September 1992

UILU-ENG-92-2234
CRHC-92-18

*Center for Reliable and High-Performance Computing*



# PERFORMANCE EVALUATION AND MODELING TECHNIQUES FOR PARALLEL PROCESSORS

**Robert Tod Dimpsey**



*Coordinated Science Laboratory*
*College of Engineering*
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

# PERFORMANCE EVALUATION AND MODELING TECHNIQUES FOR PARALLEL PROCESSORS

BY

ROBERT TOD DIMPSEY

B.S., University of Illinois, 1986
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

# PERFORMANCE EVALUATION AND MODELING TECHNIQUES FOR PARALLEL PROCESSORS

Robert Tod Dimpsey, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1992
R. Iyer, Advisor

This thesis addresses the issue of application performance under real operational conditions. A technique is introduced which accurately models the behavior of an application in real workloads. The methodology can evaluate the performance of the application as well as predict the effects on performance of certain system design changes. The constructed model is based on measurements obtained during normal machine operation and captures various performance issues including multiprogramming and system overheads, and contentions for resources.

Methodologies to measure multiprogramming overhead (MPO) are introduced and illustrated on an Alliant FX/8, an Alliant FX/80, and the Cedar parallel supercomputer. The measurements collected suggest that multiprogramming and system overheads can significantly impact application performance. The mean MPO incurred by PERFECT benchmarks executing in real workloads on an Alliant FX/80 is found to consume 16% of the processing power. For applications executing on Cedar, between 10 and 60% of the application completion time is attributable to overhead caused by multiprogramming. Measurements also identify a Cedar FORTRAN construct (SDOALL) which is susceptible to performance degradation due to multiprogramming.

Using the MPO measurements, the application performance model discussed above is constructed for computationally bound, parallel jobs executing on an Alliant FX/80. It is shown that the model can predict application completion time under real workloads. This is illustrated with several examples from the Perfect Benchmark suite. It is also shown that the model can predict the performance impact of system design changes. For example, the completion times of applications under a new scheduling policy are predicted. The model-building methodology is then validated with a number of empirical experiments.

# ACKNOWLEDGEMENTS

I thank my advisor, Professor Ravishankar Iyer, whose guidance and persistence made the completion of this thesis possible. I would also like to thank the researchers at the Center for Supercomputing Research and Development for their assistance during the course of this work. In particular, thanks are due to Jay Hoeflinger and Perry Emrath for many useful discussions. Of course, thanks are also due to my parents and family who never doubted that I would complete this work. I thank the boys at 608 and 901, and the girls at 514 for making my many years at the University of Illinois enjoyable. Most importantly, this work would not have been accomplished without the help and encouragement of my friends at the Center for Reliable and High-Performance Computing. I would especially thank Bob Jannens, Paul Chen, John Fu, Paul Ryan, Mike Peercy, Kumar Goswami, In-Hwan Lee, Johnathon Simonson, Nancy Warter, Jeff Baxter, and John Holm.

# TABLE OF CONTENTS

# CHAPTER 1.

# INTRODUCTION

In practice, the performance evaluation of supercomputers is still substantially driven by single-point estimates of metrics (e.g., MFLOPS) obtained by running characteristic benchmarks or workloads. With the rapid increase in the use of time-shared multiprogramming in these systems, such measurements are clearly inadequate, because multiprogramming and system overhead, as well as other degradations in performance due to time varying characteristics of workloads, are not taken into account. In multiprogrammed environments, multiple jobs and users can dramatically increase the amount of system overhead and degrade the performance of the machine. Performance techniques such as benchmarking, which characterize performance on a dedicated machine, ignore this major component of true computer performance.

Due to the complexity of analysis there has been little work done in analyzing, modeling, and predicting the performance of applications in multiprogrammed environments. This is especially true for parallel processors, in which the costs and benefits of multiuser workloads are exacerbated. While some may claim that the issue of multiprogramming is not a viable one in the supercomputer market, experience shows otherwise. Even in recent massively parallel machines, multiprogramming is a key component. It has even been claimed that a partial cause of the demise of the CM2 was the fact that it did not efficiently support time-sharing [1]. In the same paper, Gordon Bell postulates that, "Multi-computers will evolve to multiprocessors" "to support efficient multiprogramming." Therefore, it is clear that parallel processors of the future will be required to offer the user a time-shared environment with reasonable response times for the applications. In this type of environment the most important performance metric is the completion or response time of a given application. However, there are few evaluation efforts addressing this issue.

This thesis addresses the issue of evaluating the performance of applications in real multiprogrammed workloads on a parallel processor. One of the driving forces behind the work is the desire to create evaluation methodologies which rely on real machine measurements. Methodologies based on analytical methods which evaluate parallel processors have been previously proposed [2]-[6]. However, restrictive assumptions on the models weakens the applicability of the results. Real machine and workload measurements are preferred and used in this thesis because they provide more representative models of system behavior.

This thesis introduces a number of performance evaluation methodologies based on real measurements. The methods are illustrated on an Alliant FX/8, an Alliant FX/80, and the Cedar supercomputer but are applicable to most systems available today.

From a high-level point of view the thesis can be broken into two major (related) parts. First, the performance degradation caused by time-shared multiprogramming is investigated. The degradation of performance due to overhead, particularly multiprogramming overhead, is an important issue in the continuing development of parallel processing supercomputers. To understand the feasibility of multiuser, time-shared supercomputers, the degradation caused by multiprogramming in a real environment must be studied.

The second part of the thesis introduces a measurement-based methodology capable of modeling the behavior of an application in real workloads. The methodology allows for the prediction of the completion time distribution of the application. It also allows for the evaluation of system design changes. For instance, the model can predict the completion time of an application executing in real workloads under a new scheduling paradigm.

In general, this thesis is based on the idea that parallel processors are often required to execute more than one application at a given time, and yet, all evaluations ignore this basic fact and assume a dedicated machine. This thesis addresses the following questions: How will my application be affected by other work on the computer? How long will my application take to finish in multiuser

workloads? With these workloads in mind, what changes can be made to the system and my application to improve the performance?

A preview of each chapter and the contributions of the chapter will now be presented.

Chapter 2 summarizes the current state of the art in parallel processor evaluation. The chapter places the work of this thesis in proper context with other work that has been and is currently being done. The chapter also motivates the current work and delineates differences between it and other studies. It is proposed that one of the most important performance indices is application completion time in real workloads and systems should be designed with application performance in mind.

Chapter 3 introduces the experimental environment. The Alliant and Cedar architectures and operating environments are described. Measurement facilities used in this thesis are detailed, and a number of key terms are defined.

Chapter 4 presents two methodologies which quantify the lower bound on overhead attributable to multiprogramming overhead. The techniques are illustrated on the Alliant FX/8 and the Cedar supercomputer. The effect of multiprogramming overhead has on the performance of parallel applications is determined. There are two major contributions in the chapter. First, the methodologies are the only formal methods known to quantify the lower bound on multiprogramming overhead. Second, the measures obtained on the machines under investigation are intersting and may be useful for simulation or analytical models in the future.

Chapter 5 introduces a methodology whereby multiprogramming overhead incurred in real workloads can be measured. The methodology is illustrated through an extensive study of overheads found in real workloads of an Alliant FX/80. In addition, statistical analysis is performed to relate the measured overheads with the workload characteristics. It is the only study which the author knows of which isolates and quantifies the cost of time-shared, interactive multiprogramming in real workloads on a multiprocessor system.

Chapter 6 conducts a multiprogramming overhead analysis similar to that presented in Chapter 5 on the Cedar supercomputer. Because Cedar is a prototype machine and not heavily utilized, a number of real synthetic workloads are created. The multiprogramming and system overheads incurred in these workloads are determined using the methods of Chapters 4 and 5. It is found that multiprogramming overhead on the Cedar is sometimes extremely high. Further analysis presented in the chapter determines that overhead caused by synchronization of loops spread across all processors of Cedar is the cause of the degraded performance in multiprogrammed workloads. In general, the results argue for gang scheduling on parallel processors when fine grain parallelism is being exploited.

Chapter 7 outlines a measurement-based model-building methodology capable of predicting the completion time distribution of an application executing in real workloads. The methodology models the behavior of a given class of applications executing in real workloads on a given machine and is able to predict accurately the effect on performance of system design changes. For instance, the model can predict performance effects of the addition/subtraction of processors, changes in scheduling, and reduction of overheads. The model is constructed using both Markov and statistical clustering analysis. The methodology is illustrated by modeling the performance of computationally-bound, parallel applications in real workloads on the Alliant FX/80.

Chapter 8 uses the model constructed in Chapter 7 to conduct a thorough performance evaluation of different processor configurations and scheduling paradigms for the Alliant FX/80. The performance effects of adding more processors to the system and reducing multiprogramming overhead are also investigated. A number of scheduling paradigms and processor configurations which improve the performance of parallel applications are identified. It is one of the few parallel processing scheduling evaluations that is based on real workload measurements. The chapter also illustrates both the power and the flexibility of the model.

Chapter 9 presents the results of empirical validation experiments. The accuracy of the model is shown by predicting the performance of a new scheduling paradigm, and then collecting real workload

measurements to show that the prediction was correct. A similar experiment is conducted on a machine with a processor removed. The chapter presents four empirically based validations of the method and the model. Finally, in Chapter 10, a summary of the work is given with suggestions for future work.

# CHAPTER 2.

# RELATED WORK AND MOTIVATION

Performance evaluation techniques for uniprocessors have evolved over the years. However, these techniques have not been able to answer fully the performance questions posed by parallel processing systems. While some techniques such as benchmarking may remain useful if certain modifications are made, other techniques may have to be abandoned altogether when evaluating parallel processors. The prodigious amount of research done in developing evaluation techniques for parallel processors has propelled the development of these machines, but many questions remain unanswered. In this section, a general overview of evaluation techniques for parallel processors is provided. The purpose of this chapter is to place the work presented in this thesis in the context of other current and past work.

The tight link between the architecture of a parallel machine and the performance of a parallel application on that machine makes the evaluation of application performance extremely important for parallel systems. It has been argued that parallel processor performance evaluation should be driven by application performance [7]. In other words, parallel processors should be judged by how quickly they execute real applications and should be designed with application performance as the major goal [8],[9]. The work summarized in this chapter, as well as the work presented in this thesis, pertains to application performance, how overheads and interactions of real workloads affect this performance, and how models can be constructed which allow design decisions to be based on application performance.

Because they are the most common applications used for performance evaluation, this chapter will first review common benchmarks. The additional requirements placed on a parallel benchmark (as compared to a uniprocessor benchmark) will be highlighted in the discussion. The chapter will then review current work which evaluates and tunes applications for execution on parallel processors

in dedicated environments. Following this, it will be argued that due to complex overheads and multi-job interactions, it is crucial on a parallel processor to consider also the performance of the application in multiuser workloads. Work which evaluates the overheads of multiuser machines and which investigates application performance in these usage environments will then be reviewed.

The work presented in this thesis adds to or is different from all of the work which will be summarized in this chapter in a number of important ways. First, no other study distinguishes, quantifies, and analyzes the difference between total system and multiprogramming overhead. Second, no other study investigates the effect of these overheads on application performance in real workloads. Third, the question of determining the response time distribution of an application executing under real workloads has not been addressed before. Fourth, the model-building technique presented in this thesis which accounts for application performance while simultaneously giving feedback to make credible design decisions is completely original. Finally, few of the mentioned studies use real measured data to build models, and even fewer of them use real empirical data to validate the predictions of the models.

### 2.1. Benchmarks

Benchmarking has long been the most common and consequently the most controversial form of computer performance evaluation. The simple task of running a *representative* application or section of code on different architectures to compare them is intuitively pleasing, but can be misleading. The foibles and pitfalls of straightforward benchmarking are well-documented [8],[10]-[12]. First, the true representativeness of the benchmarks is always in question. Second, benchmarks (even suites of benchmarks) tend to boil performance down to a single magical (and maybe meaningless) number [13]. Third, benchmarks often quantify only processor speed and do not stress memory hierarchies or I/O. Fourth, differing levels of benchmark optimization (by the user or by the compiler) make it difficult to compare different systems. Does a benchmark perform better on one system because of hand-optimization, the compiler optimization, or is the machine actually faster? Fifth, benchmarks are

normally executed and measured in a dedicated environment and performance loss due to multiple job interactions is not quantified. Finally, and maybe most importantly, benchmarking provides little insight into the reasons behind a system's achieved performance and hence is of little value from the perspective of system design.

The above problems aside, when used correctly, benchmarks play a prominent positive role in computer evaluation. In addition to providing a platform to compare machines, they can be used as workloads to compare the effects of system design enhancements [14]. The most common benchmark suites include the Whetstone benchmark [15], the Linpack codes [16], the Drystone benchmark [17], the Livermore FORTRAN Kernels [18], and the SPEC benchmarks [19],[20]. Of these, the SPEC benchmarks are probably the most influential today. The SPEC benchmarks represent the current philosophy that real, large applications are necessary to truly test a machine. The ten SPEC benchmarks were chosen so they could not be easily "optimized" away, had verifiable output, stressed the cache and memory systems, and had long enough run times to minimize timing variances [20].

This philosophy of using large, real applications to quantify a machine's performance has carried over into the benchmarking of large, parallel super and mini-super computers. The most significant differences between uni- and multiprocessor systems in terms of benchmarking is the sheer increase in magnitude and complexity of the parallel system. For instance, memory hierarchies are normally magnitudes larger on parallel processors, requiring benchmarks to be that much more extensive to test the limits. The added intricacies of parallel machines (cache coherency, synchronization, communication, network contention) link the application and architecture close together. This makes the quantification of performance into a single number for parallel machines impossible. The performance achieved is as much a result of matching the algorithm to the architecture as the "speed" of the machine.

Currently the most influential suite of benchmarks for large, parallel machines is the PERFECT Club benchmarks [21]-[23]. The PERFECT club consists of 13 large, parallel, computationally inten-

sive scientific applications (e.g., molecular simulation, seismic migration analysis). The codes represent present-day, large-scale scientific computing. The PERFECT club has been used to evaluate machines such as the Alliant FX/8 and FX/80, CRAY Y-MP and X-MP, iPSC/1, NCUBE, NEC SX/2 and SX-3, IBM RS6000, IBM 3090-600S, and the Encore Multimax. A number of PERFECT club benchmarks are used in this thesis as test applications.

To evaluate a machine, the PERFECT club applications are ported and timed. Application optimizations may then be performed; however, the type of optimization and amount of time spent optimizing are recorded for future analysis. A goal of the PERFECT effort is to delineate a relationship among applications, architectures, and useful optimizations. The major complaints against the PERFECT club codes have been the extensive effort that is needed to port the codes and the absence of significant memory and I/O stressing in the applications. These complaints are being addressed in the creation of PERFECT 2.

Other recent efforts in parallel computing benchmarking are the EuroBen benchmarks [24], the SPLASH benchmarks [8], and the Genesis project [25]. The goal of the EuroBen suite is to create a benchmark with a hierarchical structure. There are four levels of the benchmark corresponding to levels of complexity. The first level consists of basic functions (e.g., assignments, broadcasts); the second level contains common numerical algorithms (e.g., matrix-vector multiply); the third level tests the system's I/O; the last level contains applications on the PERFECT benchmark scale. There are plans for interactive, multiuser benchmarks. The SPLASH codes attempt to provide a set of applications that are not "toys" which can be used to guide the design of parallel systems.

