSquared

float procTotal

float numEntries

}

RunEntryArray Structure {

RunEntry theArray[MAX_MEM_ENTRIES]

}

ExecEntry Structure {

string index

time date

int numExecuted

RunEntryArray runTimeTotal[MAX_PROC_ENTRIES][MAX_EXEC_TIME_ENTRIES]

findRunEntry()

Inputs:

float numProcs float execTime float memSize

Outputs:

RunEntry foundRunEntry

}

Figure 4.4: The Historical Profiler's Internal Design

cution time, and memory usage combination, the Historical Profiler first finds the appropriate ExecEntry structure in the repository using information about the executable and user. Then it calls the `findRunEntry()` method with the number of processors, execution time, and memory usage to find in the `runTimeTotal` array the appropriate `RunEntry` structure containing the desired data.

Each `RunEntry` consists of data for determining an execution time estimate and error for a specified combination of number of processors, execution time, and memory usage. It includes a `numEntries` field for the number of jobs used to obtain this data. The `timeTotal` field is the sum of the execution times of each of the jobs included in this entry, while the `timeSquared` field is the sum of the squares of the execution times. These fields can be used to calculate the average and standard deviation of execution times. For this version of the Historical Profiler, the times in these fields are wall clock times. However, when processor times are available in LSF, these will be more appropriate times to use. A single RunEntry could contain data for different numbers of processors, since each RunEntry consists of a subset of the entire range of processor allocations. As a result, `procTotal` field is required, containing the sum of the number of processors used for all the jobs included in the entry. Combined with the `numEntries` field, this may be used to calculate the average number of processors for all the jobs that fit in this subset of processor allocations.

Each executable and user combination has an ExecEntry structure containing multiple RunEntry structures, each with different average numbers of processors. As a result, it is possible to obtain from these RunEntry structures several execution time and standard deviation estimates for different numbers of processors. If more than two points are available (due to the job being run at least three times with different processor allocations), these estimates may be used to approximate an execution time function. The method of actually approximating the function is discussed in detail in Section 4.3.3.

Thus far, this design leaves several issues unresolved. The first is the size of the `runTimeTotal` array and the related issue of determining the functions to map from the number of processors, execution times, and memory usage to an index this array. Since the array is three-dimensional, increasing the size of one dimension of the array tends to greatly increase the permanent storage space required for the repository. As a result, the natural way of indexing "number of processors" dimension of the `runTimeTotal` array by the actual number of processors is infeasible. Instead, it is necessary to group several processor allocations into a single category. Since speedup curves of jobs tend to have decreasing slope for increasing number of processors (as the inefficiencies due to parallel processing get more and more significant), finer granularity of data is desirable for few processors, while coarser granularity is sufficient for a greater number. As a result, a logarithmic function $\lceil \log(n) \rceil$ for n processors will be used to categorize the data. This function yields categories for one, two, three and four, five to eight, and nine to sixteen processors. Similar logarithmic functions, with different bases, will be used for the other dimensions of the array.

A disadvantage of this design is every job is weighted equally. If the use of a particular executable changes, the information in the repository will be slow to reflect the new usage, and the estimates will be inaccurately reflecting outdated information. It would be more desirable to weight more recent jobs more heavily than older jobs. Unfortunately, such weighting would greatly complicate the calculation of confidence intervals. The calculation of confidence intervals assumes that every job is distributed around an actual mean, and every job is as relevant as every other job. Weighting jobs differently contradicts these assumptions. One possible solution is to scale down the weighting of the existing data by a constant multiplier whenever RunEntry is updated. However, if this multiplier is anything other than one, the error estimates obtained by the current method are invalidated. In this thesis, such a weighting will not be used.

An unresolved difficulty is dealing with jobs whose processor allocations are not constant throughout the execution of the job. This is an important issue, since it is likely that in the future, dynamic schedulers will become more prevalent due to their good performance. This thesis will avoid the issue and focus only on jobs with constant processor allocations.

4.3.3 Methods of Estimating Executable Information

The repository can supply all the data required by the Historical Profiler, but the profiler is required to manipulate this information into a form usable by the scheduler. In particular, the profiler has to be able to make estimates of job lengths and approximations of execution time functions, and do hypothesis testing. Error estimation and hypothesis testing both require some assumptions about the distribution of the data being examined. In this case, the assumption will be that the execution times are Normally distributed about an actual mean. This Normal distribution will be approximated by a Student-t distribution.

Sevcik's execution time function (Equation 2.7) is more general than most alternatives. The execution time function (Equation 2.6) based on Dowdy's execution signature is non-increasing, whereas Sevcik's function is not. The fact that Sevcik's function can increase is particularly important in a distributed environment with high communication overheads, since it is likely that some jobs will slow down when run on too many processors. In order to simplify approximations, the $\phi(p)$ term that appears in Sevcik's function will be approximated by a constant ϕ term. Thus, the execution time function used in approximations will be:

$$T(p) = \phi \frac{W}{p} + \alpha + \beta p \tag{4.1}$$

Least squares is a standard way of approximating functions such as the execution time function. However, the data is not evenly distributed. It could be that a user runs a job once with one processor, and ten times with sixteen processors. In this case, it is inappropriate to give both estimates the same weighting. Thus, a weighted least squares approximation is more reasonable. This method of approximating equations is discussed in detail by Draper and Smith [DS81]. Briefly, this method requires solving the following formula for b:

$$b = (X'V^{-1}X)^{-1}X'V^{-1}Y$$

where

$$b = \begin{bmatrix} \phi W \\ \alpha \\ \beta \end{bmatrix}, Y = \begin{bmatrix} t_1 \\ \vdots \\ t_n \end{bmatrix}, X = \begin{bmatrix} p_1^{-1} & 1 & p_1 \\ \vdots & \vdots & \vdots \\ p_n^{-1} & 1 & p_n \end{bmatrix}, V = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_n^2 \end{bmatrix}$$

The execution time function has three terms. The three terms of the b matrix for which we solve are the three coefficients in Sevcik's execution time function. The Y matrix contains n observed execution times with different processor allocations, $p_1, ...p_n$. The X array contains the evaluation for the specified processor allocations of the indeterminates in the three terms in Sevcik's function. Thus, since the coefficients in Sevcik's function for which we are solving are multiplied by $\frac{1}{p}$, 1, and p, X contains the a row containing the evaluation of these entries for each observed execution time. For instance, if the execution time for the executable were available from when a job was run on two processors, one row of X array would be $[\ \frac{1}{2}\quad 1\quad 2\ ]$. V is a diagonal matrix containing standard deviations of the execution time estimates in Y. Error bounds and confidence intervals for equations derived by this formula may be calculated. With an allocation of p_0 processors, a δ confidence interval for an execution time estimate Y_0 based on n observations is:

$$Y_0 \pm t(n, \frac{1+\delta}{2})\sqrt{X_0'(X'V^{-1}X)^{-1}X_0}$$

where:

$$t(n, \frac{1+\delta}{2})\text{: Student-t function}$$

$$X_0 = \begin{bmatrix} p_0^{-1} \\ 1 \\ p_0 \end{bmatrix}$$

To further clarify the procedure used to obtain these estimates, an example is useful. Suppose an executable were run a total of sixty times, some of which were with one processor, some with four processors, and some with eight processors. Furthermore, suppose the observed average execution times with each of these processors were 8350, 2500, and 1700 seconds respectively and the observed standard deviations were 4175, 1250, and 850 respectively. Then:

$$Y = \begin{bmatrix} 8350 \\ 2500 \\ 1700 \end{bmatrix}, X = \begin{bmatrix} 1 & 1 & 1 \\ \frac{1}{4} & 1 & 4 \\ \frac{1}{8} & 1 & 8 \end{bmatrix}, V = \begin{bmatrix} 4175 & 0 & 0 \\ 0 & 1250 & 0 \\ 0 & 0 & 850 \end{bmatrix}$$

We can then calculate b to be:

$$b = \begin{bmatrix} \phi W \\ \delta \\ \beta \end{bmatrix} = (X'V^{-1}X)^{-1}X'V^{-1}Y$$

$$b = \left(\begin{bmatrix} 1 & \frac{1}{4} & \frac{1}{8} \\ 1 & 1 & 1 \\ 1 & 4 & 8 \end{bmatrix} \begin{bmatrix} \frac{1}{4175} & 0 & 0 \\ 0 & \frac{1}{1250} & 0 \\ 0 & 0 & \frac{1}{850} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ \frac{1}{4} & 1 & 4 \\ \frac{1}{8} & 1 & 8 \end{bmatrix} \right)^{-1} * \begin{bmatrix} 1 & \frac{1}{4} & \frac{1}{8} \\ 1 & 1 & 1 \\ 1 & 4 & 8 \end{bmatrix} \begin{bmatrix} \frac{1}{4175} & 0 & 0 \\ 0 & \frac{1}{1250} & 0 \\ 0 & 0 & \frac{1}{850} \end{bmatrix} \begin{bmatrix} 8350 \\ 2500 \\ 1700 \end{bmatrix}$$

$$b = \begin{bmatrix} 8000 \\ 300 \\ 50 \end{bmatrix}$$

The approximation for the execution time function for this executable would be:

$$T(p) = \frac{8000}{p} + 300 + 50p$$

Thus, the estimate for the amount of time the job would take to complete with 16 processors would be 1600 seconds. A 95% confidence interval C for the estimate is:

$$C = Y_0 \pm t(n, \frac{1+\delta}{2})\sqrt{X_0'(X'V^{-1}X)^{-1}X_0}$$

$$C = 1600 \pm t(60, \frac{1+0.95}{2}) * \sqrt{\begin{bmatrix} \frac{1}{16} & 1 & 16 \end{bmatrix} \left(\begin{bmatrix} 1 & \frac{1}{4} & \frac{1}{8} \\ 1 & 1 & 1 \\ 1 & 4 & 8 \end{bmatrix} \begin{bmatrix} \frac{1}{4175} & 0 & 0 \\ 0 & \frac{1}{1250} & 0 \\ 0 & 0 & \frac{1}{850} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ \frac{1}{4} & 1 & 4 \\ \frac{1}{8} & 1 & 8 \end{bmatrix} \right)^{-1} \begin{bmatrix} \frac{1}{16} \\ 1 \\ 16 \end{bmatrix}}$$

$$C = 1600 \pm 2.0 * \sqrt{8006.48} = [1421, 1779]$$

In the Historical Profiler, ExecutionTime objects encapsulate data for solving these equations. An ExecutionTime object stores b, the execution time function which is the solution to Equation 4.3.3, and the $(X'V^{-1}X)^{-1}$ term of Equation 4.3.3. The Execution-Time objects can use this information to estimate the execution time and a confidence interval for a job using a specified number of processors.

If a scheduler requests an execution time estimate rather than the execution time function, the Historical Profiler calculates the execution time based on an execution time function approximation. It uses the executable name, user name, current execution duration, and maximum memory size inputs to find the three or more data points required to calculate the execution time function. Then the execution time function is evaluated for the specified number of processors to obtain the execution time estimate and confidence interval that are returned. It could be that there is insufficient data to approximate an execution time function, but there is data available for a specified number of processors. This could happen if a user only runs a particular job using the same number of processors each time. In this case, an estimate and confidence interval for the execution time are calculated using standard techniques. The estimate is the average execution time for that number of processors, while the confidence interval is estimated using an assumption of a Student-t distribution and the standard deviation of the observations. If there is still not enough data, the data for the executable including all users is used to provide an estimate. If there is still insufficient information, then all executables ever run in the system are used.