The work presented in this thesis has different goals than those of benchmarking. It is concerned with methods which make intelligent system design decisions. Instead of using benchmarks as test cases, real uncontrolled workloads are measured and models are created which aid in the system design decisions. While, the measures used by the technique are not repeatable like benchmarks, they provide a more realistic representation of workloads on the system in question.

### 2.2. Application Modeling and Performance

Benchmarking is generally most useful when comparing the performance of machines [26]-[29]. However, it does not provide insight into how an application from a given domain will execute on a machine nor provide much feedback as to how a certain design modification will affect application performance. However, these tasks are crucial for both application and system design. This section will summarize research that attempts to predict or quantify the performance of applications executing on parallel machines.

One common method used to predict application performance is to break the application into sections for which performance is known. Koss proposed using measured, representative kernels of code to predict the performance of a full application [12]. The execution times of the kernels have been previously measured, and the application in question is divided into parts which are similar to the kernels. A more detailed attempt at characterizing individual code sections and then using these to predict the performance of an application was attempted by Saavedra-Barrera [30]. In his work, a machine analyzer is introduced which measures the execution of FORTRAN constructs on a machine. In addition, a program analyzer is used to deconstruct the primitive structure of the application. Combining results from these two allows the completion time of an application executing on a dedicated machine to be predicted. The methodology has been successful for applications which do not stress the memory hierarchy and perform limited I/O.

A methodology to characterize the behavior of shared-memory hierarchies for multiprocessors was introduced by Gallivan [31]. The technique consists of a family of parameterized kernels which provide empirical results usable in understanding the performance of applications. Understanding application performance through vector performance has been attempted by Fatoohi [32],[33]. Vector performance on a NEC-SX2, Cray-2, Cray Y-MP, and ETA 10-Q was investigated. Gustafon has methods to gauge the performance of certain applications on massively parallel hypercubes [34]. Bodin et al. investigated algorithm performance on a BBN GP1000 [35]. Bradley et al. characterized

the public workload of a CRAY X-MP/48 by the requirements of its major programs [36]. Finally, Gallivan et al. investigated methods to predict the performance of parallel numerical applications [37].

All of the work mentioned above is concerned with the prediction and understanding of application performance on a dedicated machine. However, such studies ignore the multiprogramming and system overheads caused by the interactions and time-varying characteristics of real workloads. As will be seen in this thesis, these effects can often be substantial. In the next section, the need to quantify the overheads caused by real multiuser workloads and understand how these affect application performance will be discussed. It is a fundamental tenant of this thesis that application performance in multiuser workloads must be evaluated in addition to that in dedicated environments. Also, system design changes should be based on information garnered from real multiuser workloads.

### 2.3. Overheads and Complexities of Multiuser Environments

Performance evaluation is normally conducted in a single-user dedicated environment. However, the vast majority of machines, even parallel machines, execute in multiuser environments. While evaluating the performance of an application on a dedicated machine will definitely provide insight into its behavior in a multiuser environment, multiple users and processes interact in complex ways while contending for shared resources that cannot be understood in a single user setting. In addition, multiprogramming and systems overheads affect the performance of individual applications differently [38],[39]. Therefore, it is beneficial to evaluate or predict the performance of an application within the setting of a multiuser workload in addition to evaluating it on a dedicated machine.

The complexities introduced when moving from a single-user environment to a multiuser one are much greater for a parallel processor than for a uniprocessor. Multiple users on a uniprocessor introduce cache flushes, contention for main memory, and I/O overlapping. Multiple users on a parallel processor often exacerbate these intricacies. In addition, problems such as preemption affecting synchronization and migrating processes have undesirable cache effects [40]-[43].

However, little work has been done in quantifying the costs and effects of multiuser workloads. Williams quantified the percentage of processing power consumed by operating system overhead for a CRAY supercomputer under real workloads [39]. She postulated that an efficient operating system would use less than 10% of the available processing power. In other publications, it has been stated that the operating system may use more than 25% of the processing cycles [44]. Obviously, system overhead is highly machine-dependent, and a variety of real machines should be studied before any general conclusions can be drawn.

This thesis argues that, due to complex interactions among jobs, more emphasis should be placed on the evaluation of application performance in multiuser environments and that limiting evaluation to dedicated machines may be detrimental.

This argument can be substantiated with a slightly contrived example of application performance on the Cedar supercomputer [45]-[47]. Cedar is an experimental supercomputer with a hierarchical shared memory organization which will be described in detail in Section 3.1.2. For the purpose of this example, it is important to know that Cedar is made up of clusters. Each cluster is a modified 8-processor Alliant FX/80 [48] with a local cluster memory. A modified Omega network is used to connect each cluster to a large shared global memory. The prototype Cedar machine is a 4-cluster, 32-processor machine.

Using Cedar FORTRAN constructs, the PERFECT club benchmark TFS [22] was ported to the Cedar supercomputer. On a dedicated Cedar, using all 32 processors, the application completes in 75 s. The same application was also written to run on a single cluster — eight processors with all data residing in cluster memory. This version, called TFS_SC, completes in 110 s on a dedicated machine.

Three separate multiple application experiments were then conducted to simulate multiuser environments. In the first experiment, multiple copies of TFS (written for Cedar) were executed. In the second experiment, multiple copies of TFS_SC were executed on the same cluster. In the third experiment, multiple copies of TFS_SC were executed on different clusters. Figure 2.1 shows the

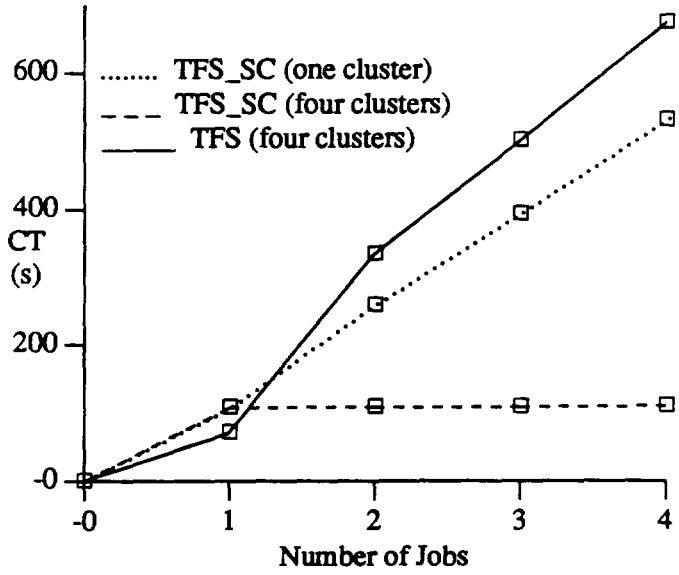



Figure 2.1 Effect of Multijob Environments (Cedar)

completion time of the each version of the application as a function of the number of applications running.

If one application is run (as stated early), the full cluster implementation is faster. However, when more copies are run, the single cluster version (TFS_SC) begins to outperform the full machine. This is true even for the experiment in which all copies of TFS_SC are run on the same cluster. The case in which the applications are run on separate clusters shows an even larger improvement; four copies of TFS multiprogrammed on 32 processors (four clusters) take six times longer than four copies of TFS_SC each on its own cluster.

Now there are problems with the example but the message is clear: In a multiprocessor machine, ignoring the effects of multiuser workloads can lead to erroneous evaluation. The effects of multiuser environments are significant on an individual application's performance and these effects are not uniform across applications.

Up until this point, multiuser environments in parallel processors have been addressed mainly from a scheduling point of view [49]. For instance, under *realistic* workloads, Leutenegger found through simulation that "policies that allocate an equal fraction of the processing power to each job in

the system perform better, on the whole, than policies that allocate processing power unequally" [50]. Zahorjan has found (through simulation) that dynamic scheduling policies often outperform their static counterparts [51]. Gupta determined that the type of synchronization used within an application has tremendous performance impact depending on the workload [43]. Innovative work in this area has also been done by Polychronopolous [52], Ousterhoust [53], and Majumdar [54]. All of these studies have used either simulation or analytical techniques with *realistic* values for overheads caused by context switching, scheduling, and general multiprogramming overhead. However, research shows that overheads are workload-dependent and cannot be characterized by a single value [38].

This thesis presents work in this area using measurements from real machines. First, a comprehensive study of overheads due to interactions caused by multiple users on parallel processors is presented. Techniques are introduced which quantify these overheads; they are illustrated on the Alliant FX/8, Alliant FX/80, and Cedar supercomputers. Results from real workloads on these machines are also presented. Statistical analysis is then done to relate these overheads to workload characteristics.

Following the overhead measurements, the thesis introduces a methodology for modeling the behavior of a given domain of applications executing in real workloads on a particular machine. The model is constructed from real measured data obtained during normal machine operation. A key component of the model quantifies the overhead caused by multiple job interaction quantified in the first part of the thesis. The model is capable of predicting the distribution of completion times in real workloads for a given application. The predictions are useful in gauging how quickly an application will execute, or in predicting the performance impact of system changes such as scheduling modifications. The methodology is illustrated by modeling the execution of computationally bound, parallel applications running in real workloads on an Alliant FX/80 and a Cedar supercomputer. The model is used to evaluate the effect on application performance of a variety of system design changes including: processor reconfiguration, scheduling paradigm modifications, and overhead reductions.

The work in this thesis attempts to address a number of the performance evaluation problems that have been referred to in the previous paragraphs. Multiuser, multijob environments are addressed, real workloads are measured, and feedback is provided to assist in system design changes.

# CHAPTER 3.

# EXPERIMENTAL ENVIRONMENT AND DEFINITIONS

This chapter details the three major components which make up the experimental environment of this study: 1) the measured machines, 2) the measurement facilities, and 3) the target applications. In addition, necessary vocabulary is introduced. The first section of this chapter will summarize the architecture and the operating environments of the machines measured— the Alliant FX/8, Alliant FX/80, and Cedar. Emphasis is placed on the scheduling paradigms and concurrency strategies for these machines. The next section details the measurement facilities used to measure the real workloads. The methodologies presented in this thesis require real applications with known resource requirements and characteristics to be executed and monitored in real workloads. The third section of this chapter will describe these applications which will be referred to as *target applications*. Finally, the last section will define a number of terms that are used throughout the thesis.

This chapter should be viewed as a reference summary to be referred to throughout the reading of the thesis. The three components described are tools which make the research presented possible. The methodologies presented in the following chapters can be used in conjunction with other systems, measurement facilities, and target applications.

## 3.1. Computer Systems Monitored

### 3.1.1. Alliant FX/8 and FX/80

The Alliant FX/8 and FX/80 are shared memory, multiprocessor mini-supercomputers (Figure 3.1) [48]. They can best be understood as two groups of processors: Computational Elements (CEs) and Interactive Processors (IPs). There are eight CEs and up to twelve IPs on any given Alliant. The Alliant FX/8 used in this study has three IPs while the FX/80 is equipped with six. The FX/80 had 96 M of main memory and a 512 K cache. The FX/8 had a main memory of 32 M and a cache of 128 K.

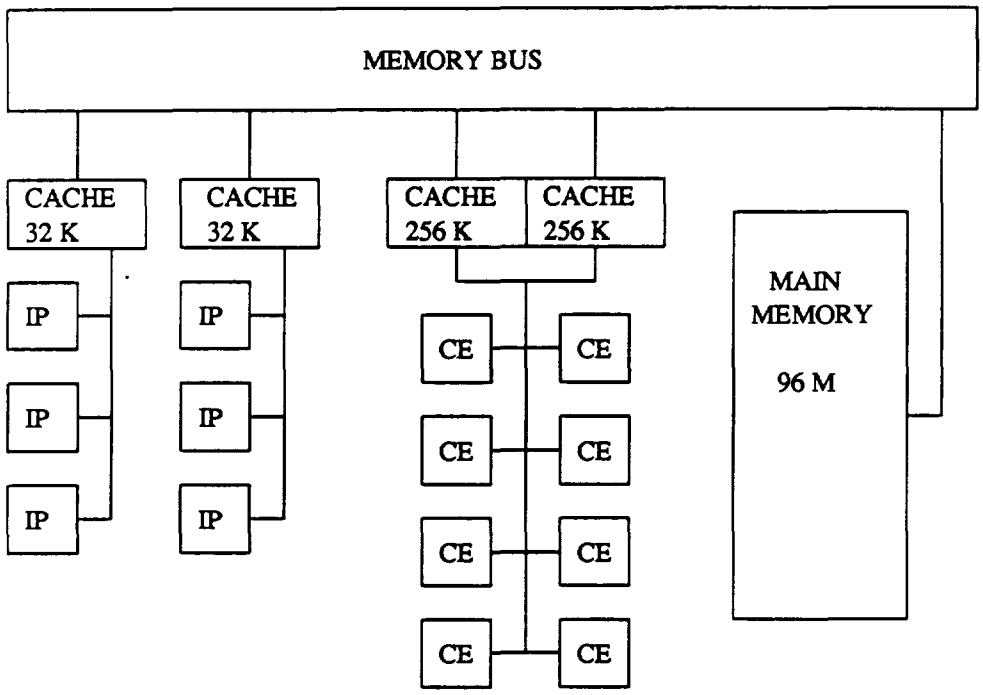



Figure 3.1 The Alliant FX/80 Architecture

The FX/80 also has faster processors than the FX/8. Other than this, the basic architecture, configuration, and scheduler on the two machines studied are basically the same.

The Alliants measured for this study were located at the Center for Supercomputing Research and Development (CSRD) at the University of Illinois in Urbana. The machines were used as testbeds for Cedar applications and operating system functions. The workloads consisted largely of algorithm development work and general scientific computing.

The operating system on the measured Alliants, called Xylem, was designed at CSRD for the Cedar supercomputer. Xylem is a modified version of Concentrix, Alliant's Unix-based operating system. Xylem allows applications written with Cedar FORTRAN constructs to be executed on a single Alliant by emulating the shared global memory of Cedar. In one sense, the Alliants measured are actually single-cluster Cedar supercomputers.

The CE complex executes all parallel applications, as well as most serial user programs. For this reason it will be the focus of this study. The IPs handle interactive jobs, I/O, and a large portion of the

operating system. Their purpose is to give fast response times to interactive jobs and to offload system work from the CEs.

The CE complex operates in one of three modes: *traditional complex mode*, *static detached mode*, or *dynamic mode*. In the traditional complex mode, all CEs are simultaneously applied (gang scheduled) to the execution of a single, parallel application (i.e., the CE complex is a single resource which multiprograms parallel jobs). In the static detached mode, a fixed number of the CEs are *clustered* and the rest of the CEs are *detached* (Figure 3.2). The clustered CEs multiprogram parallel applications, while each individual detached CE concurrently executes an independent serial job. In the dynamic mode, all of the CEs (as a group) dynamically switch between the two configurations, detached and clustered (Figure 3.3). In the detached configuration, the eight CEs are used as independent processors and serial jobs are multiprocessed on the individual CEs. In the clustered configuration, all eight CEs are gang scheduled to execute single, parallel applications.

The mode in which the CEs are configured is an operating system option set at boot time. The machine monitored for this study had the CEs set up in the dynamic mode at all times. Therefore, the eight CEs dynamically switched between detached and clustered configurations.

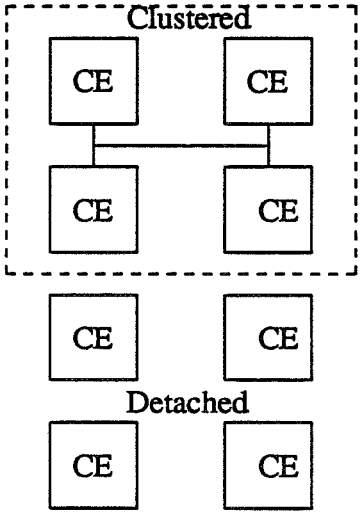



Figure 3.2 Static Detached Mode

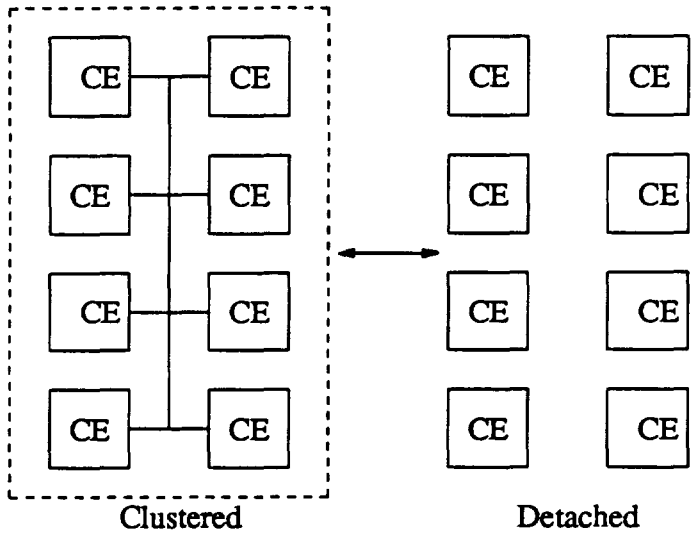



Figure 3.3 Dynamic Detached Mode

The scheduling paradigm used by the Alliant is based on job classes. The scheduler on the Alliants measured in this study is based on six classes of jobs. The class of a job specifies the resource needed to process the job (single CE, clustered CEs, or IP) and the priority of the job. Parallel jobs fall into two job classes: *type A* and *type C* cluster jobs. Type C jobs are those which have been compiled to execute on the Cedar supercomputer (they may use global memory or some Cedar FORTRAN constructs). Type A jobs are normal Alliant applications. To execute a cluster job (either type) eight CEs in the clustered configuration are required. There are four classes of serial jobs. Two of these require a single CE to execute and are referred to as CE jobs, *type A* and *C* (CE (A) and CE (C)). The remaining two serial job types are referred to as *IP* and *IP/CE* jobs. An IP job require an IP to execute, while an IP/CE job can be processed by either an IP or a single CE.

Table 3.1 summarizes the scheduling algorithm for the CEs of the Alliant FX/8 and FX/80. The scheduler steps down the levels of the table granting the specified time quantum to a job of the choice 1 job type. If there is no choice 1 job in the system, a choice 2 job is scheduled. If a choice 2 job is unavailable, a choice 3 job is scheduled, and so on. All jobs within a class are scheduled in a fair round-robin fashion. When it is time for a cluster job to execute (levels 1 and 2), the CEs become

physically clustered. The CEs are in the detached configuration at levels 3, 4, and 5 (if IP/CE or CE jobs are available). A similar algorithm is used to schedule the IPs.

In combination with the above algorithm, the Alliant provides an interesting scheduling optimization. If there is only one job requiring a detached CE (IP/CE, CE (A), or CE (C)) in the system, then the CEs do not detach at levels 3, 4, and 5. Instead, they remain clustered and execute the job on a single CE, effectively avoiding the overhead of switching CE configurations

### 3.1.2. Cedar supercomputer

The goal of Cedar supercomputer project is to demonstrate that a hierarchical shared memory machine can provide supercomputer performance across a wide range of applications [45]-[47]. The Cedar supercomputer consists of multiple clusters connected to a large global memory across a network (Figure 3.4).[1] Each cluster is a modified Alliant FX/8 with a local memory of 64 M. Cedar currently consists of four clusters and a global memory of 64 M.

Parallel applications written for Cedar are made up of tasks [55]. Each task executes on a cluster and uses all eight CEs (in the clustered configuration) of that cluster. An application can be split into as many tasks as needed; however, normally four are chosen because there are four clusters on the current Cedar machine. The tasks of an application are distributed to the clusters by a program called the Xylem server. Once the tasks of a parallel job are distributed to the appropriate clusters, they

Table 3.1
FX/8 and FX/80 Scheduling Algorithm (Policy A)

| Level | Quantum | Choice 1 | Choice 2 | Choice 3 | Choice 4 | Choice 5 |
|---|---|---|---|---|---|---|
| 1 | 300 ms | cluster (A) | cluster (C) | IP/CE | CE (A) | CE (C) |
| 2 | 400 ms | cluster (C) | cluster (A) | IP/CE | CE (C) | CE (A) |
| 3 | 200 ms | CE (C) | CE (A) | IP/CE | cluster (C) | cluster (A) |
| 4 | 200 ms | CE (A) | CE (C) | IP/CE | cluster (A) | cluster (C) |
| 5 | 200 ms | IP/CE | CE (C) | CE (A) | cluster (C) | cluster (A) |

[1]Notice that the term *cluster* refers to a group of processors (an Alliant) of Cedar, while, simultaneously, the term *clustered* refers to a configuration in which the CEs on an Alliant may exist. The author apologizes for the multiple meanings of the term.

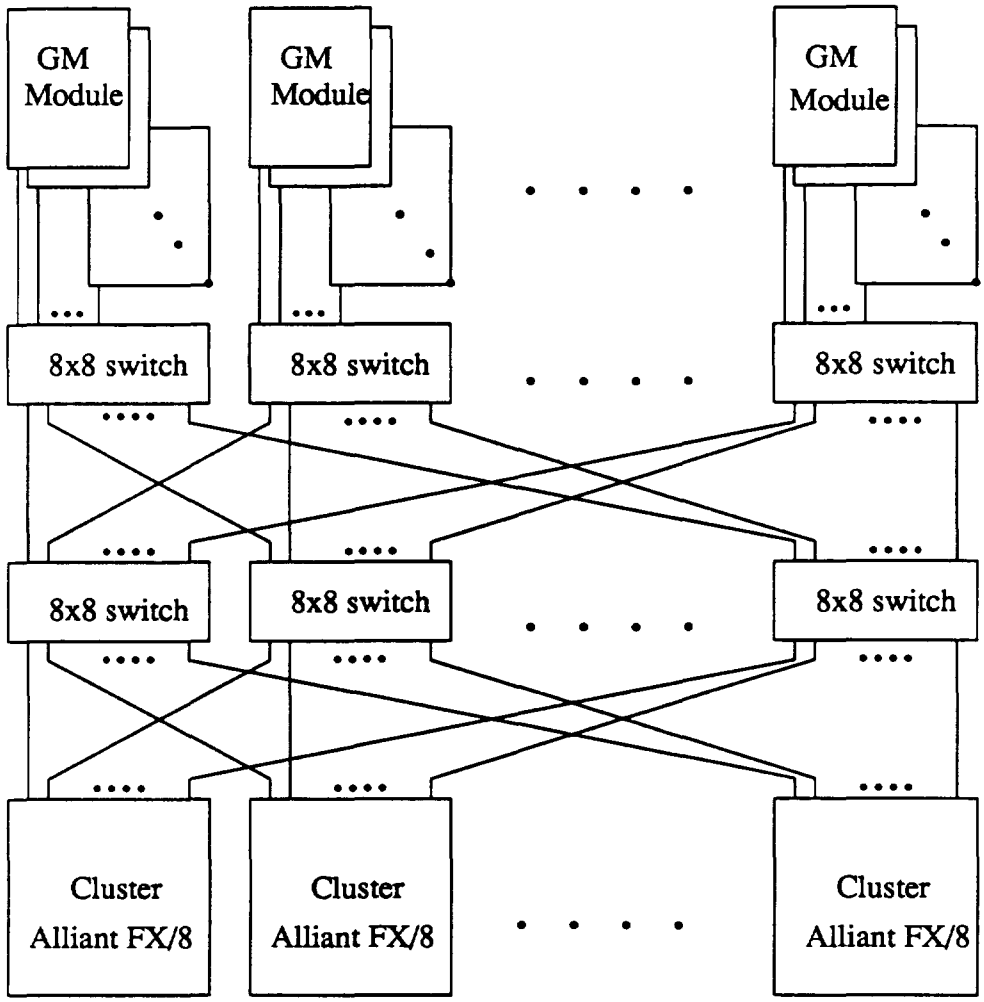



Figure 3.4 The Cedar Supercomputer

remain there and are scheduled using the Alliant scheduler. Communications among the tasks is done through the global memory and through synchronization variables.

Similarly to the Alliants measured, the clusters of the Cedar supercomputer have their CEs set up in the dynamic mode. This means that they dynamically switch between being clustered and detached. There are four classes of jobs which are scheduled on each cluster (individual Alliant): cluster, CE, IP, and IP/CE jobs. Cluster jobs are parallel jobs which require all eight CEs in the clustered configuration. They are the individual tasks of a Cedar application (generated from the Cedar compilers), or they are Alliant parallel jobs (generated from the Alliant compiler). The other three job

types are serial application. Similarly to the Alliant, IP jobs execute on a single IP, CE jobs execute on a single CE, and IP/CE jobs can execute on either an IP or CE.

The scheduler on each cluster is summarized in Table 3.2. The table is read in a manner identical to that of Table 3.1. Note that unlike the scheduler on the Alliants, the distinction of whether a job has been compiled with Cedar constructs (type C) or with the regular Alliant compiler (type A) does not matter when scheduling. Both job types are grouped into the same class and scheduled in a fair round-robin fashion.

Cedar FORTRAN provides two distinct forms of concurrency which exploit the processors of all clusters on the Cedar supercomputer. The first will be referred to as task concurrency, and the second will be called loop concurrency.

Task concurrency is a coarse-grained parallelism in which the application forks off independent tasks which run separately on the different clusters. The tasks themselves may then exploit parallelism at the loop level similarly to a parallel job executing on a single Alliant. To busy all 32 processors requires 4 tasks each executing a loop on the eight processors of each cluster. The tasks share address space for communication purposes, but all synchronization is specifically handled by the user. The Xylem server schedules the forked tasks to clusters, and, once assigned to a cluster, it remains there and the individual cluster handles scheduling as shown in Table 3.2.

Loop concurrency is a finer grain of parallelism. It allows iterations of loops to be directly spread across all 32 processors. Cedar FORTRAN constructs are provided to specify easily loops which have no dependencies (DOALL) or limited dependencies (DOACROSS) [56],[57]. As of this writing, the DOACROSS loops may execute only on a single cluster. Facilities are in place so that DOALL loops

Table 3.2
Scheduling Algorithm on Cedar Clusters

| Level | Quantum | Choice 1 | Choice 2 | Choice 3 |
|---|---|---|---|---|
| 1 | 300 ms | cluster | IP/CE | CE |
| 2 | 300 ms | CE | IP/CE | cluster |
| 3 | 300 ms | IP/CE | CE | cluster |

spread iterations of the loop across all the processors of the machine. To do this, a CDOALL loop is nested in an SDOALL in the source code. Physically, helper tasks are created and scheduled to each cluster. The iterations of the outer loop (SDOALL) are then self-scheduled one at a time to each helping task. The inner loop (CDOALL) is then spread across the eight processors of the cluster.

The memory of Cedar is divided into four exclusive areas defined by the access permission (shared or private) and the locality (local or global) [58]-[60]. Data in shared memory are accessible by all tasks of an application, while data in private memory are accessible only by one task. The most commonly used memory is shared global and private cluster.

### 3.2. Measurement Facilities

The models and methodologies presented in this thesis are all based on measurements of real workloads, workloads found during normal machine operation over which there is no control. For the models and results to be accurate, the facilities used to gather the results must be reliable and must not significantly perturb the workloads. To accomplish this, a number of monitoring tools designed specifically for the high resolution timing of the Cedar and Alliant supercomputers were used. This section briefly describes the monitoring facilities used in this study and the types of measurements obtained.

Extensive work on multiprocessor monitoring tools has been done at CSRD [60]-[65]. Hardware, software, and hybrid tools, as well as performance visualization packages have all been developed. Measurements of the real workloads on the Alliants were obtained with three software measuring facilities: *Q*, *HRTIME*, and *VMSTAT*.

The Q facility monitors the utilization of each processor. The facility records the amount of time each processor spends idling, spinning on kernel locks, handling interrupts, executing system code, and executing user code. It maintains separate logs for each IP and CE while detached, but only a single log for the CEs while clustered. Measurements of the clustered CEs are actually measurements of

the master CE while clustered (the master CE dynamically changes according to the scheduler). The Q facility can also take a snapshot of the workload present at a given time. When used this way, Q reports the number of each type of job in the system (i.e., the number of cluster (A), cluster (C), CE (A), CE (C), IP, and IP/CE jobs).

The second facility, HRTIME, measures the completion time of an application with 10 $\mu s$ accuracy. The VMSTAT facility maintains counts of system activities such as context switches, CE configuration changes, device interrupts, and disk accesses. Normally, software monitoring facilities are not recommended for use with multiprocessors [66]. However, on the Alliant, all monitoring software was executed on the IPs so that the perturbation to the workloads on the CEs was minimal.

Measurements of the Cedar system were obtained with a distributed version of Q, a software facility called *getvmetc*, and a hybrid monitor called *P3S*. Each cluster of Cedar has its own copy of the Q facility. The facility collects the same metrics as those collected by the Q facility on the Alliants. The metrics are kept in the kernel memory of each separate cluster. The *getvmetc* system call returns virtual memory usage, paging, and memory overheads for the calling application. The facility is used to monitor the number of pages of each memory type (e.g., global shared, local private) that an application requires. The Parallel Program Performance evaluation System (P3S) allows accurate time stamps to be collected. The user indicates the time stamp in the user code, and it is collected with hardware monitors. A comprehensive history of context switch times is also maintained. The facility was used in this study to measure the completion time of applications accurately, as well as the completion time of loops and tasks within an application.

### 3.3. Target Applications

The methodologies presented in this thesis require the system to be monitored while a parallel application (the *target application*) executes under the real workloads. The target applications were chosen from the Perfect Club Benchmark Suite [55] and are summarized in Table 3.3. Dyfesm is a

Table 3.3
Target Applications (Alliant)
(NA - value not measured)

| Appl. | FX/8 Completion Time (s) | FX/8 % of clustered time exec system code | FX/80 Completion Time (s) | FX/80 % of clustered time exec system code |
|---|---|---|---|---|
| Dyfesm | 241 | 1.3 | 200 | 1.2 |
| Flo52 | 119 | 3.1 | 88 | 3.4 |
| Track | 147 | NA | 94 | 3.3 |
| Spec77 | 303 | 4.2 | NA | NA |
| BDNA | NA | NA | 134 | 7.7 |

computationally intensive application which performs two-dimensional dynamic finite-element structural analysis. Flo52 analyzes transonic flow past an air foil; Track performs signal processing to track a missile's path. BDNA simulates dynamic molecular behavior. These applications represent the current requirements of many parallel applications. All are computationally intensive and were compiled to execute as type A cluster jobs on the Alliants; they need all eight CEs in the clustered configuration to execute.