This method of estimating run times has the major disadvantage that the estimate of the remaining run time could be less than the run time already attained. This is unreasonable, particularly considering that information about the attained run time is available and used by the Historical Profiler when making its predictions. In this case, if the sum of the estimate E and three times the confidence interval requested is still less than the run time already attained, R, a new estimate E' will be used:

$$E' = \lceil \frac{R}{E} - 0.5 \rceil * E$$

Thus, the estimate increases in multiples of E as the job duration increases.

Hypothesis testing comparing the mean execution times for two jobs identified by name, user, number of processors, current execution duration, and memory usage is the remaining feature provided by the historical profiler. The mean execution times and standard deviations are calculated for the specified jobs with the specified allocations of processors. Using the assumptions that the distributions of run times are Normal, standard statistical techniques using the Student-t distribution determine whether to accept the hypothesis. Allen's formulation [All78] of these equations follows. The two jobs, x and y have, based on n and m observations, mean execution times estimates of $\bar{x}$ and $\bar{y}$ and sample standard deviations of s_x and s_y. The hypothesis is that the difference in the actual mean execution times is greater than d_0, or $\bar{\mu_x} - \bar{\mu_y} > d_0$. With confidence δ, the hypothesis is true if $t > t(v, \delta)$, where $t(\delta, v)$ is the Student-t function, and

$$t = \frac{\bar{x} - \bar{y} - d_0}{\sqrt{(s_x^2/n) + (s_y^2/m)}}$$

$$v = \frac{((s_x^2/n) + (s_y^2/m))^2}{(s_x^2/n)^2/(n-1) + (s_y^2/m)^2/(m-1)}$$

For example, suppose x and y have been run 30 and 40 times respectively, and are observed to have mean execution times of 700 and 300 seconds and standard deviations of 900 and 500. If we want to determine if x is 95% likely to have an actual mean execution time more than 300 seconds greater than y, we calculate t and v to be:

$$t = \frac{700 - 300 - 300}{\sqrt{(900^2/30) + (500^2/40)}} \approx 0.5484$$

$$v = \frac{((900^2/30) + (500^2/40))^2}{(900^2/30)^2/(29) + (500^2/40)^2/(39)} \approx 42.29$$

Since 0.5484 is less than $t(42.29, 0.95) \approx 1.682$, the hypothesis is not supported. A similar hypothesis would be true if a confidence of only 60% were requested, since $0.5484 > t(42.29, 0.95) \approx 0.255$.

Chapter 5

Schedulers that Use Application Knowledge

In order to evaluate the usefulness of the Historical Profiler to schedulers, it is necessary to compare the performance of various scheduling algorithms. In this thesis, only static algorithms, algorithms that do not permit applications to change the number of requested processors, will be examined. In addition, all of the algorithms will be space-sharing algorithms; jobs will have exclusive access to the processors they are assigned. Eight variants of three basic algorithms will be compared:

1. First Come, First Serve (FCFS)
2. First Come, First Serve Fill (FCFS-fill)
3. Least Estimated Work First (LEWF)
4. Least Estimated Work First Fill (LEWF-fill)
5. Least Estimated Remaining Work First (LERWF)
6. Least Estimated Remaining Work First Fill (LERWF-fill)
7. EASY-kill
8. EASY-preemptive

Although many of the complications of implementing scheduling algorithms in a NOW environment are overcome by developing it on top of the LSF and JSIC layers, some difficulties are simply abstracted to a higher level. Instead of making the UNIX hosts perform the actions specified by the scheduling algorithm, the problem becomes making LSF perform the actions specified by the scheduling algorithm. Additionally, there are sometimes distributed systems issues that are not handled by LSF or the JSIC.

This chapter will discuss issues associated with designing and implementing each of these scheduling algorithms in the NOW environment described in Section 4.1. First, the general design that all the algorithm have in common will be discussed in Section 5.1. Then each of the algorithms will be defined and their use in this context will be justified. Finally, in Section 5.7, the heuristics used by preemptive disciplines to assign processors will be discussed.

5.1 Structure of Algorithms

All the non-preemptive algorithms have a very similar high-level design. All the algorithms have two queues, a pending queue and a running queue. No jobs in the pending queue execute, while all jobs in the running queue[1] execute. Jobs are initially submitted to the pending queue. Whenever a job finishes, the algorithm sorts the pending queue and selects the first job in this queue. If sufficient processors are free to run the job, the job is scheduled by assigning the hosts for the job, and moving it to the running queue. The pseudo-code for this algorithm is shown in Figure 5.1.

There are two different ways this general scheduling algorithm can be modified to create specific scheduling algorithms. First, the sort procedure on line 3 can be modified. By changing the sorting method, different algorithms can be created. For instance, if the sort routine sorts the jobs in increasing order of submission times, the scheduling discipline will be First Come First Serve.

The second way of changing the functionality of the algorithm is by setting the `fillingOn` flag (in lines 6 and 17) to `TRUE`. If this flag is `FALSE`, the algorithm will schedule the jobs in strictly the order specified by the sorting algorithm. For instance, if the sorting algorithm has a 1-processor job followed by a 16-processor job and there

[1]Since all the jobs in the running queue execute, this queue actually functions as a set; all jobs have the same priority and are treated the same way. However, this set is implemented using an LSF queue, so, for consistency, the queue terminology will be used.

```
while (TRUE) {
   numFreeProcessors = findNumFreeProcessors()
   sortJobs(pendingQueues)
   currJob = getFirstEntry(pendingQueue)

   while (fillingOn == TRUE AND
          getNumProcessors(currJob) > numFreeProcessors)
       currJob = getNextEntry(pendingQueue)

   while (getNumProcessors(currJob) <= numFreeProcessors) {
      assignFreeHosts(currJob)
      removeFromQueue(currJob, pendingQueue)
      addToQueue(currJob, runningQueue)
      numFreeProcessors = numFreeProcessors - getNumProcessors(currJob)
      currJob = getNextEntry(pendingQueue)

      while (fillingOn == TRUE AND
             getNumProcessors(currJob) > numFreeProcessors)
         currJob = getNextEntry(pendingQueue)
    }

    waitForJobCompletion()
}
```

Figure 5.1: Pseudo-code for Scheduling Algorithms

are 16 processors in the system, the 1-processor job will run, the 16-processor job and all other jobs in the queue will be blocked, leaving 15 processors idle. If this flag is true, the order of activation will no longer be strict. Instead, if there are available processors but the first job in the sorted pending queue requires more processors than are available, the algorithm will search through the pending queue looking for another job that can use those processors. If it finds such a job, it starts that job immediately. So according to this algorithm, if there are 16 processors in the system, and the five jobs in the ordered queue require one, sixteen, nine, seven and six processors respectively, the jobs requiring one, nine and six processors are started.

LSF and the Job and System Information Cache provide much of the basic functionality required to implement the scheduling algorithm. Initial submittal of the jobs can be done by users through LSF's batch interface. Since the JSIC queries LSF to obtain information about the state of the system whenever a job changes state (arrives, finishes, suspends, resumes, etc.), determining the number of free processors (as in line 2 of the algorithm) is a simple function call to the JSIC. The combination of the JSIC and LSF also contains the queue management functionality required in lines 4, 7, 11, 12, 14 and 17, as well as a way to assign specific processors to jobs, as in line 10. As would be expected from load-balancing software, LSF supports the remote execution of parallel jobs so that jobs are started when they are moved to the run queue.

The preemptive algorithms have a similar structure, but the pseudo-code for the algorithms becomes quite complex. Instead of sorting only the jobs in the pending queue, the jobs in both queues are given an absolute ordering in the system. Then it is determined, using a similar method of allocating processors in order of the jobs in the queues, which jobs are to be running and which jobs are not. If a job is not supposed to be running but currently is in the run queue, the job is moved to the pending queue and suspended. If a job is supposed to be running, but is currently in the pending queue and has never been started, it is assigned processors and moved to the run queue to start it. If a job is supposed to be running, but is currently suspended, and the processors on which it was running are free, the job is resumed and moved to the running queue. Thus, overall the algorithm is mostly the same, except for some extra details for preemption and resumption, and the fact that all the jobs in the system are examined rather than just those in the pending queue.

5.2 First Come, First Serve

FCFS is a very basic run-to-completion scheduling algorithm. Arriving jobs are placed in a first in, first out queue. The first job in the queue is blocked (as are all jobs that arrived later than that job) until at least as many processors as it requested are free. The job is then removed from the queue and run on the requested number of processors. The whole process is repeated for the next job in the queue whenever a job is completed and there are new processors available for assignment.

This algorithm is attractive for its simplicity. However, it has the disadvantages of generally having both poor efficiency and poor mean response times (since, in the no-fill version, a running job requiring a single processor can block the entire queue if the first job in the queue requests every processor). However, in the context of the thesis, despite the fact that it does not use application knowledge, this algorithm is worthwhile for comparison for several reasons. First, it is a simple, well-known baseline for comparing algorithms. Second, it is very similar to the EASY-kill scheduler, so comparisons between these two algorithms are interesting. Finally, the algorithm is used in some production systems.

The design and implementation of this scheduler is simple; it has the same basic structure of Figure 5.1. In order to ensure the FCFS property, the sorting algorithm that is used on the third line of this algorithm sorts in order of increasing submission time. For the basic FCFS algorithm, `fillingOn` is set to `FALSE` so that there is strict FCFS ordering. In order to get the filling version of the algorithm, FCFS-fill, the `fillingOn` flag is set to `TRUE`. For a description of the effects of filling, see Section 5.1.

5.3 Least Estimated Work First

Least Estimated Work First is a run-to-completion policy that is similar to FCFS, but uses a different queuing order. Instead of ordering jobs by submission time in line 3 of

Figure 5.1, jobs are ordered by increasing estimated execution time. The result is that jobs expected to be short are run first, while jobs expected to be long wait for all the short jobs to finish.

This strategy is worthwhile because it leads to a low mean response time in many situations, and provably optimal average response times in some situations (e.g. scheduling on sequential machines and scheduling parallel applications with perfect speedup that can be assigned any number of processors [Sev94]). This algorithm is appropriate for study in this context because it requires some knowledge of the characteristics of the applications.

The design of the LEWF algorithm is largely the same as general scheduling algorithm. The only additional issues that are raised by this algorithm are the methods of estimating the execution time. There are many different ways of obtaining estimates, including having the user inform the scheduler, having the application inform the scheduler itself, and obtaining the data from an outside source such as the Historical Profiler. In this thesis, two methods will be used to obtain estimates by all algorithms that require them. The first will be the application giving the scheduler perfectly accurate information about its run time. The second will be using the Historical Profiler. The estimate from the profiler will be calculated by requesting a confidence interval for the execution time for the job. The estimate will be the sum of the estimate and half the 95% confidence interval. In other words, it will be equal to the greatest value in a confidence interval that has a 95% chance of including the mean.

As with FCFS and FCFS-fill, there is a filling version of LEWF called LEWF-fill. This is the same algorithm as LEWF, but has the `fillingOn` flag set to `TRUE`, whereas with LEWF, this flag is `FALSE`. This flag ensures that a job will not be blocked if there are sufficient processors available to run the job, as described in Section 5.1.