The applications were first measured on a dedicated machine (single-user mode) so that the base resource requirements of each could be determined. Measurements of completion time and percentage of system code executed in the parallel environment for both the FX/8 and FX/80 are summarized in Table 3.3. The amount of system code executed when an application is run on a dedicated machine is a measure of the inherent system work in that application.

The Cedar target applications are summarized in Table 3.4. The table shows the completion time on a dedicated Cedar, the number of clusters used by the application, and the type of concurrency the application exploits (loop or task). TFS is the Flo52 perfect benchmark ported to the Cedar supercomputer. It has been renamed because it is physically a different piece of code (Cedar FORTRAN constructs were used to write it). The application uses all four clusters and exploits loop concurrency. TFS_SC is actually the same piece of code as Flo52 shown in Table 3.3. It is used when a parallel application which executes on a single cluster is needed. It has been renamed to avoid confusion.

The applications ARC and MCP are both codes written for Cedar using task parallelism. ARC analyzes three-dimensional fluid flow problems by solving the Euler and Navire-Stokes equations; MCP is a sparse matrix solver. On a dedicated Cedar, MCP executes on a single cluster for 77.5 s and then forks off three tasks and completes in another 39.5 s. ARC starts by forking 3 tasks and all four tasks need 73 s of dedicated machine time before finishing.

The applications LOOP_CON and TASK_CON were created as minimum overhead examples of the two forms of Cedar concurrency under investigation. LOOP_CON is a large parallel loop of additions which are spread across the 32 processors. TASK_CON creates four tasks and each independently performs concurrent additions. Both applications exploit the full concurrency of the machine with minimum overhead.

### 3.4. Definitions

The terms multiprogramming, multiprocessing, and parallel processing have taken many different meanings in computer literature. In this thesis, multiprogramming refers to multiple users time-sharing processing resources. Multiprogramming is resource sharing across time. Multiprogramming's main purpose is higher throughput for short jobs. It is commonly implemented on both uni- and multiprocessors. Multiprocessing will be used to describe the concurrent execution of independent jobs on separate processors. Parallel processing will be used to describe the execution of a single application across multiple processors. Parallel and multiprocessing are resource sharing

Table 3.4
Target Applications (Cedar)

| Appl. | type of concurrency | # of clusters used | Completion Time (s) |
|---|---|---|---|
| TFS | loop | 4 | 73.5 |
| TFS_SC | NA | 1 | 108 |
| ARC | task | 4 | 73.4 |
| MCP | task | 1/4 | 117 |
| LOOP_CON | loop | 4 | 44.0 |
| TASK_CON | task | 4 | 53.7 |

across space. The Alliant and Cedar machines support multiprogramming, parallel processing, and multiprocessing.

There are two factors in multiprogrammed environments that increase a job's completion time. The more significant factor is the time-sharing of the processing resources. With more than one application present, any single application will receive less processor time and take longer to complete. The second factor is multiprogramming overhead (MPO), which is the extra work created as a result of the time-shared multiprogramming environment. The work can be due to increased system work (e.g., context switching, scheduling) or contentions among jobs (e.g., paging, spinning on locks).

An example will clearly illustrate the difference between the MPO and the time-sharing of resources. Assume a simple uniprocessor using a fair, time-shared, round-robin scheduling policy with time quantums several orders of magnitude smaller than job lengths. Now assume that a job takes $CT_{Ded.\ Mac.}$ to complete on the simple machine when multiprogramming is not in effect (dedicated machine). If this job is multiprogrammed with two other copies of the job (assuming no I/O), the completion time may be written as: $(3 \times CT_{Ded.\ Mac.}) + MPO$, where MPO is the processor time spent over and above that directly used by the user jobs (Figure 3.5).

One possible taxonomy of the MPO parameter breaks it into two components: the *base component* and the *real workload component.* The base component is the least amount of processor time needed to maintain persistent multiprogramming. It is a lower bound on the MPO and is constant independent of the workload on the machine. Its main component is time spent context switching. The real workload component varies with the load on the machine. It is caused mainly by increased resource (memory, network, kernel, spin lock) contentions.

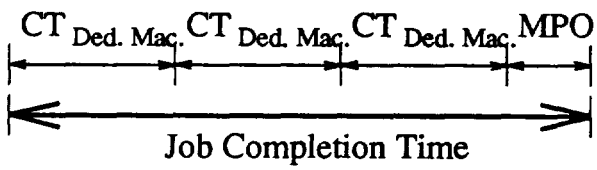



Figure 3.5 Completion Time in Multiprogrammed Environment

On a parallel machine with gang scheduling (such as the Alliant), the MPO is a subset of total system overhead. Total system overhead consists of all system work: MPO, as well as system work inherent in the user applications (such as I/O). An example will clarify the distinction between multiprogramming and total system overheads. Assume that a program is executed on a dedicated machine and requires Y amount of paging. Now, assume it is executed in a multiprogrammed environment and requires Y+Z amount of paging. The overhead associated with the Z amount of paging is counted as multiprogramming overhead, while all of the paging, Y+Z, is counted as total system overhead.

On a parallel machine without gang scheduling (such as Cedar), the MPO may incur an additional component due to spinning on user synchronization locks. For instance, assume that a task executing on cluster one is waiting on a lock held by a task on cluster two. In a multiprogrammed environment, the task holding the lock may be context switched off cluster two leaving the task on cluster one spinning helplessly. This extra time spent spinning is MPO. It is not normally considered system overhead because the processors are actually servicing the user program. On a gang scheduled machine, this component does not occur because all tasks are scheduled at once.

A few other terms should be discussed. The phrase *real workload* is used in the current literature to refer to anything from a single benchmark running on a machine to a group of synthetic programs. In this thesis, real workload refers to the work being done on a machine during normal operation. In other words, there is no control over what work is submitted to the system. Kernel spin locks refer to synchronization locks used to protect critical sections of code in the kernel. Spinning on them is considered system overhead. User spin locks are those created and used in the user application. As previously mentioned, spinning on these is not considered system overhead, but may be multiprogramming overhead.

# CHAPTER 4.

# MULTIPROGRAMMING OVERHEAD: BASE COMPONENT

The degradation of performance due to overheads, particularly multiprogramming overhead, is an important issue in the continuing development of parallel processing supercomputers. This is especially true as time-shared, interactive, multiprogramming environments become more prominent on parallel machines. In these environments, multiple jobs and users can dramatically increase the amount of system overhead which degrades the performance of the machine. Measurements of this overhead are therefore important for performance evaluation purposes.

Measurements of MPO are also useful from a supercomputer system design point of view. The practicality of a supercomputer which runs both large, parallel jobs and smaller, interactive serial jobs (such as editing) in a multiprogramming environment must be investigated. Would it be wiser to use a supercomputer as the back end of a computer hierarchy instead of as an all-purpose machine? If this were the case, only large, parallel jobs would be executed on the supercomputer, and a batch-oriented scheduler or larger time quantums could be used to reduce MPO.

Measurements of MPO will also prove useful for simulation and analytical modeling purposes. Overhead parameters used in simulations are often roughly estimated. With real workload overhead measurements, the accuracy of these parameters need no longer be questioned.

In this chapter and the next chapter, the question of multiprogramming, as well as system overhead will be investigated. In this chapter, two techniques that estimate the base component (lower bound) of the MPO are introduced and then illustrated using the Alliant FX/8 and Cedar supercomputers. The techniques are called the *Completion Time Estimation Technique* and the *Limit Technique.*

Most scheduling paradigms divide jobs into different classes and the jobs of different domains will be affected differently by multiprogramming. For instance, multiprogramming serial jobs may cause more or less overhead than multiprogramming parallel jobs. Therefore, the lower bound on

MPO must be determined for a given job domain executing in a given workload type. For instance, using the Alliant FX/8 and the above techniques, the lower bound on MPO is determined for multiple parallel jobs (type A cluster jobs) and for multiple parallel jobs executing with serial jobs (type A, CE jobs). The effect of serial jobs on parallel jobs is then isolated.

On the Cedar supercomputer, the lower bound on MPO for executing multiple parallel jobs written with loop concurrency and multiple jobs written with task concurrency is determined. Executing a loop concurrent job with multiple task concurrent jobs, and a task concurrent with multiple loop concurrent jobs is also investigated. It is found that the base component of overhead for multiprogramming parallel jobs is about twice as high on the Cedar as the Alliant FX/8. The techniques presented in this chapter are adaptable to most computer systems available today.

### 4.1. Lower Bound MPO: Alliant FX/8

The techniques which determine MPO require total control over the workload and environment of the system. This is achieved by placing the Alliant FX/8 in single-user (SU), dedicated mode. Although the workloads are completely constructed, measuring performance on real machines always introduces uncontrollable factors. Because of this, even the completion time of an application running alone on a dedicated machine will vary with different executions. Therefore, all experiments throughout this thesis that are done in a controlled environment are performed at least five times. The largest and smallest measures of the multiple experimental runs are discarded and the rest are then averaged. Therefore, although a single value is presented, it is actually an average of repeated experiments.

The techniques to determine the lower bound on MPO require a target application to be executed and monitored in controlled workloads consisting of *dummy jobs*. Measurements of target application completion time and the percentage of system overhead executed in these controlled workloads are then used to determine the lower bound on MPO.

The dummy jobs are constructed to contribute negligible system overhead when executed (i.e., no system calls or paging). Their purpose is to create workloads with different degrees of multiprogramming. As mentioned, the techniques require a specified number of dummy jobs to execute while a target application is run. To limit system overhead in this situation, the dummy jobs must begin before and finish after the target application. In this fabricated workload, the minimum amount of overhead caused by multiprogramming is seen by the target application.

The lower bound (base factor) on MPO is determined for three programming situations: multiprogramming parallel jobs, multiprogramming multiple parallel and serial jobs, and multiprogramming a single parallel job with serial jobs.

#### 4.1.1. Lower bound MPO: parallel jobs

This section uses the two techniques to quantify the lower bound on MPO when only parallel jobs are executed. Dummy jobs were constructed to be type A cluster jobs consisting of a tight loop of concurrent additions. Three target applications were used for this experiment: Dyfesm, Spec77, and Flo52. Each target application was executed and measured in six different controlled workloads: first by itself, and then with from one to five dummy jobs. Using the measurements from these runs, the lower bound on parallel job MPO was determined to be 4% of the parallel processing time.

##### 4.1.1.1. Completion time estimation technique

The Completion Time Estimation Technique consists of measuring the completion time (CT) of the target application while it is running in the dummy job workloads. The completion time assuming no MPO ($CT_{w/o\ MPO}$) is then estimated. The amount of MPO is obtained by subtracting the estimated completion time from the measured completion time ($CT - CT_{w/o\ MPO}$). The $CT_{w/o\ MPO}$ for a workload can be computed using the completion time of the target application on a dedicated machine ($CT_{Ded.\ Mac.}$), the number of dummy jobs in the workload, and details of the scheduler.

The $CT_{Ded.\ Mac.}$ was obtained from Table 3.3, and the estimation of $CT_{w/o\ MPO}$ is given by Equation (4.1). The equation is based on the fact that the target application and dummy jobs share the clustered CEs equally. It is accurate for applications that are computationally bound, but because of I/O overlapping, it is less accurate for applications with significant I/O.

$$CT_{X,w/o\ MPO} = (X+1) \times CT_{Ded.\ Mac.} \tag{4.1}$$

$CT_{X,w/o\ MPO}$ : completion time with X dummy jobs assuming no MPO

Figure 4.1(a) shows the measured $CT$ (solid lines) and $CT_{w/o\ MPO}$ (dotted lines) for each target application as a function of the multiprogramming level (number of dummy jobs). The MPO is the distance between the solid and dotted lines for each application. The average difference between the solid and dotted line for Dyfesm (the application with the least I/0) is 4% of the corresponding measured completion time. Therefore, 4% of the parallel processing time was consumed by MPO. This is a lower bound because the low overhead dummy jobs guarantee that there do not exist workloads which can cause less MPO.

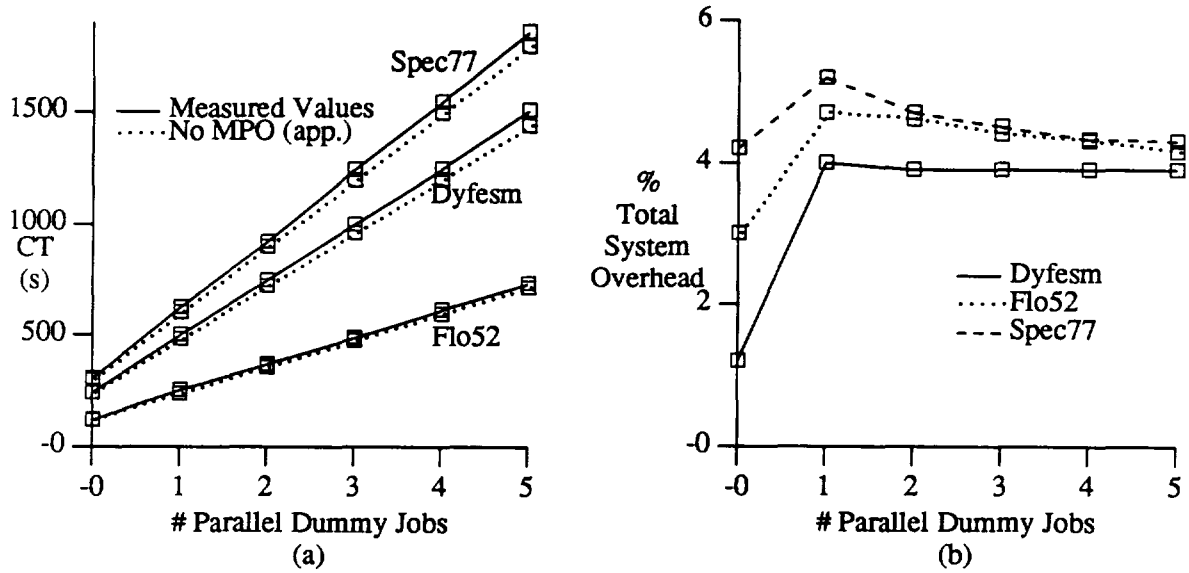



Figure 4.1 Multiprogramming Parallel Jobs — Completion Times and System Overheads

#### 4.1.1.2. Limit technique

The Limit Technique does not require the target application to be computationally intensive to calculate the lower bound on MPO. The technique is based on the fact that dummy jobs contain little inherent system overhead; thus total system overhead consists of a higher percentage of MPO as the level of multiprogramming increases. In other words, as a target application is executed with more dummy jobs, the system overhead approaches the multiprogramming overhead.

Equation (4.2) can be used to explain this phenomenon more clearly. In the equation, $B$ is the base percentage of MPO executed, $i$ is the raw amount (not percentage) of inherent system code in the target application, and $t$ is the completion time of the target application when executed with the additional dummy jobs. Because the dummy jobs add little system work of their own, all system code can be attributed to either MPO or system code from the target application. Therefore, the percentage of clustered time spent executing system code, $S$, can be expressed by Equation (4.2). As the number of additional cluster jobs increases, $t$ also increases, while $i$ remains constant: $S$ (the parameter measured) approaches $B$ (the parameter desired).

$$S = B + \frac{i}{t} \tag{4.2}$$

The Q facility was used to measure the percentage of clustered CE time executing total system overhead (S). Figure 4.1(b) shows a graphs of system overhead as a function of the multiprogramming level. Notice that for all three applications, system overhead approaches 4%. This result corroborates the result obtained with the previous technique. The base component of MPO for parallel jobs consumes 4% of the parallel environment processing power.

### 4.1.2. Lower bound MPO: parallel and serial jobs

The lower bound (base component) of overhead due to multiprogramming both cluster (parallel) and CE (serial) jobs is now quantified. The cost of reconfiguring the CEs between the clustered and detached configurations is an integral component of this overhead.

The experiments of the last section were repeated, except that in each case, two streams of CE (A) jobs were run while the target and dummy applications executed. A stream of CE (A) jobs is a continuous string of identical jobs CE (A) (i.e., when one job finishes an identical one is started) which have been constructed to minimize their inherent system overhead. At any given time in the experiments of this section, there are two CE (A) jobs, the target application, and a number of dummy jobs executing on the system.

Two CE job streams (instead of one or three) were used for two reasons. First, two CE jobs are the minimum number needed to cause the CEs to physically reconfigure. Second, it was found that additional CE jobs after the first two did not affect the MPO. Therefore, the results obtained using two streams of CE jobs are valid for any environment with two or more CE jobs running.

#### 4.1.2.1. Completion time estimation technique

The Completion Time Estimation Technique was used to determine the lower bound on MPO for the case of both parallel (cluster) and serial (CE) jobs by modifying the equation for $CT_{w/o\ MPO}$ to account for the CEs being clustered only 54% of the time (Equation (4.3)). The CEs were clustered only 54% of the time because the other 46% was used to execute the CE serial job streams (see Alliant scheduler, Table 3.1).

$$CT_{X,CE} = \frac{1}{0.54} \times (X+1) \times CT_{Ded.\ Mac.} \tag{4.3}$$

$CT_{X,CE}$ : completion time with X dummy jobs,
CE job streams, and no MPO

Figure 4.2(a) shows the measured completion times of the target applications (solid lines) and the approximations of completion times assuming no MPO (dotted lines). As before, the difference between a solid-dotted pair is the amount of time spent executing MPO on the cluster. For Dyfesm, the average difference between the two values made up 5.5% of the measured completion time.

Therefore, when both parallel and serial jobs are executed on the Alliant, at least 5.5% of clustered time is consumed by multiprogramming overhead.

#### 4.1.2.2. Limit technique

The lower bound on MPO due to parallel and serial jobs was also estimated using the Limit Technique. No changes had to be made in this case to use the methodology. With serial jobs present, the percentage of clustered time spent executing system code still approaches the MPO as additional parallel dummy jobs are added to the system. This can easily be seen by reexamining Equation (4.2). In fact, the completion time, $t$ increases even more quickly due to the serial jobs.

Figure 4.2(b) summarizes the results of this experiment. For both applications, the percentage of clustered time spent executing system code approaches 5.3% as more parallel dummy jobs are executed. Therefore, this method estimates the base cost for multiprogramming continuous CE and cluster jobs to be 5.3%. This corroborates the value obtained with the Completion Time Estimation Technique.

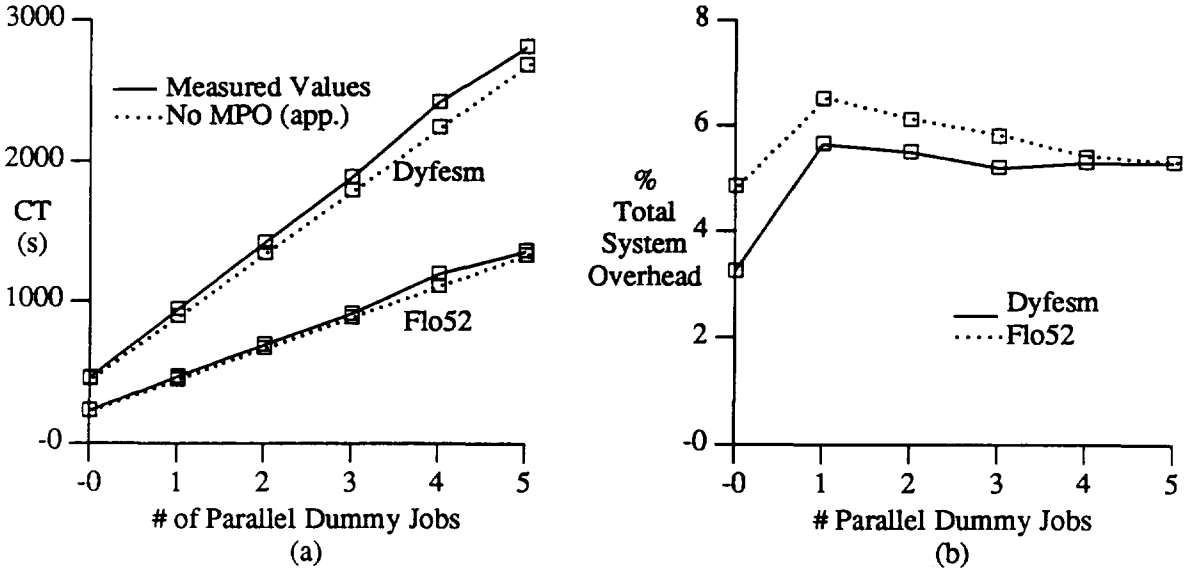



Figure 4.2 Multiprogramming Parallel and Serial Jobs — Completion Times and System Overhead

#### 4.1.3. Lower bound MPO: serial jobs

The last section quantified the lower bound on MPO in the parallel environment (clustered CE time) when there were both parallel and serial jobs executing. In this section, multiprogramming overhead degrading the performance of parallel jobs caused entirely by serial jobs will be isolated and quantified. Instead of multiprogramming multiple parallel and multiple serial jobs, a single parallel job is multiprogrammed with many serial jobs. The experiments will show that, similarly to cluster jobs, extra CE jobs, after two, do not increase the base factor of MPO.

The experiments consisted of executing and measuring a single target application (Dyfesm or Flo52) with single and multiple (2, 3, and 4) CE job streams. With a single CE job stream in the system, the CEs remained clustered the entire time (recall from Section 3.1.1); with two or more CE job streams, the CEs were clustered 54% of the time.

##### 4.1.3.1. Completion time estimation technique

The Completion Time Estimation Technique translated directly to this situation by setting $X$ equal to zero in Equation (4.3). In other words, no matter how many streams of CE jobs are added to the system, the completion time of a parallel job will remain constant if there is no MPO.

Figure 4.3(a) shows the completion time of the target applications in the multiple CE job stream environments. The projected $CT_{w/o\ MPO}$ is not shown because it is so close to the actual completion time. However, the calculation showed that, for these experiments, 2% of the CE clustered time was consumed by MPO. Therefore, in a workload with multiple CE jobs and a single cluster job, at least 2% of the clustered time will be consumed by MPO.

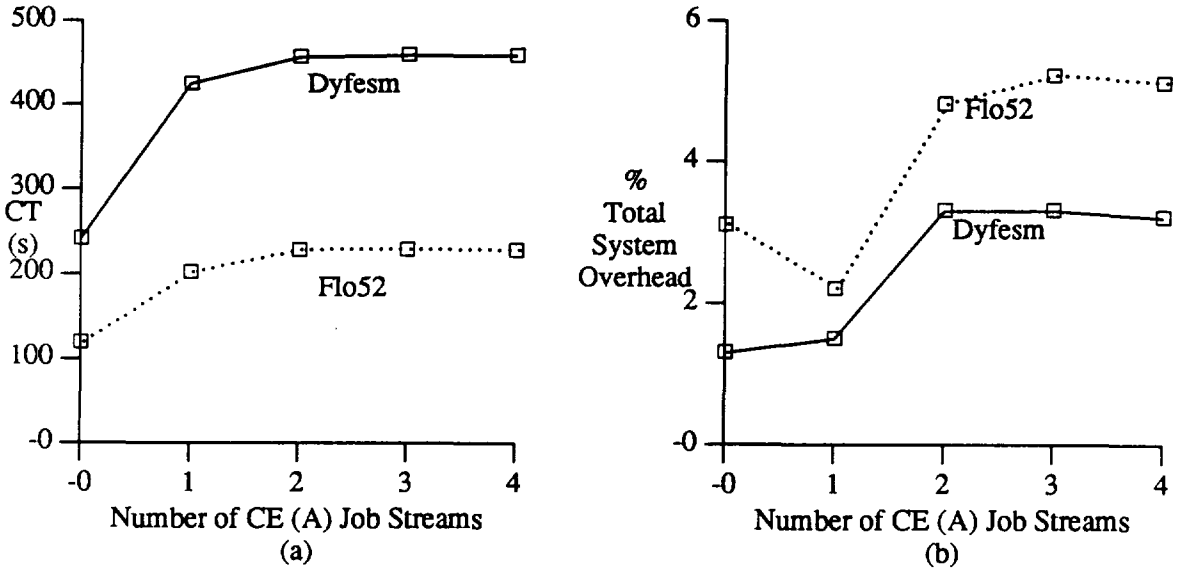
Notice that the completion time of an application executing with two or more CE job streams is more than that when it is executing with only one stream. Recall that if there is only one CE job in the system, the CEs may remain clustered and execute it. Therefore, the added completion time in the environments with two or more streams is largely due to overhead caused by repeatedly reconfiguring

the CEs between detached to clustered configurations. Notice also that the completion times of applications executing with two or more detached job streams are nearly identical. This indicates that the base factor of MPO remains unchanged when there are two or more CE jobs in the system.

#### 4.1.3.2. Limit technique

Figure 4.3(b) shows the percentage of clustered time spent executing system overhead for these experiments. The Limit Technique could not be directly used to determine the MPO because dummy jobs were not added to the system for these experiments. Therefore, the target applications' completion times did not increase, and the system overhead inherent in the target application could not be eliminated. In other words, the $t$ of Equation (4.2) remained relatively constant, allowing $i$ to remain significant.

However, the MPO has been determined by subtracting the percentage of system overhead executed by the clustered CEs when the application ran alone from the percentage executed when the application was run with the CE job streams. This difference is the increase in system overhead caused by the CE job streams. The results of this test confirm the 2% measurement obtained by the




Multiprogramming Serial Jobs — Completion Times and System Overhead

previous technique. It should be noted that a large portion of this 2% is due to exploding and coalescing the CEs.

### 4.2. Lower Bound MPO: Cedar Supercomputer

The lower bound on overhead for multiprogramming the various types of jobs on the Alliant FX/80 can be assumed to be close to that found for the Alliant FX/8 in the last sections. This is because the architectures of the machines are identical, the scheduling paradigms and job classes are the same, and the operating systems are similar. However, the lower bound on overhead for multiprogramming on Cedar cannot be predicted from studying the numbers on individual Alliants.

It can be fairly assumed that multiprogramming parallel jobs on the four clusters of Cedar will create more overhead than multiprogramming on an Alliant FX/8 (single cluster). After all, the Cedar operating system (Xylem) is built on top of Concentrix. Therefore, scheduling on Cedar adds an extra software layer (the Xylem Server) on top of Alliant's scheduler. The effect on the base component of multiprogramming overhead though is less clear.

In this section, the base component of overhead caused by multiprogramming parallel jobs which use all processors of Cedar is quantified. The Completion Time Estimation Technique will be used to study both loop concurrent and task concurrent parallel jobs. Two modes of Cedar operation will be investigated: single-user mode (such as that on the Alliant) and batch mode. It will be seen that the base component found in the single-user (SU) mode is similar to that found for parallel jobs on the Alliant: approximately 4%. The base component in the batch mode is significantly higher at approximately 11%. It will also be found that loop concurrent jobs tend to have a slightly larger base component than task concurrent jobs.

To control the level of multiprogramming, two types of dummy jobs were constructed: *loop dummy* and *task dummy* jobs. As in the case with the Alliants, the dummy jobs were constructed to cause a minimum of system overhead and paging. The loop dummy jobs consisted of a never ending

SDOALL loop. Each iteration of the SDOALL loop was a CDOALL loop which did nothing but add the same two numbers together repeatedly. The task dummy jobs consisted of 4 separate tasks, each a loop of concurrent additions. Therefore, both dummy jobs kept all 32 processors consistently busy with additions.

#### 4.2.1. Lower bound MPO: Cedar parallel jobs (loop concurrent)

The LOOP_CON program (Table 3.4) was used as the target application to determine the lower bound on MPO when a loop concurrent application is multiprogrammed with other parallel applications. Two different types of parallel workloads were investigated. One consisted of the target application and varying numbers of loop dummy jobs. The other was made up of the target application and varying numbers of task dummy jobs. Each experiment was conducted both in single-user mode and in the batch mode.

Determining $CT_{w/o\ MPO}$ was accomplished using Equation (4.1). The equation holds because all parallel jobs executing on Cedar have the same priority and are scheduled in a round-robin fashion. Conceptually, for these experiments, Cedar can be viewed as a single resource being shared by multiple consumers (target application and dummy jobs) all with the same access privileges. The parameter $CT_{Ded.Mac.}$ is found in Table 3.4.

The solid line of Figure 4.4(a) shows the completion time of LOOP_CON as a function of the number of loop dummy jobs executing (in the batch mode). The solid line of Figure 4.4(b) shows the completion time as a function of task dummy job multiprogramming level (in the batch mode). The dashed line in each figure show the completion time of LOOP_CON in the different environments with Cedar in single-user mode. The dotted line in each figure shows the approximated completion time of LOOP_CON with no MPO. For both cases the average difference between the dotted (ideal) and solid (real batch) line was about 11.5% of the real completion time. The corresponding value found in single-user mode was 4.7%.

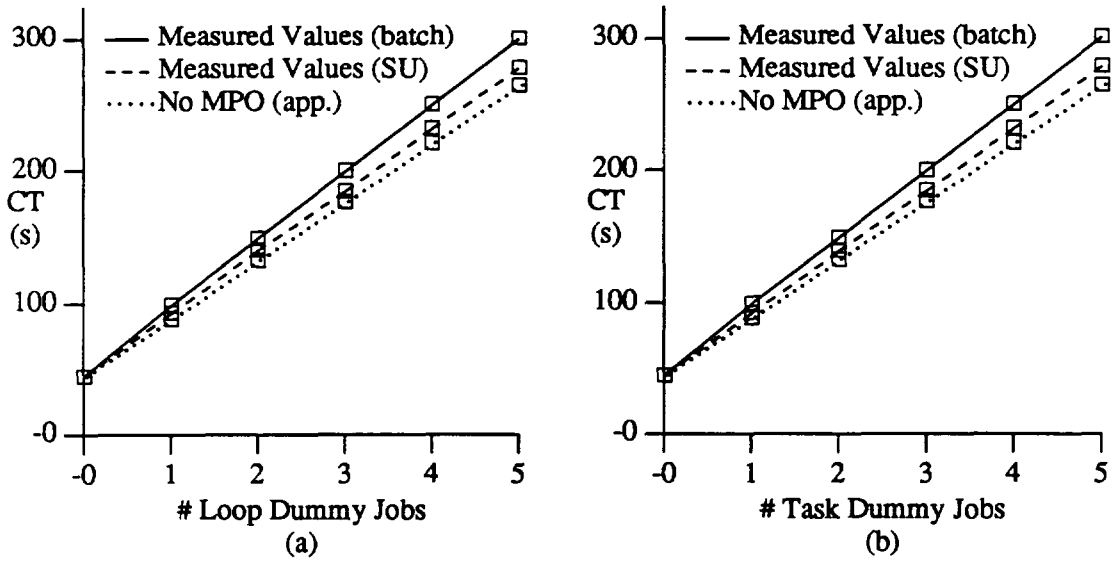



Figure 4.4 Multiprogramming Cedar Loop Parallel Jobs

It is interesting to note that the type of background workload (loop or task concurrent) does not greatly affect the lower bound on multiprogramming overhead. Therefore, if a parallel job written with the loop concurrency constructs is executed with other parallel jobs (either loop or task) on Cedar then at least 4.7% of the processing power will be consumed by MPO.