5.4 Least Estimated Remaining Work First

Least Estimated Remaining Work First is similar to LEWF, but instead of being run-to-completion, LERWF is a preemptive discipline. Whenever a job finishes or a new job arrives, all the jobs in the system are ordered by estimated *remaining* execution time.

Then, in order, jobs are assigned their requested numbers of processors until not enough processors are available to fulfill the requirements of the next job in the queue. If the first job in the queue cannot run, all the jobs with estimated execution duration greater than that job will be blocked. Any jobs that were previously running, but are no longer assigned processors are preempted. Any preempted jobs that are reassigned processors are resumed.

As would be expected, this algorithm has many of the benefits of LEWF, with the additional benefit of allowing long running jobs to be preempted to make way for newly-submitted jobs. It is particularly interesting for use with the Historical Profiler, since the Historical Profiler's job duration predictions increase as the length of the job increases. At the same time, the actual work done increases, so the amount of work remaining decreases. The difference between the two is likely to lead to estimates that decrease as the job is run. However, as soon as the job duration becomes large enough that the profiler excludes a set of data for jobs that were shorter than the current duration, the estimate will increase suddenly before resuming the decreasing trend. The result may be similar to a multi-level feedback queue, where a job runs at a high priority for a few minutes, but priority decreases as the job ages.

The addition of preemption to the LEWF algorithm to form LERWF leads to some complications. The process of preemption is relatively easy. The job is moved to the pending queue for preemption, and back to the running queue for resumption. LSF will suspend any job that is moved to the pending queue. However, there is no migration for parallel jobs, only preemption. As a result, not only do there have to be the required number of processors available when a paused job continues, but the available processors must include the processors on which the job was running originally.

There are several methods of ensuring that, with high probability, the original processors are available. One would be to find the optimal scheduling so that the correct processors are available when they are required. With multiple suspended jobs and multiple queued and running jobs, this solution is both complicated and computationally expensive. Instead, a simple heuristic approach can be used. The processors that are needed the soonest by the pending jobs will be the last to be assigned to new jobs. As a result, the processors required by the next suspended job will have a higher probability of being free when the job is ready to run. However, if a paused job is scheduled to run, but at least one processor it requires is in use, the paused job (and all other jobs in the pending queue) will be delayed until the processors become available. This heuristic is not optimal, but it is both simple and computationally feasible. One alternative to this heuristic is discussed in Section 5.7.

One additional overhead of preemption is specific to the use of LSF. Whenever a preempted job is resumed, it takes some time for LSF to restart that job. It requires up to 30 seconds from the time a suspended job is first moved to the run queue to when it actually starts running. This extra overhead is not negligible, particularly with an algorithm like LERWF, where a single long-running job can potentially be preempted and resumed many times as shorter jobs arrive.

As with all the other algorithms already described, there is a filling variant of the LERWF algorithm, LERWF-fill. Section 5.1 describes how this filling algorithm is different from the regular algorithm.

5.5 EASY-kill

Lifka's EASY scheduler [Lif95], discussed in Section 2.4, is a FCFS with backfilling algorithm which has proven popular in some multiprocessor environments. Users appreciate its simplicity and predictability. The perceived "fairness" and the benefits of knowing the latest time that a job will start compensates for the inefficiencies arising from the scheduling strategy. In Lifka's implementation, users are required to provide estimates of job durations, and if jobs exceed these durations, they are killed. This implementation will be similar, but perfectly accurate "estimates" and Historical Profiler estimates will be used instead of user estimates. The algorithm will, like Lifka's algorithm, kill jobs that exceed their predicted run times.

For several reasons, this algorithm is of interest. The first reason follows from the results of the chapter 3. In some multiprocessor systems, parallel jobs make up almost 50% of the workload, while in the NOW environment, parallel jobs are still relatively infrequent (11% in one environment we examined and less than 1% in the other.) The availability of a popular and familiar multiprocessor scheduler in the NOW environment could encourage users to view the NOW as a parallel system, rather than as just workstations. This may make them more comfortable running parallel jobs in the environment. A second reason is that, thus far, the EASY algorithm has not been implemented in a NOW environment, so the issues associated with a distributed implementation have not been explored. Also, EASY is suited to a distributed environment because it does not require preemption, migration, or reallocation of processors. A final reason is that with EASY, whenever a user submits a job, a maximum limit on the execution time must be provided. The scheduler uses this data to predict when jobs can be scheduled, and to determine when it can perform backfilling without delaying previously submitted jobs.

It should be noted that this algorithm is very similar to FCFS-fill, but slightly different. In FCFS-fill, jobs will not be blocked from running if there are sufficient processors available to start the job. With EASY, the backfilling property is slightly different. Jobs will only be backfilled if they do not delay a previously submitted job. Thus, in EASY, a job A can be blocked from running even if there are sufficient available processors to start the job. This will happen if starting job A now will result in a previously submitted job B being unable to start at some later time (due to job A running on the processors required by job B when B is scheduled to start). If EASY and FCFS-fill are presented with the same set of jobs submitted at the same time, EASY might not immediately run all the jobs that FCFS-fill does, even if there are processors available. However, if there are processors available, FCFS-fill will not delay starting any job that EASY starts.

5.5.1 High Level Design

Since this algorithm is more complicated than the other algorithms, a discussion of the design of the algorithm requires more detail. Figure 5.2 shows the structure of EASY. The idealized model of EASY as a single queue where jobs are ordered for execution using a FCFS algorithm with backfilling is replaced with a two queue model, as with the other algorithms discussed. The two queues appear at the bottom of Figure 5.2. The Job and System Information Cache and LSF layers are the next two layers, as they are in the Historical Profiler. As before, they supply data to the scheduler and save

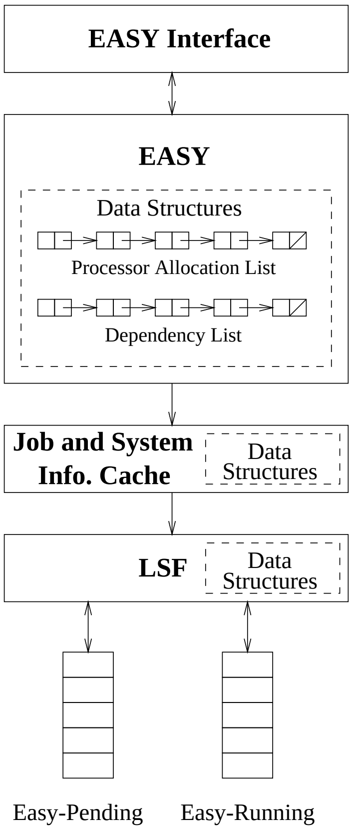

Figure 5.2: The EASY Scheduler

information for quick access. Above the JSIC is the EASY scheduler itself. It has its own data structures, the Processor Allocation List and the Dependency List, which are discussed in Section 5.5.3. One additional layer not required by the other algorithms is the EASY Interface layer. This layer permits users to query the scheduler to find information specific to the EASY scheduler that is unavailable through LSF.

The overall structure of the algorithm is very similar to the algorithm presented in Figure 5.1 with the `fillingOn` flag set to `TRUE`. The sort algorithm of line 3 would sort in order of increasing submission time, as with FCFS. The primary difference is in the filling. Instead of starting a job when it is not on the front of the queue, but there are sufficient processors available, more analysis has to be done. The schedule of prior submitted jobs has to be deduced. If scheduling that job to run now would mean that a prior submitted job would be delayed because it requires a processor that would be in use by the current job, the current job cannot be started.

5.5.2 Interface

The functionality provided by LSF is sufficient for the interfaces of the other disciplines. However, EASY has several features which LSF does not support directly:

- **getjid**: returns the job id of the job running on the specified node.
- **spfree**: returns the status of the various nodes. Since EASY and LSF have different views of the system, the status according to LSF can differ from the status according to EASY.
- **spwhat**: returns the number of processors currently free, and the amount of time for which they will be free.
- **spwhen**: for an existing job, returns the latest time that job will start executing. For a hypothetical job specified by a number of processors and a maximum execution time, returns the latest time for that job to start executing.
- **spq**: returns a list of all the jobs in the system, with their estimated or actual start times.

- **spusage**: returns a list of each node in the system, the user currently using the node, and the time by which the user will finish using the node.

Perhaps surprisingly, implementing this interface is one of the most difficult tasks in creating EASY in a NOW environment. The difficulty arises due to the distributed nature of the system. There is a single scheduling algorithm for the entire system on one machine in the system. However, users on different workstations have to be able to interface with this central scheduler. Thus, the EASY Interface module is required to allow the users of individual workstations to communicate with the single EASY scheduler. The EASY Interfaces must query LSF to determine on which workstation the EASY scheduler is running. LSF has this information since LSF is responsible for starting the EASY scheduler originally. The EASY interfaces can then establish their own socket interfaces to the scheduler for communication.

Despite these interface issues, LSF and the Job and System Information Cache are still able to handle many of the other interface requirements. For example, Interface issues such as job submission and queue management, handled by these layers for the FCFS algorithm discussed in Section 5.2, are also relevant for EASY.

5.5.3 Implementation Issues

The primary goal when implementing EASY was for the implementation to be robust, while the secondary goal was efficiency. Robustness is particularly important in a distributed environment where workstations and networks can fail. The algorithm is implemented so that whenever a job finishes, every job in the entire system is rescheduled. This strategy is required, since if a job finishes prematurely, it could potentially change the time when every other job is scheduled to start. This feature also ensures that any changes in the scheduling strategy due to temporary errors in the system are corrected as soon as any job finishes.

EASY has two primary structures to provide functionality that is unavailable in the JSIC or LSF layers, the processor allocation list and the dependency list. Due to efficiency concerns, it is undesirable to reschedule the entire system whenever a new job arrives, or a user queries the system. As a result, it is necessary for EASY to maintain a processor allocation list, from which the current schedule can be deduced. The processor allocation list may be quickly examined to determine information such as when the usage of particular processors is scheduled to change.

The second internal structure retained is the dependency list. With EASY, it is possible to specify that a job cannot start until another job has finished. Although LSF provides this feature when it is doing the scheduling directly, it is not in our context, so it is necessary to duplicate this functionality inside EASY using the dependency list. This list contains all the jobs that cannot start until some other specified job finishes. Whenever the system is rescheduled, EASY must consult this list to ensure that it does not start a dependent job before the prior job finishes.

5.6 EASY-Preemptive

EASY-preemptive is very similar to EASY. The algorithm is the same, except that instead of killing a job when its time expires, EASY-preemptive preempts the job and moves it to the back of the FCFS queue, just as if it had been resubmitted. The estimate of the duration of the job is set to its original value. Then, instead of starting the job at the appropriate time, EASY resumes the job. As a result, the predictability of EASY is maintained, but jobs are not killed.

This algorithm is included in this study for several reasons. The primary reason is that using the Historical Profiler with EASY has one major disadvantage. When the maximum execution time elapses, EASY kills the job. However, the Historical Profiler is unlikely to always make perfect predictions; it is probable that jobs will arise that run longer than the duration predicted by the profiler. Hence, some jobs will be prematurely killed. The EASY-preemptive scheduler attempts to remedy this problem. This algorithm may still not be usable in practice, since swap space is required for the preempted jobs, but it is an improvement over EASY. A second reason this algorithm is examined is to compare this algorithm to both LERWF, the only other preemptive algorithm, and the original EASY algorithm.