#### 4.2.2. Lower bound MPO: Cedar parallel jobs (task concurrent)

The TASK_CON program (Table 3.4) was used as the target application to quantify the lower bound on MPO created when multiprogramming a task concurrent application with other parallel jobs. Identically to the previous section, the TASK_CON program was executed in both loop and task dummy job environments. The experiments were also conducted in both single-user and batch modes. Equation (4.1) could again be used to determine $CT_{w/o\ MPO}$ for TASK_CON in the multiple dummy job environments. The reasoning behind the use of the equation is identical to that given in the last section.

Figures 4.5(a) and b show the results. The solid lines shows the actual measured completion time found in batch mode while the dashed line shows the completion time found in single-user mode.

The dotted lines show the approximation of $CT_{w/o\ MPO}$ from Equation (4.1) and the numbers of Table 3.4. In the batch mode, for both the multiple task dummy and multiple loop dummy environments, approximately 10.5% of the processing power is consumed by MPO. In single-user mode this number is much less at 3.8%. Notice that the single-user number is close to that found for parallel jobs on the Alliant.

Note that both situations (batch and single-user) give slightly lower values for multiprogramming the TASK_CON job than the LOOP_CON job (3.8 vs. 4.7 and 10.5 vs. 11.5). However, the difference is not that large. The higher value for the loop concurrent application is probably due to the extra synchronization needed among the helper tasks when executing the loop across 32 processors. Recall that TASK_CON creates four noncommunicating tasks, while the helper tasks of LOOP_CON must synchronize occasionally.

In conclusion, measurements from the batch mode place the base component of multiprogramming parallel jobs on Cedar at approximately 11%. Single-user mode does much better with the base component of multiprogramming parallel jobs placed at 4.2%. The difference is most likely due to extra daemons executing while the machine is in the batch mode. For both cases, the type of parallel

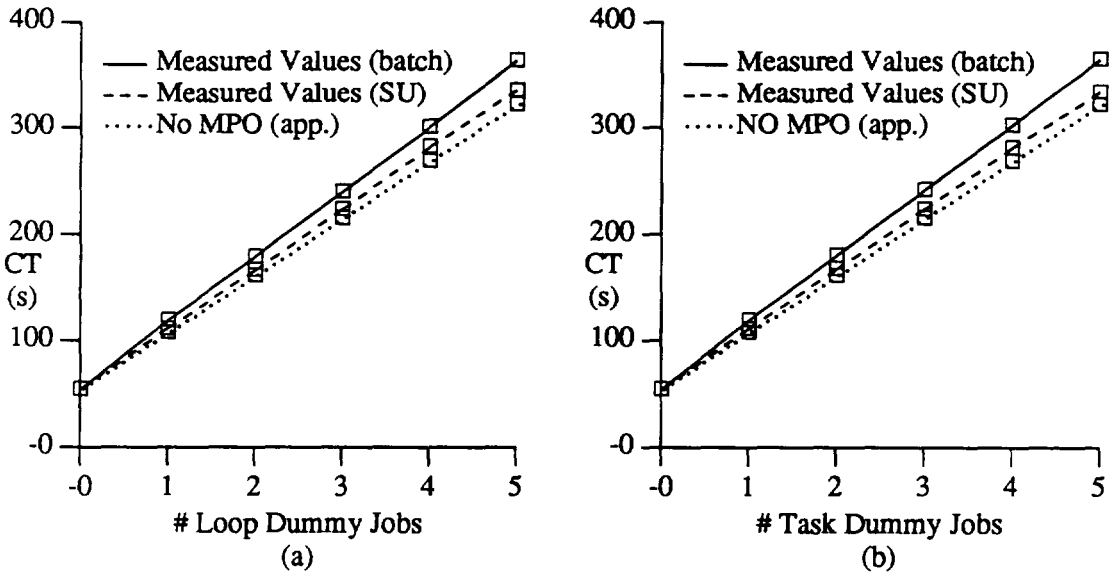



Figure 4.5 Multiprogramming Cedar Task Parallel Jobs

job— loop or task— does not seem to make a large difference in the base component of MPO. However, this does not mean that in real workloads using real applications, the MPO incurred by task parallel and loop parallel jobs will be the same. The next chapter will deal with this issue in more detail.

# CHAPTER 5.

# MULTIPROGRAMMING AND SYSTEM OVERHEADS: REAL WORKLOADS ON THE ALLIANT FX/80

Lower bound measurements of MPO are valuable because they are constant, single numbers that provide insight into the best-case behavior. They can be easily used to compare the performances of different machine environments. Unfortunately, the MPO found in real workloads usually exceeds the lower bound. In addition, the MPO incurred in real workloads is highly dependent on the characteristics of the workload. Therefore, it would be beneficial to estimate MPO for a large number of real workloads to understand the range of values that MPO may take for a given machine. Also, to understand better the causes of MPO and system overheads, it would useful to correlate the multiprogramming and system overheads found in real workloads to the characteristics of the workloads.

In this chapter, a technique to estimate MPO in real workloads (workloads found during the normal operation of the machine) is presented. The method is similar to the Completion Time Estimation Technique introduced in the last chapter. However, instead of executing a target application in a controlled workload, it is executed during the normal operation of the machine. Pertinent aspects of the real workload are periodically sampled as the target application executes. From the collected data, the $CT_{w/o\ MPO}$ and correspondingly the percentage of MPO incurred under that workload are estimated. The technique is adaptable to most computer systems available today. In this chapter the methodology is illustrated with real workloads found on the Alliant FX/80.

Real workload measurements of other overheads such as total system overhead and kernel lock spinning are also collected. These measurements are then used in a comprehensive case study of performance degradation due to overheads. Results from the Alliant workloads show that MPO makes up well over half of the total system work done by the machine. Most of the real workload MPO estimates fall between 10% and 23% of the processing power, with the mean being 16%. This is well above the lower bound on MPO which was estimated to be 5.5%. Measurements also show that

degradation caused by handling interrupts is small, while the time spent spinning on kernel locks is substantial (5%-10%). Correlation analysis suggests that for the workloads studied, MPO is not dependent on the number of parallel jobs being multiprogrammed. In addition, it is found that through increased kernel lock spinning, serial jobs, even those executing on peripheral processors, contribute greatly to MPO.

This chapter is organized as follows. Section 5.1 introduces the machine-independent methodology which estimates the MPO in workloads (real or synthetic). Section 5.2 then presents details necessary to use the methodology on the Alliant FX/80. Section 5.3 summarizes the types of workloads measured and presents a number of preliminary measurements which characterize them. Section 5.4 contains real workload multiprogramming and system overhead measurements. The system overhead measurements are measured directly with the Q facility, while the multiprogramming overhead measurements are estimated using the methodology. Section 5.5 investigates the relationship between the workload sampling period and the accuracy of the MPO estimate. Finally, in Section 5.6, correlation analysis is used to investigate the relationships between overheads and workload characteristics.

### 5.1. MPO Estimation Methodology: Machine Independent

This section introduces the methodology which estimates the MPO in real workloads. Conceptually, the machine-independent methodology is extremely simple. However, as will be seen in the other sections of this chapter, the actual use of the methodology becomes complex when real workloads over which there is no control are encountered.

To estimate the MPO of a workload, a modified version of the Completion Time Estimation Technique is used. A target application is executed in the workload under investigation, and its completion time is measured. The completion time of the application in the same workload, assuming no MPO ($CT_{w/o\ MPO}$), is then estimated. The percentage of processor time spent executing MPO (%MPO) can then be estimated using Equation (5.1).

$$\%MPO = \frac{CT - CT_{w/o\ MPO}}{CT} \times 100 \tag{5.1}$$

At this point the reader may be aware of an inconsistency in the use of the term overhead and the calculation of such. Admittedly, overhead is normally determined by dividing by the completion time without the overhead instead of the actual completion time. In other words, the true definition of overhead requires the denominator of Equation (5.1) to be $CT_{w/o\ MPO}$ and not CT. As the equation stands, %MPO is the percentage of the completion time of the target application that is consumed by MPO. It will be seen that this number is what is needed for the construction of our application execution model (Chapter 7). Therefore, the reader is cautioned to bear in mind the definition of overhead as used in this thesis.

Equation (5.1) is machine independent; however, the determination of $CT_{w/o\ MPO}$ is not. To compute $CT_{w/o\ MPO}$, complete information about the other jobs in the workload (e.g., types of jobs, priority of jobs, lengths of jobs), the scheduling paradigm, and the base resource requirements of the target application are needed. With this information, the execution of the job in the workload, assuming no MPO, can be analytically simulated and $CT_{w/o\ MPO}$ can be estimated.

For example, assume a uniprocessor with a fair, time-shared, round-robin scheduling policy with time quantums several orders of magnitude smaller than the job lengths. Assume that a target application is submitted to this system and the workload for the entire execution of the application consists of three other jobs (all jobs have the same priority). If there were no MPO then the completion time of the target application would be four times its completion time on a dedicated machine. Of course, this is a simple example, but the philosophy is used to determine $CT_{w/o\ MPO}$ for more complicated scheduling paradigms and changing workloads.

In real situatoins, the scheduler and base resource requirements for the target application are usually not difficult to obtain. However, a constantly changing workload is hard to monitor because jobs of different classes and priorities enter and leave the system at undetermined times. The workload information needed, however, can be approximated by frequently sampling the parameters pertinent to

scheduling. If the sampling period is small enough, the sampled data will accurately reflect the actual workload.

Figure 5.1 illustrates the sampling technique. The solid line in the figure shows the number of class X jobs in an imaginary system as a function of time. The queue length of this job class is sampled at the instances indicated by the arrows. The dotted line in the figure shows the workload reconstructed from the samples. Obviously, if the sampling period is small enough, the reconstructed workload will accurately represent the actual workload.

In the following section the methodology and sampling technique are illustrated with details for workloads on the Alliant FX/80.

### 5.2. MPO Estimation: Alliant FX/8 and FX/80

To estimate the MPO for a given workload, a target application from Table 3.3 is executed in the workload and the completion time is measured. The $CT_{w/o\ MPO}$ is then estimated, and Equation (5.1) is used to determine the percentage of the parallel environment which was spent executing MPO.

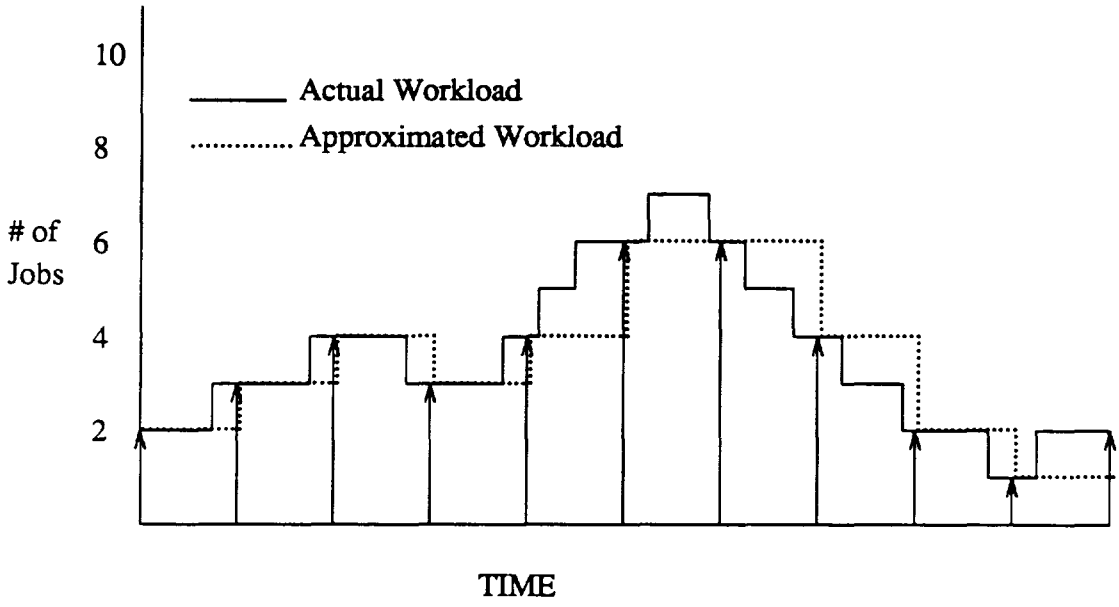



Figure 5.1 Sampling Single Job Queue

As mentioned in the previous section, $CT_{w/o\ MPO}$ is estimated using scheduling related information about the jobs in the workload, the scheduling paradigm, and the base resource requirements of the target application. The Alliant scheduler (Table 3.1) and base resource requirements of the target applications (Table 3.3) have already been discussed. The workload parameters crucial to scheduling and therefore sampled were the numbers of each type of job on the system. The sampling procedure is illustrated by Figure 5.2.

Using the sampled workload data and the scheduling information, $CT_{w/o\ MPO}$ is analytically estimated. For the Alliant, the goal is to determine the percentage of CE clustered time (time when the CEs are clustered) that is consumed by MPO. This requires two numbers, to be estimated. The first number is the percentage of time the CEs were clustered and executing cluster jobs (type A or C) while the target application executed. This will be referred to as the *Percentage of Available Clustered Time (%ACT)*. The second number is the percentage of the %ACT that would be granted to the target application if there was no MPO. From these two numbers $CT_{w/o\ MPO}$ is estimated using Equation (5.2). The equation takes the rate of processing power ideally given to the target application (no MPO) and divides it into the needed processing power ($CT_{Ded.\ Mac.}$).

$$CT_{w/o\ MPO} = \frac{CT_{Ded.\ Mac.} \times 100 \times 100}{\%ACT \times (\%ACT\ granted\ application)} \tag{5.2}$$

For instance, assume the target application requires 100 s of clustered CE time to complete, $CT_{Ded.\ Mac.} = 100\,s$. Also assume that the application is executing in a workload which would grant 5% of the clustered CE time to that application if there were no MPO (denominator of Equation (5.2)). Then, the $CT_{w/o\ MPO}$ would be $\frac{100}{0.05} = 2000\,s$.