The design of the EASY-preemptive algorithm is essentially the same as the design for EASY. The main difference is that the scheduler has to store information about the preempted jobs so that it can resume them at the appropriate time on the same processors that they originally were using. The heuristic used when assigning processors is identical to the algorithm than for LERWF. Newly started jobs are assigned the processors required by the suspended job that will run next only if there are no other processors available. If a job cannot resume because a processor it was using is unavailable, it waits until the processor is available.

EASY and EASY-preemptive both already have their own method of backfilling. Consequently, it does not make sense to have special "fill" versions of these two schedulers.

5.7 Processor Assignment Heuristics for Preemptive Disciplines

The preemptive algorithms currently use a simple heuristic to determine which processors to assign to which jobs. For any job that is about to run, the scheduler attempts to avoid assigning any processor required by the next preempted job in the pending queue. This strategy helps to ensure that these processors are not in use when the preempted job is supposed to run. In an attempt to improve the performance of the preemptive algorithm, a slightly different method for assigning processors is tested.

The problem with the original policy is that it does not take into account the total number of processors in the system when assigning processors to a new job. Suppose there are a total of sixteen processors. The suspended job with the least work requires nine, and the job to be started requires eight. It is of no use to avoid assigning the processors required by the suspended job to the new job because the two jobs cannot ever run concurrently anyway. Avoiding allocating them in this case has no benefit, and can actually hurt.

Suppose that you have the jobs in Table 5.1 with a LERWF-fill algorithm using this poor method of allocating processors. First A is submitted and started on processors 1 to 3. Then B is; it preempts A and is assigned processors 2 to 16, avoiding processor 1 since it is required by A. Now suppose that C is submitted. B is the next job to run, so C has to avoid the processors that B wants to use. So it has to use processor 1, since it is the only processor that B has not been allocated. So C is allocated processors 1, and 4 to

Table 5.1: An Inefficient LERWF Allocation Procedure

Job	Remaining Work	# Processors	Processors Assigned
A	300	3	1-3
B	200	15	2-16
C	100	7	To be determined

Table 5.2: LERWF Assigning Processors to Jobs

Job	Remaining Work	# Processors	Processors Assigned
A	300	2	15-16
B	200	7	7-14
C	150	2	13-14
D	100	11	1 - 11
E	50	6	To be determined

9. Despite the fact that it should be possible to run jobs A and C at the same time since there are enough free processors, with this processor allocation it is not possible, because both A and C require processor 1. This is a problem that is particularly noticeable with the fill variant of LERWF where many long jobs requiring few processors are started due to the filling.

One alternative method of assigning processors uses two rules. First, whenever a new job is assigned processors, the suspended jobs are first examined to see which jobs can run immediately, at the same time as this job. If there are such jobs, the processors required for those jobs are assigned last to this job. If there are are still more processors available than can be used for both the current job and the suspended jobs that can be immediately run, the processors used by suspended jobs that could *potentially* be run at the same time as the current job are assigned last.

An example can help to clarify these policies. Suppose we have jobs in Table 5.2 in a system with sixteen processors, numbered from one to sixteen. Job E is the job that has just arrived. Before it arrives, Jobs C and D will be running, since they have the least remaining work, and they can run at the same time. Job A will also be running, since its processors are not being used by C or D. When E arrives, it is noted that E and D cannot run concurrently, because seventeen processors would be required. D has more work remaining, so it is suspended. Now the problem is determining which processors to assign to E. First, it looks at the jobs that could potentially be run concurrently. A, B, or C can be. C has the least remaining work, so it should definitely be resumed; therefore, E will not be assigned the processors required by C. If both E and C are running, then B cannot be resumed, since B requires a processor, number 14, that will be in use by C. This leaves A, which can be resumed, so the processors assigned to A cannot be assigned to E. So after the first set of decisions, A can be assigned any processors except numbers 13 to 16.

Next, it is necessary to consider the second rule that processors assigned to jobs that could *potentially* be run concurrently with E should not be assigned. Obviously, D cannot ever run at the same time as E, since there are insufficient processors. However, if C were to finish, E and B could run at the same time. Therefore, E should also not be assigned the processors required by B, processors 7-12. At this point, there are no more jobs that could potentially be started at the same time as E, so the processors that have not already been eliminated are assigned to E, up to the number requested by E. So E would end up running on processors 1-6. If B had required eight processors rather than seven, and the extra processor was not one being used by jobs A or C, E would have been forced to use one of the processors required by B, since E has to be started.

This alternative method initially appears as though it could be an improvement over the other algorithm, since it appears to increase the probability that small jobs can run concurrently in the system. However, some informal experimentation indicates that this is *not* true. In fact, this alternative method of assigning processors has a worse performance, by 5 to 25%. The cause of this seems to be that this strategy tends to assign the same processors to all the jobs requiring few processors. If there is a long running job requiring few processors that is at the front of the queue, all the other jobs that are run in the mean time end up being assigned the same processors. So when the long job finishes, none of the remaining jobs are able to run concurrently. The original algorithm does not suffer from this problem to the same extent. Instead, near the end of the test, most of the processors are required by approximately the same number of preempted jobs.

It is clear from these results that the particular heuristic a preemptive algorithm uses for assigning processors can affect the performance of the algorithm a great deal. The original algorithm appears better according to experimental results, and is the algorithm used in the rest of this thesis. Further research and experimentation is still required to find a better heuristic.

Chapter 6

Evaluation of Algorithms

This chapter discusses the methods of testing the performance of the schedulers presented in the previous chapter, and presents the results of such tests. The primary criterion by which performance will be judged is mean response time. The response time for a job is the amount of time the job spends in the system, from the time it is submitted until it completes. The version of EASY that kills job will be excluded from the mean response time calculations, since the killed jobs make comparisons meaningless. Secondary factors that will be examined include the utilization, the total time required to process all jobs, the number of jobs killed and suspended, and the number of times any particular job is suspended. The utilization is the average percentage of processors running jobs during the test.

Section 6.1 discusses the methods of evaluating the different scheduling algorithms. The test workload, the arrival rate, and the data initially available to the profiler are examined. Section 6.2 presents and analyses the results of the experiments.

6.1 Method

There are several factors that can affect the performance of the scheduling algorithms and comparisons among these algorithms. This section will discuss the factors that can affect performance and how these factors will be dealt with when testing the algorithms. It will discuss the characteristics of test workload and the issues associated with the Historical Profiler. A summary of the parameters used in the experiments is presented in Table 6.1.

Table 6.1: 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

One of the main factors that can affect the apparent performance of a scheduler is the test workload. For instance, FCFS and LEWF algorithms perform identically if every job is exactly the same, but very differently if there is a high variance in execution times. The workload consists of parallel applications that can either be real or synthetic. Real applications are somewhat simplified versions of typical applications that are used in production environments. The SPLASH-2 benchmarks [WOT+95] would be an example of this type of application. Synthetic applications are special applications that are designed to resemble real applications, but have less complexity. Therefore, their characteristics are both predictable and modifiable. We choose to use synthetic applications due to their flexibility and predictability. Because of these traits, it is relatively easy to run several tests with the assurance that each scheduling algorithm treats precisely the same set of jobs with the same characteristics. A difficulty with choosing to experiment with real applications is that few parallel algorithms have been ported to run reasonably on NOWs.

In the experiments, one synthetic application with characteristics that can be specified by command line parameters will be used to simulate multiple applications. This synthetic application is a simple program that takes as parameters a Dowdy fraction parameter, the total work in seconds, and the processors on which to run. The job is initially started on one processor. It spawns identical threads on all the other processors.

All the threads go to sleep for a period of time t. After the sleep time elapses, all the threads of the job terminate and the job finishes. The sleep time t is calculated as follows:

$$t = W * (D + \frac{1 - D}{p}) \tag{6.1}$$

where W is the total work parameter, D is the Amdahl's fraction sequential "parameter" (discussed in section 2.3), and p is the number of processors parameter. This equation is a variant of Dowdy's execution time function (Equation 2.6), derived by setting $W = C_1 + C_2$ and $D = \frac{C_1}{W}$. By varying the parameters, this application can simulate applications with different amounts of work and different efficiencies. Since all of the scheduling algorithms provide exclusive access to machines, the scheduling results are not affected by the fact that none of the jobs do real work. The jobs act as a "place-holders", indicating which processors are in use. This allows real use of the system to continue during the tests without affecting the results significantly.

Since the Historical Profiler uses information based on executable name for job predictions, the test workload is required to have different executables. In this case, there are thirteen executables, each an instance of the synthetic application. Each executable has a fraction sequential with value 0.001, 0.01, or 0.1, a unique average amount of work, and a unique coefficient of variation in work. In order to obtain realistic estimates of the work and variance in work, the NASA Lewis workload is used. This set of log files is chosen because it is a NOW system, and contains more parallel jobs than the other NOW system.

The work parameter to pass to the synthetic application to simulate the different executables is derived from these NASA Lewis log files as follows. The first twelve executables of the workload have mean work equal to $\frac{1}{120}$ of the product of the number of processors and the mean execution time for the twelve most frequently run parallel executables. The thirteenth job comprises all the other parallel executables that were run in the system (approximately 35% of the jobs). The $\frac{1}{120}$ scaling factor is used so that a test with many jobs finishes in a reasonable amount of time. Of course, this scaling substantially increases the impact of the scheduling overhead on the results of the tests.

In addition to the mean work, some variability in the applications' run times is re-

Table 6.2: Artificial Workload: Executable Specifications

Executable	Fraction (%)	Mean time (s)	C.V.	Frac. Sequential D
Test1	14.4	5778.8	1.9	0.1
Test2	14.4	106.9	3.7	0.01
Test3	11.6	6.2	2.1	0.001
Test4	4.0	165.7	0.8	0.01
Test5	3.8	703.2	1.4	0.001
Test6	3.5	122.0	1.1	0.1
Test7	2.8	184.9	1.0	0.01
Test8	2.5	4980.4	1.5	0.1
Test9	2.3	2.4	0.5	0.01
Test10	2.0	4.7	1.0	0.001
Test11	1.7	11.1	1.1	0.01
Test12	1.5	360.9	1.2	0.1
Test13	35.4	1147.2	3.9	0.01

quired, so that all the jobs for a given executable do not run for the same duration. In this case, the coefficients of variation for the applications are chosen to be the same as the coefficients of variation for the corresponding executables in the NASA workload. The coefficient of variation for the thirteenth executable is chosen similarly, but taking into account all the remaining jobs in the system. The specifications for all these executables are shown in Table 6.2.

The actual work W required for a given job in Equation 6.1 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. Suppose M is the mean of the distribution, C is the coefficient of variation, and $rand()$ is a function that returns a random number between 0 and 1. To calculate the work for a job with an Erlang distribution, the following formulas [SLTZ77] are used.

$$i = rand(), \quad k = \lceil \frac{1}{C^2} \rceil, \quad T = 1 - (k-1)C^2, \quad a = T + \frac{\sqrt{kT}}{C^2+1}$$

$$W_{Erlang} = \begin{cases} \sum_{i=1}^{k} (\frac{-M}{k-1+a} \log(rand())) & \text{if } i < a \\ \sum_{i=1}^{k+1} (\frac{-M}{k-1+a} \log(rand())) & \text{otherwise} \end{cases}$$

To calculate the work for an exponential distribution, the following equation is used:

$$W_{Exp} = -M \log(rand())$$

For the hyperexponential distribution, these equations are used:

$$i = rand(), \quad a = \frac{1}{2}\left(1 - \sqrt{\frac{C^2-1}{C^2+1}}\right)$$

$$W_{Hyp} = \begin{cases} \frac{-M}{2a} log(rand()) & \text{if } i < a \\ \frac{-M}{2(1-a)} log(rand()) & \text{otherwise} \end{cases}$$

To determine which executable to run next, a random determination is made. The probability of a given executable being the next job submitted is equal to the proportion of the number of jobs for this executable in the NASA Lewis workload. The experiment consists of executing two hundred jobs.