Determining %ACT will be discussed first. The Q facility can directly measure the percentage of time the CEs were clustered (%CLUSTERED). However, all of this time was not necessarily granted to cluster jobs. If there was a single CE or IP/CE job on the system (in addition to the cluster jobs), then the CEs would remain clustered when executing it. To determine %ACT, the percentage

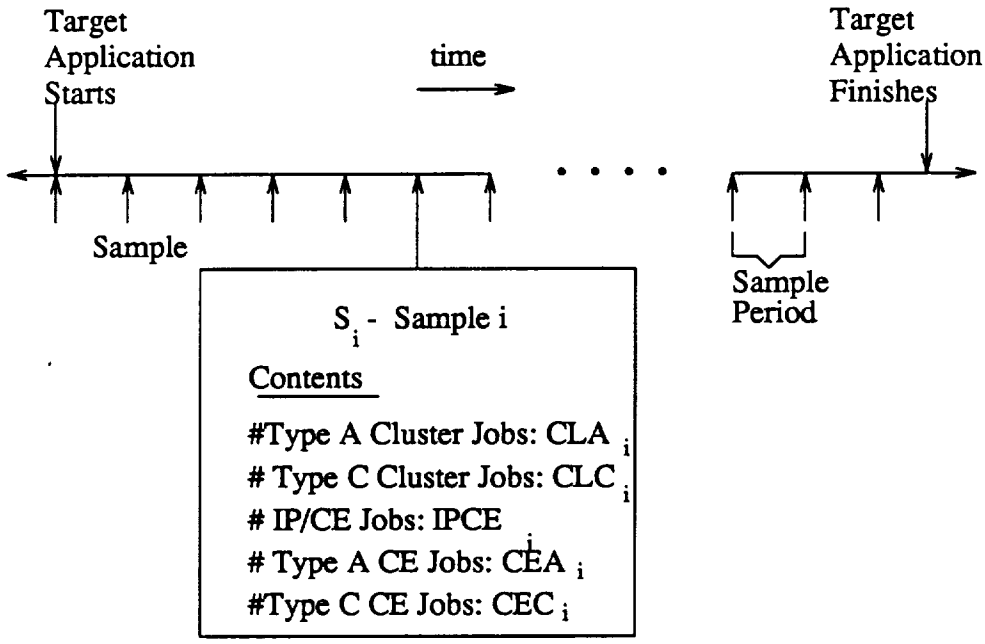



Figure 5.2 Alliant Workload Sampling Technique

of time the CEs were clustered but executing a single CE or IP/CE job is subtracted from %CLUSTERED.

The percentage of clustered time spent executing a single CE or IP/CE job is estimated from the sampled data. Let $\%CE_1$ be the percentage of samples containing one CE job and no IP/CE jobs. For this case, the CEs would remain clustered and execute the single CE job 6/13 of the time (see Alliant scheduler, Table 3.1). Similarly, let $\%IP/CE_1$ be the percentage of samples containing one IP/CE job and no CE jobs. For this case, the IP/CE job would be executed by either the clustered CEs or an IP. Assume that an IP executes the IP/CE job half of the time and the clustered CEs the other half; therefore, on average, 3/13 (1/2 of 6/13) of the clustered CE time would be spent executing the job. Equation (5.3) combines all of this information to estimate %ACT.

$$\%ACT = \%CLUSTERED - (\%CE_1 \times 6/13) - (\%CE/IP_1 \times 3/13) \tag{5.3}$$

The parameter %ACT is literally the percentage of time in which the target application is executing that the CE complex was clustered and executing cluster jobs. There are no assumptions made

about MPO in the estimate of Equation (5.3). The parameter %ACT can also be viewed as the average percentage of processing power granted to cluster jobs (including the target application). The percentage of this processor power granted to the target application assuming no MPO must now be determined. This is done by estimating this value for each sampling period and averaging all samples taken during the execution of the target application.

If sample $i$ contains $CLA_i$ type A cluster jobs (including the target application) and no type C cluster jobs ($CLC_i = 0$), then the percentage of %ACT granted to the target application for that sampling period is estimated by (5.4a). Basically, the equation states that all cluster A jobs share the available cluster time equally. If, however, there were one or more type C cluster jobs in the sample, ($CLC_i > 0$) then (3.4b) provides the estimate. For simplicity, Equation (5.4b) assumes that there were one or more serial jobs present. The $\frac{3}{7}$ accounts for level 2 in the scheduling algorithm (Table 3.1). The parameter %ACT is determined by averaging $\%ACT_i$ over all samples (Equation (5.4c)).

$$\%ACT_i = \frac{1}{1 + CLA_i} \times 100 \qquad CLC_i = 0 \tag{5.4a}$$

$$\%ACT_i = \frac{1}{1 + CLA_i} \times \frac{3}{7} \times 100 \qquad CLC_i > 0 \tag{5.4b}$$

$$\%ACT = \frac{\sum_{all\ i} \%ACT_i}{Number of Samples} \tag{5.4c}$$

Equations (5.3), (5.4a), and (5.4b) are derived from the scheduling information (Table 3.1). It should be noted that time gained by I/O cycle stealing is accounted for in this estimate. If there was significant I/O overlapping, the estimated $CT_{w/o\ MPO}$ will tend to be higher, and hence the MPO estimate will be lower. This is fine because I/O cycle stealing is a benefit of multiprogramming and should be credited in its overhead measurement.

Once $CT_{w/o\ MPO}$ is estimated using the above equations, Equation (5.1) is used to determine %MPO. The final %MPO will be the percentage of clustered CE time (not total CE time) consumed by MPO. This is acceptable because it quantifies the MPO that degrades the performance of solely

parallel jobs. The MPO executed on detached CEs or IPs will not degrade the performance of parallel jobs.

### 5.3. Alliant Workloads

The techniques introduced above were used to determine the MPO for multiple workloads on an Alliant FX/8 and Alliant FX/80 at CSRD. The Alliant machines are used heavily at CSRD for algorithm development and general scientific computing. The workloads studied on the Alliants were chosen randomly during the normal operation of the Alliant machines. There was no control over what type of work was being done while the study was conducted. This section summarizes the workloads measured and presents some general characteristics of the workloads under investigation (e.g., paging, queue lengths).

#### 5.3.1. Experiment summary

The MPO is determined by executing a target application in the workload under investigation. To illustrate that the methodology can work with a number of target applications (as long as the application is from the correct domain), three different target applications were used in the study: Dyfesm, Flo52, and Track. Another variable in the methodology not yet addressed is the frequency in which the samples are taken— the sampling period. The choice of the sampling period must balance the opposing goals of reducing intrusiveness (long sampling period) while accurately capturing the workload (short sampling period). Three sampling periods were experimented with: 30 s, 10 s, and 1.3 s. The effect of the sampling period on accuracy will be investigated in Section 5.5.

With these three target applications and three sampling periods, the MPO was determined for a large number of randomly selected times during the normal operation of an Alliant FX/8 and FX/80. Each independent determination of MPO is referred to as an *observation*. Each observation was classified into an *experiment* according to the target application, sampling period, and machine used in

collecting it. Each experiment contains all observations collected using the same target application, sampling period, and machine. Table 5.1 summarizes the experiments conducted and gives the number of observations in each. For instance, experiment Flo_30 is the results obtained by executing and measuring, with a sampling period of 30 s, Flo52 49 times in real workloads found on the Alliant FX/80.

The collection of observations for an experiment took from 2 to 4 weeks. The measurements for the entire study, 330 observations, cover an 8 month period.

The Alliant FX/8 was used for only one experiment, Dyfe_30_fx8, because access to the machine was lost and replaced by an FX/80 during the course of the study. Nonetheless, the forty-two observations were of value in comparing the two machines.

### 5.3.2. Workload characteristics

In this section, general characteristics of each experiment's workload are identified. In a later section the relationships between these characteristics and multiprogramming and system overheads will be investigated.

Table 5.2 summarizes job queue length measurements for the observations in each experiment. Mean job queue lengths are found by averaging the mean queue lengths for each observation in an experiment. An observation's mean queue lengths are the average number of each type of job in the

Table 5.1
Experiment Summary

| Experiment ID | Target Appl. | Machine | Sampling Period | # Observations |
|---|---|---|---|---|
| Dyfe_30_fx8 | Dyfesm | FX/8 | 30 s | 42 |
| Dyfe_30 | Dyfesm | FX/80 | 30 s | 61 |
| Dyfe_0 | Dyfesm | FX/80 | ~1.1 s | 35 |
| Flo_30 | Flo52 | FX/80 | 30 s | 49 |
| Flo_10 | Flo52 | FX/80 | 10 s | 41 |
| Flo_0 | Flo52 | FX/80 | ~1.4 s | 37 |
| Track_30 | Track | FX/80 | 30 s | 35 |
| Track_0 | Track | FX/80 | ~1.3 s | 30 |

system while the target application is executed. The standard deviation reported in the table quantifies the deviation among the observation means of an experiment. A large standard deviation indicates that the workload fluctuated greatly over the collection of the experiment.

The most common job was the type A cluster job. There were usually two or three of these jobs in the system while a target application executed. Type C cluster jobs were much less prevalent (except in the observations of Track_30). The CE and IP/CE jobs tended to be shorter in duration than cluster jobs, and, at a given time, there were usually only one or two in the system.

Table 5.3 summarizes additional system measurements which characterize the workload for each experiment. The values shown are averages for all observations of an experiment. The CE utilization is the average utilization of all eight CEs and IP utilization is the average utilization of all IPs. The %CLUSTERED is the percentage of time the CEs were physically clustered. Paging rate is the number of disk accesses per second and reconfiguration (Reconfig) rate is the number of times the CEs switched configurations each second.

Notice that there is, on average, little paging being done. The observations which contain the most paging were those taken on the Alliant FX/8. This is probably due to the smaller memory size of the machine. For the FX/80 experiments, observations in Flo52_30 contained the most paging,

Table 5.2
Job Queue Lengths (#)

| Experiment ID | Clustered Type (A) Job Q length | | Clustered Type (C) Job Q length | | CE Type (A,C) Job Q length | | IP/CE Job Q length | |
|---|---|---|---|---|---|---|---|---|
| | Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. |
| Dyfe_30_fx8 | 2.51 | 0.98 | 0.18 | 0.29 | 0.41 | 0.39 | 2.33 | 1.27 |
| Dyfe_30 | 2.57 | 1.46 | 0.52 | 0.68 | 0.84 | 0.71 | 0.91 | 0.63 |
| Dyfe_0 | 3.47 | 1.07 | 0.57 | 0.76 | 0.73 | 0.37 | 1.41 | 0.58 |
| Flo_30 | 3.38 | 1.72 | 1.14 | 1.27 | 0.78 | 0.64 | 0.96 | 0.78 |
| Flo_10 | 3.05 | 1.22 | 0.06 | 0.20 | 1.35 | 0.92 | 0.63 | 0.55 |
| Flo_0 | 3.67 | 1.98 | 0.30 | 0.47 | 1.02 | 0.86 | 1.23 | 1.06 |
| Track_30 | 3.63 | 1.21 | 1.32 | 1.22 | 0.90 | 0.56 | 1.37 | 0.75 |
| Track_0 | 3.20 | 0.90 | 0.24 | 0.34 | 0.64 | 0.51 | 1.11 | 0.84 |

Table 5.3
Workload Characteristics

| Experiment ID | CE Util. (%) | IP Util. (%) | % CLUSTERED (%) | Paging Rate (page/s) | Reconfig. Rate (recon./s) |
|---|---|---|---|---|---|
| Dyfe_30_fx8 | 0.83 | 0.07 | 71.9 | 0.959 | 0.441 |
| Dyfe_30 | 0.86 | 0.15 | 78.9 | 0.088 | 1.006 |
| Dyfe_0 | 0.78 | 0.31 | 61.1 | 0.027 | 1.660 |
| Flo_30 | 0.86 | 0.15 | 77.9 | 0.322 | 1.039 |
| Flo_10 | 0.81 | 0.27 | 71.5 | 0.011 | 1.360 |
| Flo_0 | 0.79 | 0.34 | 61.4 | 0.211 | 1.712 |
| Track_30 | 0.82 | 0.20 | 70.7 | 0.185 | 1.44 |
| Track_0 | 0.79 | 0.33 | 63.3 | 0.172 | 1.66 |

averaging one disk access every three seconds. Some individual observations contained a good deal of paging, but the majority of observations had no paging. For instance, for one observation in Track_30, the disk was accessed 11,143 times, for an average of 3.7 accesses a second, while 27 other observations in the experiment contained no paging. The data also show that the CEs were highly utilized throughout the experiments, while the IPs were uniformly under-utilized. This is probably due to the large proportion of parallel jobs— which can run only on the CEs— in the workload.

Figure 5.3 shows the frequency distributions of the completion times for the target applications in the real workloads. The graphs illustrate the large range of time spanned by the completion time of an application in real workloads. The completion time of Flo_52 was as long as 3229 s; this is more than 36 times longer than the application takes on a dedicated machine. The large variances are due to the two degrading components in multiprogramming environments: multiprogramming overhead and resource timesharing.

### 5.4. Multiprogramming and System Overheads

Measurements of system and multiprogramming overheads are presented in this section. System overhead is measured with the Q facility and is available for all processing resources (clustered CEs, detached CEs, and IPs). Multiprogramming overhead is estimated using the statistical sampling technique. It is determined only in the parallel environment (the CEs while clustered). Section 5.4.1 will

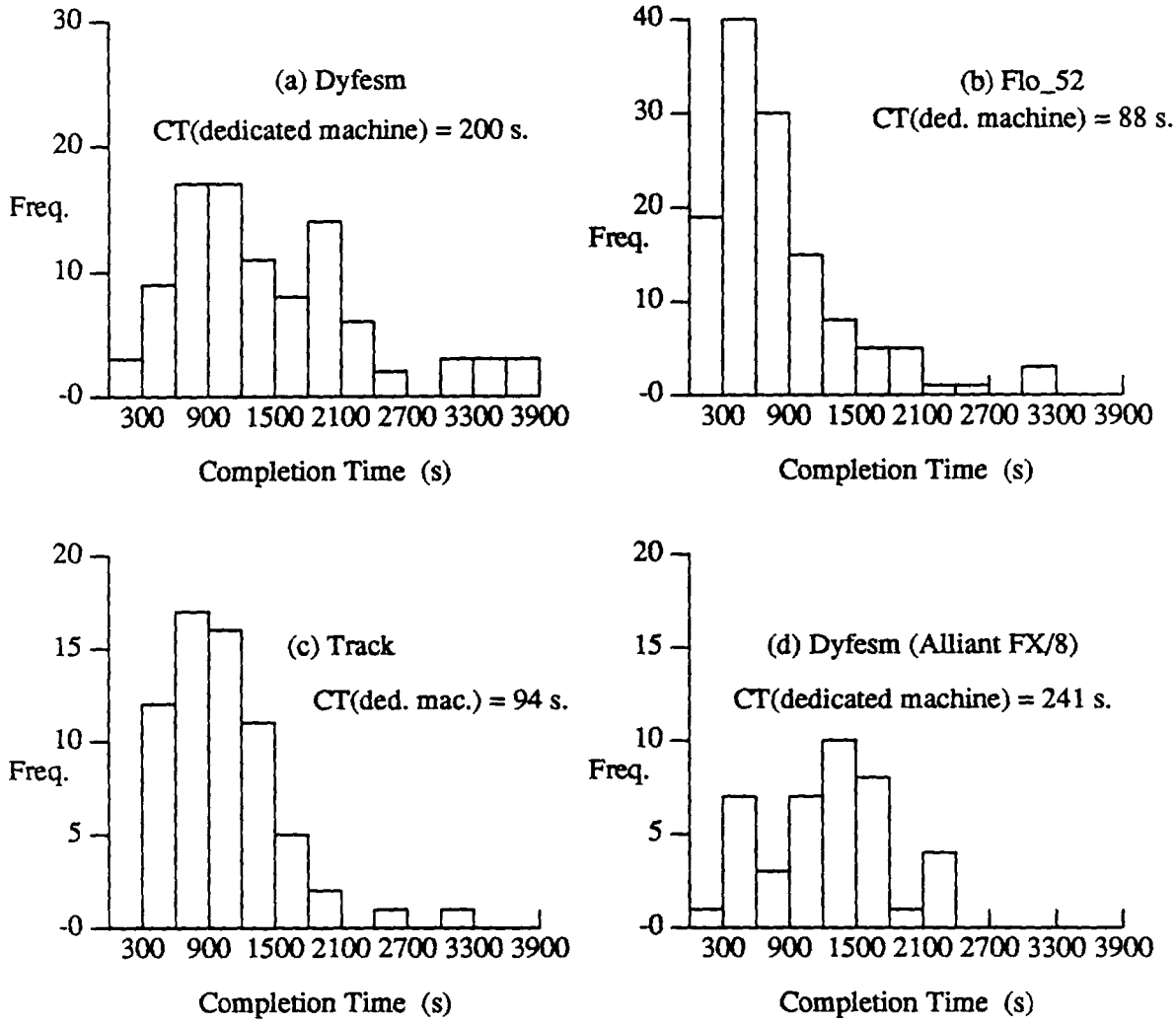



Figure 5.3 Completion Times of Target Applications

look in depth at the overheads in the parallel environment (both multiprogramming and system). Section 5.4.2 presents system overhead measurements for all processors and decomposes it into three exclusive categories: spin lock, interrupt handling, and general system work.