Since the jobs described will complete faster if they have more processors, the distribution of the processors required for the jobs can affect the results. In these experiments, only parallel jobs are used. Thus, jobs can use between 2 and 16 processors. The number actually required for a given job is randomly chosen from a Uniform distribution over the integers two to sixteen.

At this point, the applications that will be used in the tests have been specified in terms of work, number of processors, and execution time functions. However, another factor can affect the performance of the simulated system: the interarrival times of jobs. One option is to submit all two hundred jobs at once, as though a large batch workload were submitted. However, this type of workload is not very realistic if different executables are in the workload. Thus, the approach used for the tests is to have pseudo-random interarrival times between jobs. The interarrival times are chosen from an Exponential distribution with a mean time of 150 seconds. This number was chosen so that over a long enough period, the utilization of the system with efficient scheduling, would approach 75%. Thus, the system will generally have jobs executing, but not have long queues of waiting jobs.

The other factor that can significantly affect the results is the initial state of the Historical Profiler repository. It is desirable to have some data, since otherwise it will require too many jobs to execute before the Historical Profiler has any impact on the scheduling algorithm. As a result, in our experiments, the repository is seeded by twenty-five random executions of each of the thirteen executables.

The other profiler issue is the type of estimates that are used for the tests. As discussed in Chapter 4, the estimates provided by the profiler include error estimates in terms of a confidence interval. In all the tests requiring an estimate of the execution time for a job obtained from the profiler, the estimate will be equal to the greatest value in a 95% confidence interval about the mean. Specifically, it will be equal to the sum of the `estimate` and the `confidenceInterval` values that are returned by a call to `getEstimate()`.

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.

6.2 Results

This section will discuss the results of the experiments in which different schedulers are used to schedule a particular test workload. Table 6.3 classifies the algorithms examined by their preemptability, whether they use filling, and the type of knowledge they use for their predictions. In this table and the rest of the chapter, the variants of EASY that do not kill jobs will be referred to as EASY, while the variant that does kill jobs will be referred to as EASY-kill. This section will examine each of the dimensions of this table. Section 6.2.1 will discuss the relative performance of the non-preemptive algorithms. Section 6.2.2 compares the performance of the algorithms that use filling to those that do not. Following this is Section 6.2.3 which compares the performance

Table 6.3: Classification of Algorithms

	Non-Preemptive		Preemptive	
	Non-Filling	Filling	Non-Filling	Filling
No Knowledge	FCFS	FCFS-fill		
Approx. Knowledge	EASY-kill-pro LEWF-pro	LEWF-fill-pro	EASY-pre-pro LERWF-pro	LERWF-fill-pro
Exact Knowledge	EASY-act LEWF-act	LEWF-fill-act	LERWF-act	LERWF-fill-act

of the non-preemptive algorithms to the corresponding preemptive algorithms. Finally, Section 6.2.4 will compare the performance of the different algorithms using no knowledge, imperfect knowledge, and perfect knowledge.

The results for this chapter are summarized in Table 6.4. 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 total time is the time required for completing all 200 jobs. The utilization, U, is calculated using the set of all jobs in the test, J, where, for a job $i \in J$, p_i is the number of processors used by that job, and t_i is the run time of the job. In the following equation for utilization, T is the total time required for the test and P is the number of processors in the system (in our case, sixteen):

$$U = \frac{\sum_{i \in J} p_i t_i}{TP}$$

6.2.1 The Relative Performance of the Non-Preemptive Schedulers

Table 6.4 contains the result of using different schedulers to schedule the test workload. To judge the relative performance of the algorithms it is worthwhile comparing the performance of the simplest non-filling, non-preemptive versions of the algorithms using perfect service time knowledge, FCFS, EASY-act and LEWF-act. Figure 6.1 shows the

Table 6.4: Performance of Scheduling Algorithms ("pro" means use of the profiler, "act" means use of actual service times)

Algorithm	Kill	Susp./# times	Resp.(s)	Wait (s)	Total 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-kill-pro	24	0	∞[1]	602	29781	58.9
EASY-pre-pro	0	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	0	31/124	908	469	44456	63.7
LERWF-pro	0	25/121	2404	1565	48017	61.5
LERWF-fill-act	0	34/135	855	267	42684	67.3
LERWF-fill-pro	0	38/163	1678	389	45441	65.8

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

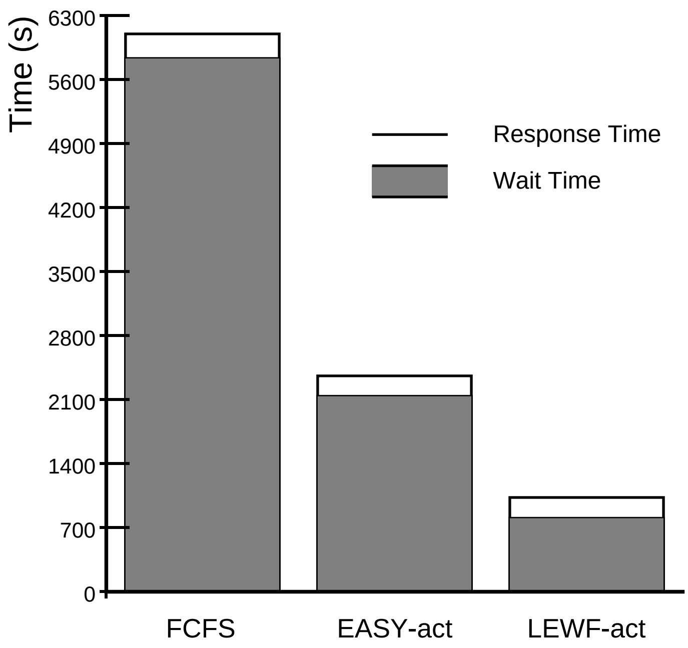

Figure 6.1: The Non-Preemptive Schedulers' Mean Response Times

relative performance of these three algorithms. The wait times in this figure are high relative to the execution times because most jobs have to wait in the pending queue for a relatively long time before they begin executing. If the job interarrival times were longer, this wait time would be shorter.

The FCFS algorithm has the worst response time by far and also has relatively poor total time for the test. This is an expected result, since jobs requiring only a few seconds of run time might wait a long time for a longer job to finish. This can be seen by the relatively large wait time for this algorithm. The total time required for the entire experiment is also high, almost comparable to some of the preemptive scheduling algorithms, despite the fact that the overhead for this algorithm is relatively low. This algorithm is the only algorithm that does not use any application knowledge for scheduling.

The EASY-act algorithm is the FCFS algorithm with backfilling. If there are available processors, and a job can run on those processors without delaying previous jobs, that job will be run. The addition of backfilling greatly improves both the response time and the total time for the entire test. Short jobs have a good chance of being run relatively quickly on the available processors, since, due to their short run time, they are unlikely to delay other jobs.

The other non-preemptive scheduler using accurate estimates, LEWF-act, has the best response time of all the non-preemptive schedulers. This is an expected result, since it means that shorter jobs will not have to wait for longer jobs to finish. This is evident from the relatively low wait time. The total time required for the experiment is reasonable, but not great. However, the throughput of the algorithm could still be improved by attempting to use as many processors as possible, rather than least work first, if the variances in run times within a group of jobs are low. If all the jobs have approximately the same run time, the order in which the jobs are run will not change the mean response time much. If it is possible to pack the jobs to make use of the idle processors, then both the throughput and mean response times will improve. This is done by LEWF-fill.

Thus, as expected, the most basic algorithm, FCFS, was the worst, EASY was better, and LEWF was the worst. This ordering provides a basis for comparing the other variants of these algorithms.

6.2.2 The Value of Filling

Filling is the first addition to the basic algorithms which will be examined. Figure 6.2 compares the mean response times for filling and non-filling algorithms. 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. However, the experiments show that in general, filling improves both the response times and the total time required to process the 200 jobs.

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%. The addition of filling to FCFS 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, and, since more jobs are being run at once, the throughput increases and total time required for the test decreases.

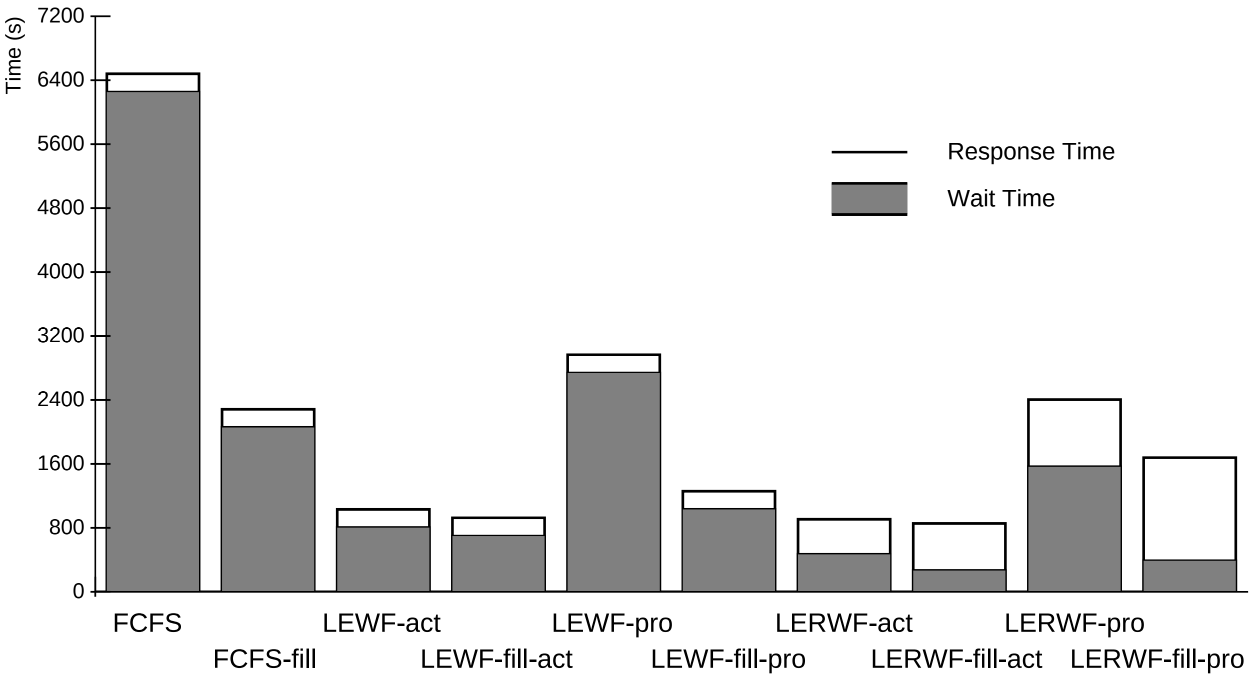

Figure 6.2: The Impact of Filling on Mean Response Times

The effects on the Least Work First algorithms are more interesting. In all cases, filling improves the mean response times and the total times. For LEWF-act and LEWF-fill-act, the mean response times and total times improve by 11% and 6%, respectively. Similar results are evident for LERWF-act and LEWRF-fill-act, where the times improve by 6% and 4%, respectively.

For the four algorithms using the profiler, the results are more extreme. LEWF-fill-pro's mean response time is only 42% of the mean response time of LEWF-pro. The total times improve slightly. The results for LERWF are similar, with the filling version having a mean response time of approximately two-thirds that of the non-filling algorithm. LERWF-fill-pro is particularly noteworthy for its short mean wait time compared to the mean response time. Jobs are started quickly, hence the short mean wait time. But they are also frequently preempted, leading to the large difference in mean response time and the mean wait time.

Thus, filling is relatively more helpful with the algorithms that use the profiler. This result may be because, with the non-filling algorithms, inaccurately long estimates for jobs requiring few processors do not hurt the performance nearly as much in the fill version, since these jobs are likely to be started relatively early regardless of the estimates. To a certain extent in the non-preemptive algorithms, these gains are reduced when a long running job requiring few processors is used to “fill”, and it later delays a short running job requiring many processors. However, it is evident from the results of these experiments that this tradeoff is definitely worthwhile.

In summary, in every case, filling leads to improved response times and reduced total times. It is more important for those algorithms that use less knowledge than it is for those that use more.

6.2.3 The Value of Preemption

Preemption is the second addition to the basic algorithms that will be examined. It is worthwhile distinguishing preemptive schedulers from the non-preemptive schedulers because the preemptive schedulers require the additional complexity of preemption. Thus, although the preemptive algorithms may have better performance, when actually deciding which algorithm to use, it is necessary to consider also the additional implementation effort required for preemption and the additional potential for error because of the complexity.

In this case, 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, LEWF-pro and LERWF-pro, LEWF-fill-act and LERWF-fill-act, and LEWF-fill-pro and LERWF-fill-pro. The relative performances of the final four pairs of algorithms is displayed in Figure 6.3. 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).

A comparison of the performance of the non-preemptive EASY scheduler using the profiler to the performance of a preemptive version of the same scheduler is meaningless. EASY-kill-pro, the version without the preemption, disposes of all the jobs faster than any other algorithm. But this is because twenty four long running jobs are killed by the scheduler when they exceed the estimate. Thus, the average response time is undefined,

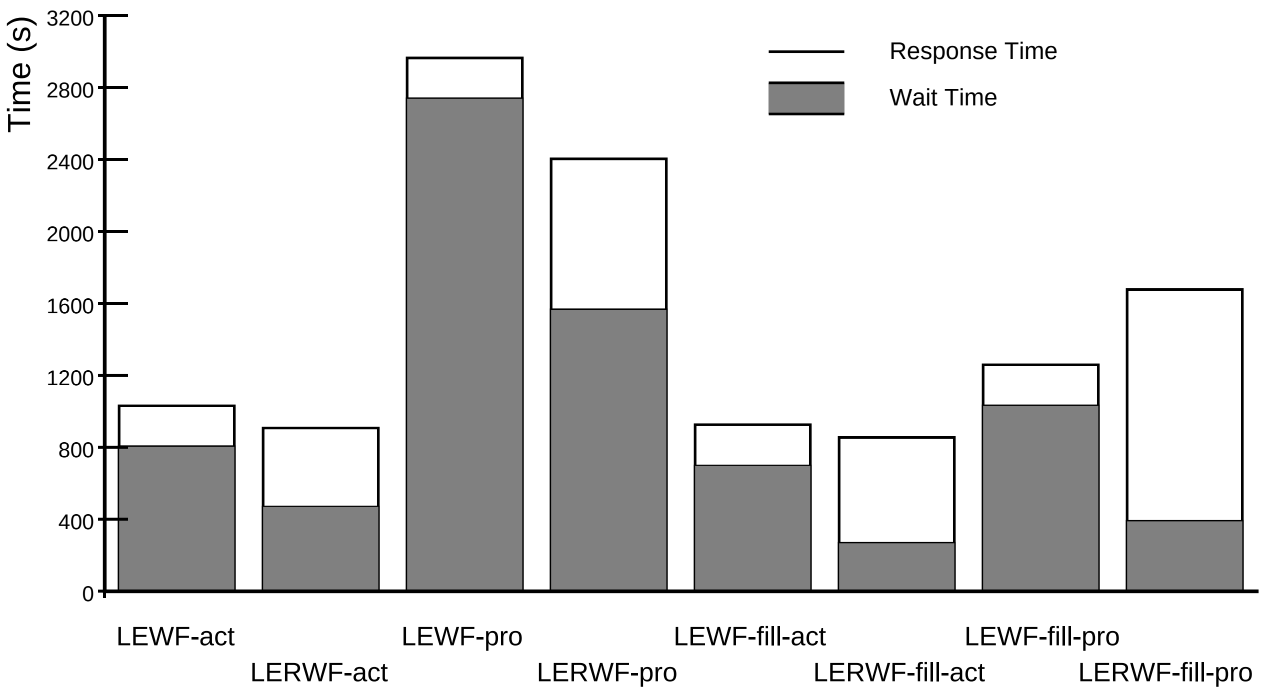

Figure 6.3: The Impact of Preemption on Mean Response Times

since these killed jobs never complete. The same twenty-four jobs are preempted by EASY-pre-pro. Because EASY-pre-pro does not kill jobs due to inaccurate estimates, it is the better algorithm. The percentage of jobs killed, 12%, is noteworthy. This gives some indication of the degree to which the job execution times exceeded the mean. The profiler was requested to return a confidence interval such that there was 95% chance the actual mean duration for jobs of a given executable be within that interval. The highest value in this confidence interval was the estimate of the job duration. Thus, from the large number of jobs killed, we can deduce that a relatively large number of jobs had execution times greater than the estimated mean, even when a high estimate of the mean was used.

The LEWF-act and LERWF-act algorithms, the first and second bars in Figure 6.3, make a more interesting comparison. The non-preemptive algorithm has a mean response time 14% longer than the preemptive version. As is evident from the big 71% difference in the wait times of the algorithms, this improved mean response time is due to 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 too few 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. The total time for the preemptive algorithm to process all the jobs is longer than the total for the non-preemptive algorithm, perhaps due to the overhead of resumption mentioned in Section 5.4 (the description of the LERWF algorithm).

The results for LEWF-pro and LERWF-pro are presented as the next two bars in Figure 6.3. Unlike the previous two algorithms, however, these algorithms use imperfect information when scheduling. As with perfect information, the non-preemptive algorithm has a 23% longer average response time than the preemptive algorithm. The improvement in response times arises from the reduction in wait times for jobs, for the reasons described in the previous paragraph. The total time required to process all the jobs is 23% higher for the preemptive scheduler than for the non-preemptive scheduler, due primarily to the overhead associated with the 121 preemptions and resumptions. Comparing these two algorithms reveals one disadvantage of using the profiler with non-preemptive algorithms: the estimate at first is relatively inaccurate due to the large confidence interval that is used. With preemptive algorithms, poor estimates can be remedied as it becomes apparent that a long job is running, but this is not possible with non-preemptive algorithms.

The relative performance of LEWF-fill-act and LERWF-fill-act helps to confirm that preemption improves the mean response time, but increases the total time required for the experiment. The preemptive version in this case had an 8% improvement in mean response times and was 12% worse for the total time required.

The LEWF-fill-pro and LERWF-fill-pro algorithms offer seemingly contradictory results. In this case, the non-preemptive algorithm has a better average response time by 33%. This is not due to jobs being started later; the average wait time for the preemptive algorithm is less than 38% of the wait time of the non-preemptive algorithm. Rather, this poor response time is due to overhead of preemption and the limitations of the current heuristic for allocating processors (which was discussed in Section 5.7). All the jobs that require few processors are often 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 fact that this algorithm easily has the highest average suspended time for all suspended jobs. It is expected that improved heuristics for assigning processors could reduce the mean response time for LERWF-fill-pro, but identifying such improved heuristics is left to future research.

In summary, preemption generally lowers the mean response times. However, a good heuristic for assigning processors to jobs is required to achieve the maximum benefits for preemption.

6.2.4 The Value of Knowledge

It is difficult to determine improvements in the performance of scheduling algorithms due to the use of the Historical Profiler because the algorithms that use the profiler not only use application knowledge, but *require* application knowledge. Thus, it is impossible to compare the performance of an algorithm not using the profiler to the same algorithm using it. As a result, to compare the effects of knowledge, comparing different algorithms is required. Three different levels of knowledge will be compared: no knowledge, imperfect knowledge, and perfect knowledge. No knowledge will be represented by the variants of FCFS, imperfect knowledge will be represented by the algorithms that use the Historical Profiler (the algorithms with a "-pro" suffix), while perfect knowledge will be represented by the algorithms that use the actual execution times of the applications (the algorithms with a "-act" suffix.) The first two algorithms in Figure 6.4 are the two FCFS algorithms. Every other pair of algorithms in the figure consists of one scheduler using perfect application knowledge and the same scheduler using imperfect knowledge.

Is Knowledge Beneficial?

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

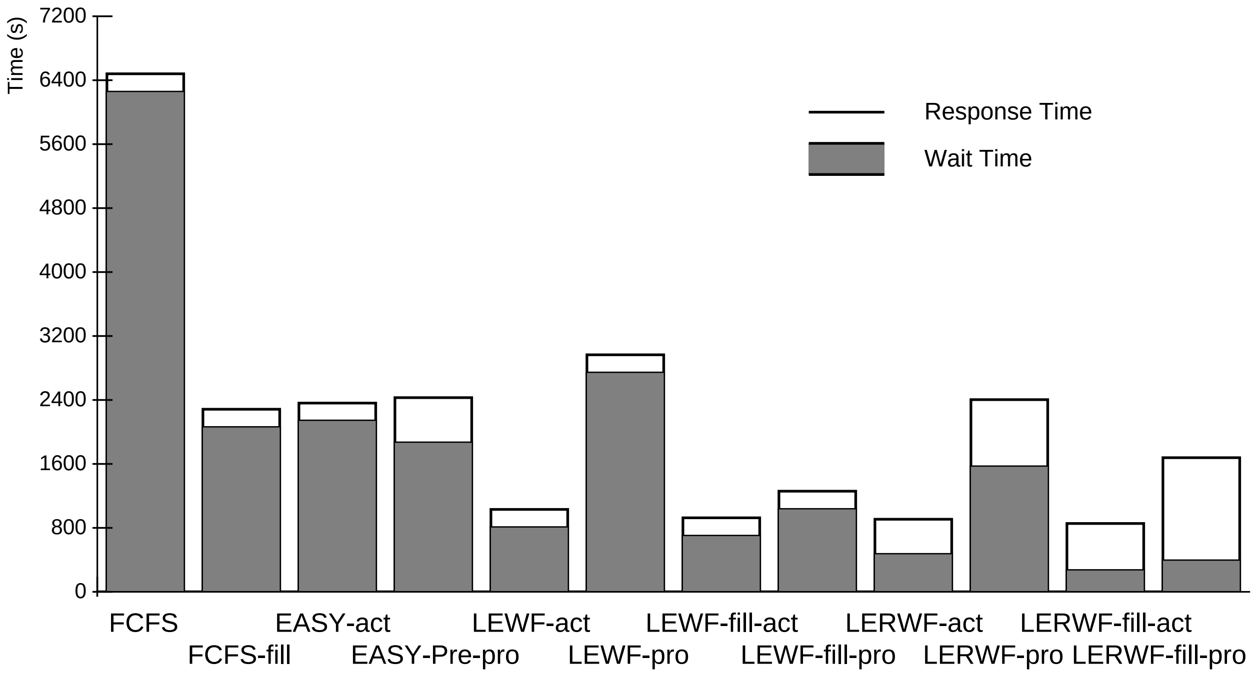

Figure 6.4: The Impact of Knowledge on Mean Response Times

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. The preemptive version of this algorithm, LERWF-pro, improves on this performance slightly, while the algorithms that use perfect knowledge have mean response times less than one sixth of that of FCFS. The performance of FCFS-fill is also poor. Once more, there is almost a 50% improvement in mean response times using LEWF-fill-pro. LRWF-fill-pro, despite the problems in allocating processors described in section 5.7, has a mean response time less than three-quarters that of FCFS-fill. The algorithms that use perfect knowledge again outperform FCFS-fill, this time with mean response times of approximately two-fifths of that of FCFS-fill. Thus, schedulers using the knowledge-based Least Work First strategy outperform by a wide margin comparable FCFS algorithms in all the cases examined.

Comparing the variants of FCFS to EASY-act and EASY-pre-pro does not reveal a significant difference. Both of these EASY variants outperform FCFS, but this is an unfair comparison since both algorithms use backfilling. Thus, it is more appropriate to compare them to FCFS-fill. Both variants of EASY have slightly worse response times than FCFS-fill, less than 10% worse. It would be expected that these three algorithms would have similar performance, since they are very similar disciplines. (For a discussion of the similarities and differences, see Section 5.5.) The reason EASY does not perform better despite its use of application knowledge is because the knowledge is mainly being used to provide predictability to users, not to improve mean response times. The only way the additional knowledge improves the performance is by allowing EASY to backfill jobs without violating the FCFS property of the algorithm. Since the FCFS-fill algorithm can already do similar filling, this does not lead to a difference in performance.

Thus, with the exception of EASY, the use of even imperfect knowledge improves mean response times a great deal over the use of no knowledge.

The Impact of the Accuracy of Knowledge on Non-Preemptive Disciplines

It is now worthwhile examining in more detail the improvements over FCFS that are possible due to the use of knowledge. In particular, it is interesting comparing the improvements over FCFS in mean response time that are attainable using imperfect knowledge to the improvements attainable using perfect knowledge. This provides and indication of what fraction of the potential improvement due to knowledge is attainable using the Historical Profiler. First, we will examine the non-preemptive schedulers.

For the LEWF algorithm, the improvement over FCFS due to imperfect knowledge is large, but still is not close to the possible improvement with perfect knowledge. LEWF-pro has mean response times 46% as long as FCFS, while LEWF-act has mean response times of only 16% as long. Both algorithms outperform FCFS in terms of total time, too, with total times of only 95% for LEWF-act and 89% for LEWF-pro of that of FCFS. The higher throughput of the algorithm using imperfect knowledge is due to the fact that the job length estimates to a large extent depend on the number of processors assigned to a job. Therefore, jobs requiring few processors in general have high estimates, and tend to be run after the jobs requiring many processors. This is particularly true due to the large confidence intervals being used for the estimates, which tend to increase the dependence of the estimate on the number of processors, and reduce the dependence on the executable name. As a result, the jobs requiring many processors, which have to be run by themselves in general anyway, tend to run first. The jobs requiring few processors run last, but since they require so few processors, several jobs can be run at once, leading to very few processors being idle.

With the filling versions of the LEWF algorithm, LEWF-fill-act and LEWF-fill-pro, imperfect knowledge does not hurt the algorithm nearly as much. The mean response time of LEWF-fill-act is approximately 41% of the mean response time of FCFS-fill, while the LEWF-fill-pro is 55% of FCFS-fill. Thus, only a 14% difference relative to FCFS-fill remains to be overcome by improving the accuracy of knowledge. The mean response times of these two algorithms are so close because inaccurately long estimates for jobs requiring few processors do not hurt the mean response time nearly as much in the non-filling versions of the algorithms. Such jobs requiring few processors are likely to be started relatively early regardless of the estimates. In terms of the throughput, both algorithms are very similar. The total time required for both experiments is approximately 4% worse than with FCFS-fill.

Thus, for the non-preemptive schedulers, the accuracy of knowledge is more important for the non-filling algorithms. The filling algorithm using imperfect knowledge has performance only slightly worse than the same algorithm using perfect knowledge.

The Impact of the Accuracy of Knowledge on Preemptive Disciplines

The preemptive LERWF algorithms have similar results to the non-preemptive LEWF algorithms. The non-filling scheduler using the profiler, LERWF-pro, has an average response time equal to 37% of the average response time for FCFS. For the scheduler using perfect information, LERWF-act, the average response time is 14% of that of FCFS. In terms of throughput, LERWF-pro shows a 9% increase over FCFS, while LEWF-act shows a 1% increase. The 23% difference in mean response times is not caused by difficulties in distinguishing between different executables. 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[1] (a result that may not necessarily be true in a real workload). 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. It is expected that this difficulty would be overcome to a large degree if the jobs' memory usage were taken into account (assuming a correlation exists in general between memory usage and execution length).

LERWF-fill-pro requires an even larger percentage improvement to achieve the performance of LERWF-fill-act. In this case, 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. The total times for the tests are 21% and 13% worse than FCFS-fill, respectively. The problem with LERWF-fill-pro, as discussed in Section 5.7, is the heuristic for assigning processors to jobs. Near the end of the test, many preempted jobs require the same processors and thus cannot be run simultaneously. Since the mean response time of LEWF-fill-pro is much lower than that of LERWF-fill-pro, it is also clear that it is possible to improve the response times of this algorithm significantly.

For EASY-act and EASY-pre-pro, there is not much difference between the performance using perfect and imperfect knowledge. The average response time using imperfect information with EASY-pre-pro is only 3% worse than the version using perfect information, even when taking into account the overhead due to preemption. The utilization is almost 5% lower. The preemptive algorithm has a lower average wait time, but more than makes up for it because jobs require longer, on average, to complete. There are three potential causes of this. First, there is additional overhead associated with preempting jobs. Second, preempted jobs require more wall-clock time from when they are first started until they finish. Third, inaccurate estimates have an effect on backfilling. Because high estimates are used (as explained in Section 6.2.3), fewer jobs can be backfilled without risking violating the guarantee that no job will be delayed by any job with

[1]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.

a later submission time. Despite these factors, the performance of the two algorithms is very close. This implies that EASY-pre-pro could be an alternative to EASY at sites where the users want the benefits of EASY, but want to have the option of allowing the scheduler to estimate job run times.

Thus, for the preemptive schedulers, the mean response times using imperfect knowledge are good, but are farther from the ideal than for the non-preemptive schedulers. Overall, the schedulers using the imperfect information of the profiler vastly outperform the FCFS schedulers. For all but the LERWF-fill-pro algorithms, the algorithms using the profiler get 75% of the way to the minimum average response time attainable using perfect information.

6.2.5 Summary

The experiments lead 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.

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.

Chapter 7

Conclusions and Future Work

Analysis of log files obtained from one multiprocessor system and two NOWs showed that classifying jobs by executable, user and degree of parallelism led to coefficients of variation in wall-clock run time much lower than for all the jobs in the entire system. This led to the idea that it is possible to estimate run times for jobs classified in this way much more accurately than would be possible only using information about the entire system.

A database called a Historical Profiler was proposed in Chapter 4 to take advantage of this result. To increase the reliability of the software and decrease the development time, the profiler was built on top of LSF and a Job and System Information Cache. This design choice made it relatively easy to obtain the data required by the profiler and facilitates putting the profiler into practice at a production site. The one disadvantage it had was that it constrained the design to the information and mechanisms provided by LSF.

It was decided that the profiler should have three features that would be useful to a scheduler:

1. A method of obtaining an execution time estimate with an error tolerance.

2. A method of obtaining an approximate execution time function with error ranges.

3. Hypothesis testing that can determine whether, with a particular level of confidence, one job's mean execution time will be greater than another's by a specified amount.

The profiler consists of two parts. The lower part is a repository containing data that is indexed by executable and user, but further classifies jobs by the number of processors used, the memory usage, and the execution time. This design allows the profiler to obtain accurate predictions by classifying jobs according to executable, user, and level of parallelism. It also allows the estimates to become even more accurate if the problem size is inferred from the memory usage and as the execution of the job progresses. If the duration of the job is known to be greater than a particular value, any historical jobs that did not run at longer than the current duration can be ignored when estimating the job's execution time.

The second half of the profiler, above the repository, consists of functions for efficiently manipulating the repository data into a form easily usable by the scheduler. These functions are needed because several complicated statistical techniques are required to transform the raw data to support the three specified features.

In order to test the Historical Profiler, scheduling disciplines were required. Chapter 5 introduced three basic algorithms, FCFS, LEWF, and EASY, with filling and non-filling versions of the first two algorithms. In addition, preemptive variants of LEWF and EASY were proposed. These algorithms were chosen because they were either well-known, or simple, or known to have good performance in certain situations. Much of the functionality required to implement the algorithms was already provided by function calls to LSF or the JSIC. Without this support, the implementation would have been much more difficult. It would have been necessary to build an interface for submitting jobs and examining the status of jobs, hosts and queues. A method of starting, preempting and resuming parallel jobs on a network of workstations would have had to be implemented. Finally, information about the resource usage of all jobs in the system would have had to be obtained for use by the Historical Profiler. With LSF, the log files already store most of the required information.

The most important results of this thesis arose from the experiments discussed in Chapter 6. Of the non-preemptive algorithms, LEWF had the best response time, followed by EASY, while FCFS was worst, and this ordering was evident for most of the variants of these three algorithms. The addition of preemption proved to be beneficial for most of the algorithms that exploited it. For most algorithms, it resulted in smaller mean response times. For the version of EASY that used the profiler, the addition of preemption resulted in jobs not being killed when the profiler returned inaccurate estimates. Filling also proved beneficial; it reduced the mean response times for all the algorithms examined.

The results of further experiments resolved the primary issue of whether the use of application knowledge derived from historical data could be used to improve the performance of a scheduler. The addition of knowledge was beneficial in all cases. Schedulers using even the imperfect knowledge of the profiler significantly outperformed the FCFS algorithms that used no knowledge. In all but one case, the algorithms using the Historical Profiler were able to achieve over 75% of the performance improvements possible by the addition of knowledge. For the remaining algorithm, LERWF-fill-pro, there was still a large improvement over the no knowledge case, even though the performance was not nearly as close to the performance of the comparable algorithm using perfect knowledge as for the other algorithms. Overall, it is clear that the improvements in mean response times due to using the Historical Profiler are substantial. The profiler is effective at predicting the characteristics of jobs.

7.1 Future Research

This work may be extended in a number of ways. The parameters for testing the existing algorithms can be varied, the functionality of the Historical Profiler can be extended, and different scheduling disciplines can be examined. The subsequent three sections will discuss these different approaches to extending the results. First, other experiments using the current algorithms will be discussed in Section 7.1.1. This will be followed in Section 7.1.2 by a discussion of improvements related to the the Historical Profiler. Finally, Section 7.1.3 will propose different scheduling disciplines that could be used for further experimentation.

7.1.1 Varying the Parameters of the Experiments

There are several parameters in the test that could potentially affect the performance of the Historical Profiler and the scheduling disciplines examined. Further experimentation could be done to determine the sensitivity of the results to the parameters used in the experiments. For instance, only one sequence of synthetic jobs was used for all the experiments. Additional experiments using different sequences of pseudo-random numbers to generate the synthetic workload would help increase confidence in the validity of the results.

Another parameter of the experiments was the interarrival times of jobs. The interarrival times were calculated so that the average utilization would be approximately 75%. Since the utilization of real systems vary, it would be worthwhile to varying the interarrival times to see how the relative performances of the algorithms change when the utilization changes. In particular, the performance of the algorithms with jobs having interarrival times of zero would be interesting, since such tests would imitate the effects of a large batch submission or a night queue that only starts processing its jobs after a particular time.

Tests with a more realistic workload could be done to verify the results of this thesis. A parallel workload from a NOW site could be used to "generate" the applications that are to be scheduled. Synthetic applications with all the characteristics of the applications from the log files could be submitted to the scheduler with the same interarrival times as existed in the original system. If a number of NOW sites were used, such experiments would provide strong evidence of the desirability of a given algorithm. A tool for generating a workload from an LSF log file in such a manner would be extremely valuable for both tuning a scheduler for a specific site and for systems software testing and evaluation.

This technique was not used for this research for three reasons. First, because there are few parallel jobs in the NASA Lewis log files that are distributed over a period of several months, the interarrival times are too long to make the test interesting. Second, there is not enough variability in the number of processors allocated to jobs. Finally, there are not enough jobs in the log files to both seed the Historical Profiler and have different jobs for the test.

However, upon more thought, these difficulties can be remedied. The actual interarrival times in the log files can be ignored, and replaced by shorter, pseudo-randomly generated interarrival times as was done for the actual experiments. The lack of variability in the number of processors can be remedied by assuming or calculating an execution time function for each executable. This calculation could be done when there is enough data available for the executable; otherwise, assuming the executable conforms to a set of random values chosen from a reasonable range would be sufficient. Once this execution time function is known, a random number of processors can be assigned to the job, as was done in this thesis. The final difficulty of too few jobs is a more difficult problem. One solution is to continue to seed the database with random jobs, as was done in these experiments, and use the "actual" jobs from the log files only for the jobs that are scheduled.

Alternatively, installing the schedulers at existing production LSF sites could be interesting. It would be difficult to make the same direct comparisons between the mean response times of the algorithms as were done in this thesis, since different jobs with different interarrival times would be scheduled. However, such installations would allow the investigation of many issues. For instance, if the workload did not evolve significantly and the tests were conducted over a relatively long period of time, using different scheduling algorithms would allow general comparisons of mean response times. The number of jobs in the queues at various times of the day could also be examined. In addition, the users' reactions to the use of different schedulers would be worthwhile investigating. Thus, this approach could lead to many interesting results.

7.1.2 Improvements Related to the Historical Profiler

In addition to examining the effects of changing the parameters of the workload, it is also worthwhile investigating the effects of changing the various inputs to the Historical Profiler. First, it would be interesting to determine the sensitivity of the schedulers to the initial job data that the Historical Profiler has available. In the experiments for this thesis, the profiler was seeded with a history of 25 jobs for each executable. The results of experiments without initial data available to the profiler or with more data available for some executables than others could also prove interesting. Such tests are particular important considering that if the profiler were installed at a production site, it would initially have no information about any executables.

A second Historical Profiler parameter worth examining is the confidence interval used in the estimates. In particular, the non-filling algorithms might be improved by using less conservative estimates from the profiler. Currently, the estimate is equal to greatest value in a 95% confidence interval. If instead a much smaller confidence interval were used, say 60%, jobs with high variability would have relatively lower estimates and would be scheduled sooner. Such a change would have a positive effect for short jobs that are instances of executables with highly variable run times, since these jobs would be less likely to wait for longer running jobs. However, it would have a negative effect for long jobs that are instances of executables with highly variable run times since long jobs could run before shorter jobs. Thus, the overall effect is uncertain.

There are several features of the Historical Profiler that could be changed to improve the performance of the profiler and the strength of the results of the experiments. First, because of limitations in the current version of LSF, neither the processor time nor the memory usage were used by the profiler for making estimates. It is likely that the inclusion of this data will increase the accuracy of predictions and reduce the difference in mean response times between the schedulers using perfect and imperfect information. Of course, for such information to affect the results of the experiments, a more complicated workload specification would have to be used that includes both processor time and memory usage. Real applications may have to be used, which would greatly complicate the testing.

A further improvement would be to add methods that do not focus on predicting the mean run time for executables, but instead can say with some confidence that a given job will finish in a certain amount of time. This could be done in the following way. Suppose that there is a given job that is an instance of a particular executable with an observed mean execution time with coefficient of variation greater than one. A method in the Historical Profiler could use a hyperexponential cumulative distribution function with the specified mean and variance to estimate a duration such that the actual duration of the job will be less than this estimate a specified percentage of the time. Such estimates would likely be more useful than the current predictions of the remaining execution time.

To make the functioning of the Historical Profiler consistent, several changes would be required, but the basic design could stay the same. All the job information required is already in the repository, and methods of calculating the mean and coefficient of variation already exist. The execution time function predictions of the Historical Profiler would still be useful if the profiler were modified in this way; estimates of the mean and coefficient of variation would still be required, and dynamic and adaptive schedulers would still require knowledge of the execution time function to determine how many processors to allocate to specific jobs. The main changes would be adding new methods to the profiler interface and adding a way to generate a cumulative distribution function with a specified mean and coefficient of variation. In addition, for consistency, a new type of hypothesis testing, which compares the distribution functions of jobs rather than just mean response times with a confidence tolerance, would have to be implemented.

A further expansion of the Historical Profiler combined with the idea presented in the previous section of generating a workload from log files could lead to a very powerful scheduling combination. The profiler could be used not only to predict the execution times of individual jobs, but also to determine the scheduling algorithm to use at a given site.

Finally, there are several features of the Historical Profiler that were not used to their full potential by any of the tests. Hypothesis testing was not used at all. The approximation of the execution time function was used by the algorithms, since they obtained their point estimates for the execution time from the calculated execution time function. However, the algorithms used did not take full advantage of this feature in the way that a dynamic algorithm could. These algorithms will be discussed in more detail in the next section.

7.1.3 Additional Scheduling Disciplines Worth Examining

This thesis 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; it is worthwhile considering the performance of the profiler with other types of scheduling disciplines.

Many of the performance problems of the LERWF algorithms were attributed to the methods of assigning processors. The problems arose because preempted jobs could not resume if the processors that they required were being used by another job, even if there were a sufficient number of available processors in the system. This problem would be avoided if the scheduling algorithms supported migration since suspended jobs would no longer be required to use specific processors. It is likely that this addition would especially improve the performance of the scheduling algorithms using imperfect information, since these algorithms tended suffer from this problem more than those that used perfect information. Thus, the difference in the mean response times of migratory versions of the LERWF algorithm using perfect and imperfect information would probably be smaller than the difference for the non-migratory versions examined in this thesis.

An examination of dynamic algorithms that use the profiler would be particularly interesting. Currently, the profiler does provide execution time function approximations for executables. As shown by Sevcik [Sev94], knowledge of the actual execution time functions can be used by dynamic algorithms to find the processor allocations that achieve close to optimal average response times. Thus, the approximated execution time function is likely to be useful to a dynamic algorithm. It would be worthwhile comparing the performance of dynamic algorithms that attempt to use the execution time function approximation to minimize mean response time to EQ, a dynamic algorithm that assigns an equal number of processors to each job in the system and that has been shown to have good performance over a large range of workloads [PS95].

However, as mention in Section 4.3.2, actually using the Historical Profiler to obtain these execution time function approximations is complicated by the fact that dynamic jobs are being scheduled. With the static algorithms, it is relatively easy to approximate the execution time function. A number of point estimates of the executable execution times with different numbers of processors can be found. A curve can be fitted to these point estimates to obtain the execution time function, as was done in this thesis. However, with dynamic algorithms, the processor allocations can change. If the allocations change, such point estimates are no longer well-defined. If a job runs on four processors for ten minutes, and then starts running on eleven processors for another ten minutes, it is unclear how to use these times to approximate an execution time function. It does not make sense to derive a single point estimate for a specific number of processors from such a job. Thus, a more complicated method is required for the Historical Profiler to approximate the execution time functions for dynamic jobs. This method is left to future research.

Future work is still required to address many of the issues raised by this thesis. However, the most important issue, the issue on which other results can be built, has been resolved. It has been shown that it is feasible to use a historical profiler to store information about previously run parallel jobs, and that this information can be used to predict the characteristics of future jobs. These predictions can improve the performance of schedulers substantially.

Bibliography

[ACP95] T. Anderson, D. Culler, and D. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, pages 54–64, February 1995.

[ADV+95] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[All78] A.O. Allen. *Probability, Statistics and Queueing Theory with Computer Science Application.* Academic Press, Toronto, 1978.

[AMB76] A.K. Agrawala, J.M. Mohr, and R.M. Bryant. An approach to the workload characterization problem. *Computer*, 9(6):18–32, June 1976.

[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.

[Che88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CHKM93] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Twentieth Annual International Conference on Computer Architecture*, pages 2–13. IEEE Computer Society Press, 1993.

[CKPK90] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmark. In *Proceedings of the 1990 International Conference on Supercomputing. ACM SIGARCH Computer Architecture News*, pages 254–266, September 1990.

[CMV94] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 33–44, May 1994.

[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.

[CS93] M. Calzarossa and G. Serazzi. Workload characterization: a survey. *Proceedings of the IEEE*, 81(8):1136–1150, August 1993.

[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.

[DI89] M.V. Devarakonda and R.K. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, December 1989.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software: Practice and Experience*, 21(8):757–785, August 1991.

[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.

[ELZ86] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[EZL89] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–23, 1989.

[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.

[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.

[HBD96] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–24, May 1996.

[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.

[Kun91] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[LC87] D. B. Leblang and R. P. Chase, Jr. Parallel software configuration management in a network environment. *Software*, 4(6):28–35, November 1987.

[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.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th Internation Conference on Distributed Computer Systems*, 1988.

[LNT85] K.A. Lantz, W.I. Nowicki, and M.M. Theimer. An empirical study of distributed application performance. *IEEE Transactions on Software Engineering*, 11(10):1162–1174, October 1985.

[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.

[ML91] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12:269–284, July 1991.

[MTS90] R. Mirchandaney, D. Towsley, and J.A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 9(4):331–346, August 1990.

[Mut92] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[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.

[RSLS95] M. E. Rosenkrantz, D. J. Schneider, R. Leibensperger, and M. Shore. Requirements of the Cornell Theory Center for resource management and process scheduling. In *Proceedings of IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 192–203, April 1995.

[SCZH96] J. Skovira, W. Chan, H. Zhou, and S. Hotovy. The EASY-LoadLeveler API project. In *Proceedings of IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 23–28, April 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.

[SLTZ77] K.C. Sevcik, A.I. Levy, S.K. Tripathi, and J. Zahorjan. Improving approximations of aggregated queueing network subsystems. In K.M. Chandy and M. Reiser, editors, *Computer Performance*, pages 1–22. North Holland Publishing, 1977.

[TL89] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 15(11):1444–1458, November 1989.

[WOT+95] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[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.

[Zho88] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.

[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.