#### 5.4.1. Overheads in parallel environment

Table 5.4 shows the mean and standard deviation of MPO, total system overhead, and application completion time for each experiment. The measurements quantify the percentage of parallel processing time (clustered CE time) consumed by the specified overhead. The last row of the table shows

the averaged results for the FX/80. As expected, the mean MPO is less than the mean system overhead for all experiments, because on a gang scheduled machine such as the Alliant, MPO is a subset of total system overhead. The MPO standard deviation is slightly greater than that of the total system overhead. This is attributable to both inherent variance in real workload the MPO and variance caused by estimation noise. The error in estimating MPO will be discussed in detail in Section 5.5.

The table shows that well over half of the total system work executed by clustered CEs is due to MPO. More specifically, on average, 16% of the processing power available to parallel applications is consumed by MPO. This is a substantial amount of processing power, much larger than the desired 10% [39].

The standard deviation is fairly large for both multiprogramming and system overheads (13.04% and 12.92%, respectively). This indicates that both of these measurements can take a wide range of values. For instance, one observation had a system overhead measurement of 82.1%, while another had a measurement of just 5.5%. These are extremes; the 5th and 95th percentiles for this value are 7.8% and 49.8%, respectively. The 25th and 75th percentiles are 12.8% and 30.5%.

The MPO was estimated to be as high as 63.9%, but this estimate was probably skewed by noise. Noise also caused some MPO estimates to be less than zero (this never occurred while sampling as

Table 5.4
Overhead Measurements (% of Parallel Processor Time)

| Experiment ID | MP Overhead | | System Overhead | | Completion Time | |
|---|---|---|---|---|---|---|
| | Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. |
| Dyfe_30_fx8 | 17.92 | 14.18 | 26.93 | 14.24 | 1313.24 | 804.04 |
| Dyfe_30 | 13.68 | 14.76 | 17.47 | 12.02 | 1122.53 | 635.37 |
| Dyfe_0 | 18.32 | 8.25 | 29.38 | 9.12 | 2103.52 | 877.13 |
| Flo_30 | 16.38 | 16.35 | 27.17 | 13.45 | 933.6 | 672.51 |
| Flo_10 | 12.27 | 10.89 | 12.85 | 4.07 | 528.80 | 246.53 |
| Flo_0 | 21.77 | 9.74 | 27.18 | 10.01 | 933.34 | 720.62 |
| Track_30 | 14.17 | 16.15 | 26.03 | 17.42 | 1222.38 | 549.08 |
| Track_0 | 17.06 | 8.36 | 30.43 | 8.78 | 788.13 | 345.48 |
| Totals | 16.05 | 13.04 | 23.55 | 12.92 | | |

often as possible). This is to be expected though, given the size of the noise terms (Section 5.5). The 25th and 75th percentiles for the MPO measurement are 10.69% and 22.14%, respectively. In other words, the majority of workloads lose 10 to 23% of their processing power to multiprogramming overhead.

For the observations of the experiments Flo_0, Dyfe_0, and Track_0, the combined frequency distributions for both multiprogramming and system overheads are given in Figure 5.4. Notice that the tails on both distributions extend well past the 33% mark. For system overhead, one-third of the observations were greater than 33%, and for MPO, one-fifth of the observations were greater than 33%. This further illustrates the substantial performance degradation caused by multiprogramming and system overheads in real workloads.

The mean MPO found on the Alliant FX/8 (17.9%) was close to that found on the Alliant FX/80 (16.1%). Assuming the workloads to be similar, it can be inferred that the cost of multiprogramming is similar for the two machines.

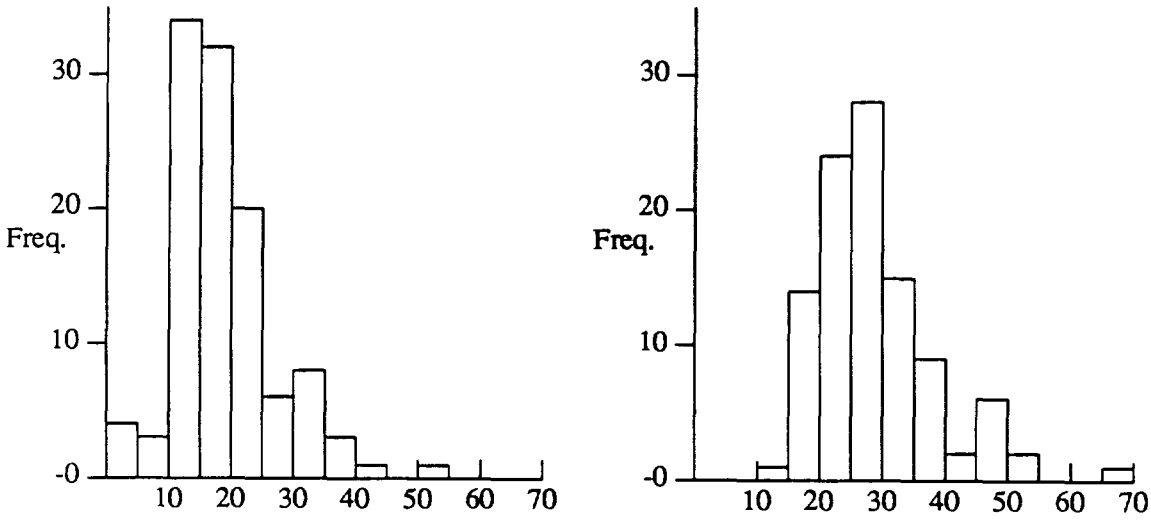



Figure 5.4 Multiprogramming and System Overheads: Distributions

### 5.4.2. System overhead components: all processors

System and multiprogramming measurements presented thus far pertain to overheads incurred while the CES were clustered or to overheads directly affecting parallel job execution. In this subsection, system overhead executed on the other processors is quantified (measured with the Q facility). The MPO is not available on these processors. In addition, for each processing resource (clustered CEs, detached CEs, and IPs), the total system overhead is divided into three mutually exclusive subcategories: kernel lock spins (SPIN), interrupt handling (INT), and general system work (SYS). Figure 5.5 illustrates this taxonomy for a single resource on the FX/80.

The SPIN parameter quantifies the time a processor spends waiting for one of the numerous kernel mutual exclusion locks to be released. The locks are implemented to share data or protect critical sections in the single copy of the operating system. The INT category clocks the time the processor spends handling interrupts (such as DMA or device interrupts). The SYS parameter is a catch-all category made up of all the other system work such as system calls.

Figure 5.6 summarizes the results for the 288 observations collected on the Alliant FX/80. The figure shows the 25th percentile, the 75th percentile, and the mean for each of the overhead subcategories (SPIN, INT, and SYS) on each of the processing resources (clustered CEs, detached CEs, and IPs). For instance, the mean percentage of time spent by IP 6 spinning on locks (SPIN) was 9.7%.

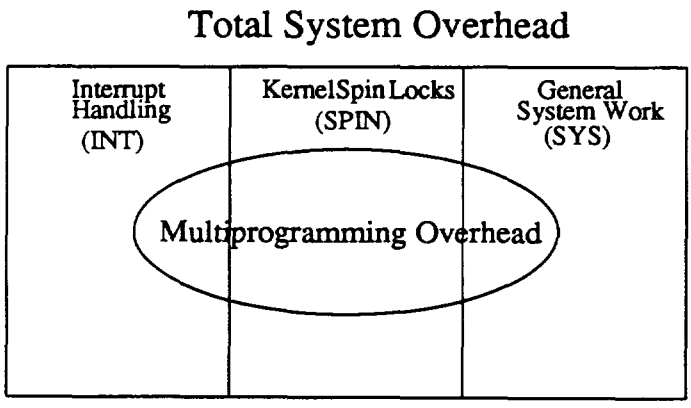



Figure 5.5 Alliant Overhead Venn Diagram

In addition, half of the observations of IP 6 SPIN time consumed between 2.5% and 13.7% (the 25th and 75th percentile) of the total IP 6 processing time. Note that the measurements for overhead executed on the clustered CEs are given as a percentage of clustered time (not total time), and, correspondingly, overheads for the detached CEs are presented as a percentage of detached time.

Figure 5.6 shows that for all processing resources (except IP 1) INT was a minor component of total system overhead. In addition, the range (shaded region) of the INT parameter was small. These two facts suggest that interrupt handling was not a major cause of performance degradation and was fairly predictable.

On the other hand, the figure shows that kernel lock spinning was a major component of system overhead for all processing resources. For instance, well over one-third of the total system overhead executed on the clustered CEs was attributable to kernel lock spinning. Kernel lock spinning is of particular interest to this study because contention for the kernel is clearly a manifestation of multiprogramming. When the target applications were executed alone on a dedicated machine, kernel lock spinning was minimal. Because kernel lock spinning is such a major component of system overhead in real workloads, and because it completely degrades the processor, kernel lock spinning is a clear target for improving system performance.

### 5.5. The Sampling Period

The real workload MPO estimation technique introduced in Section 5.2 requires pertinent workload parameters to be periodically sampled. The choice of the sampling period must balance the opposing goals of reducing the intrusiveness of the sampling software while accurately capturing the workload on the system (Figure 5.1). For the Alliant study three different sampling periods were used: ~1.3 s, 10 s, and 30 s. In this section, the effect of the sampling period length on the accuracy of the estimation of $CT_{w/o\ MPO}$ is investigated by determining the noise term associated with each sampling period.

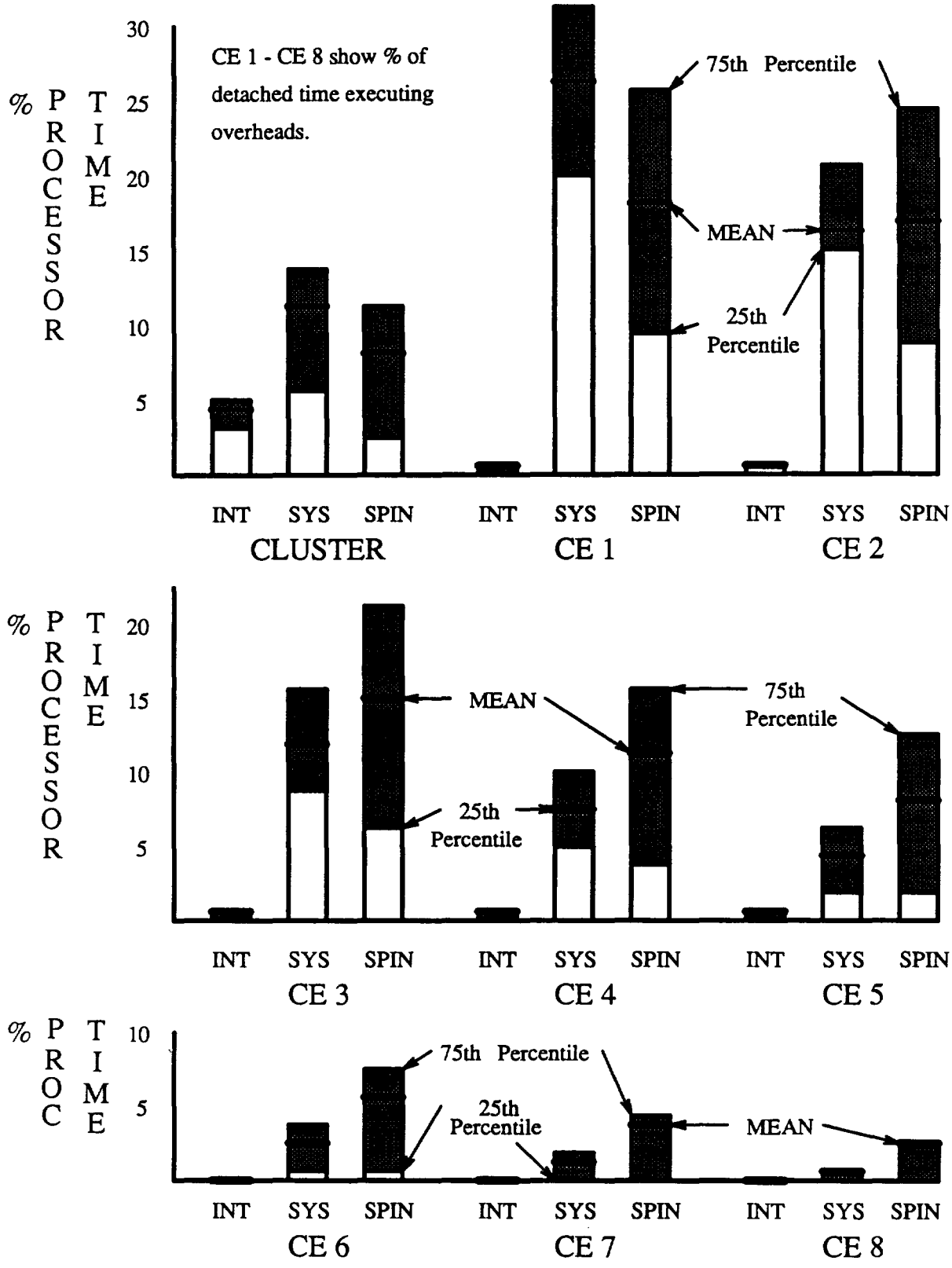



Figure 5.6 CE System Overhead Components

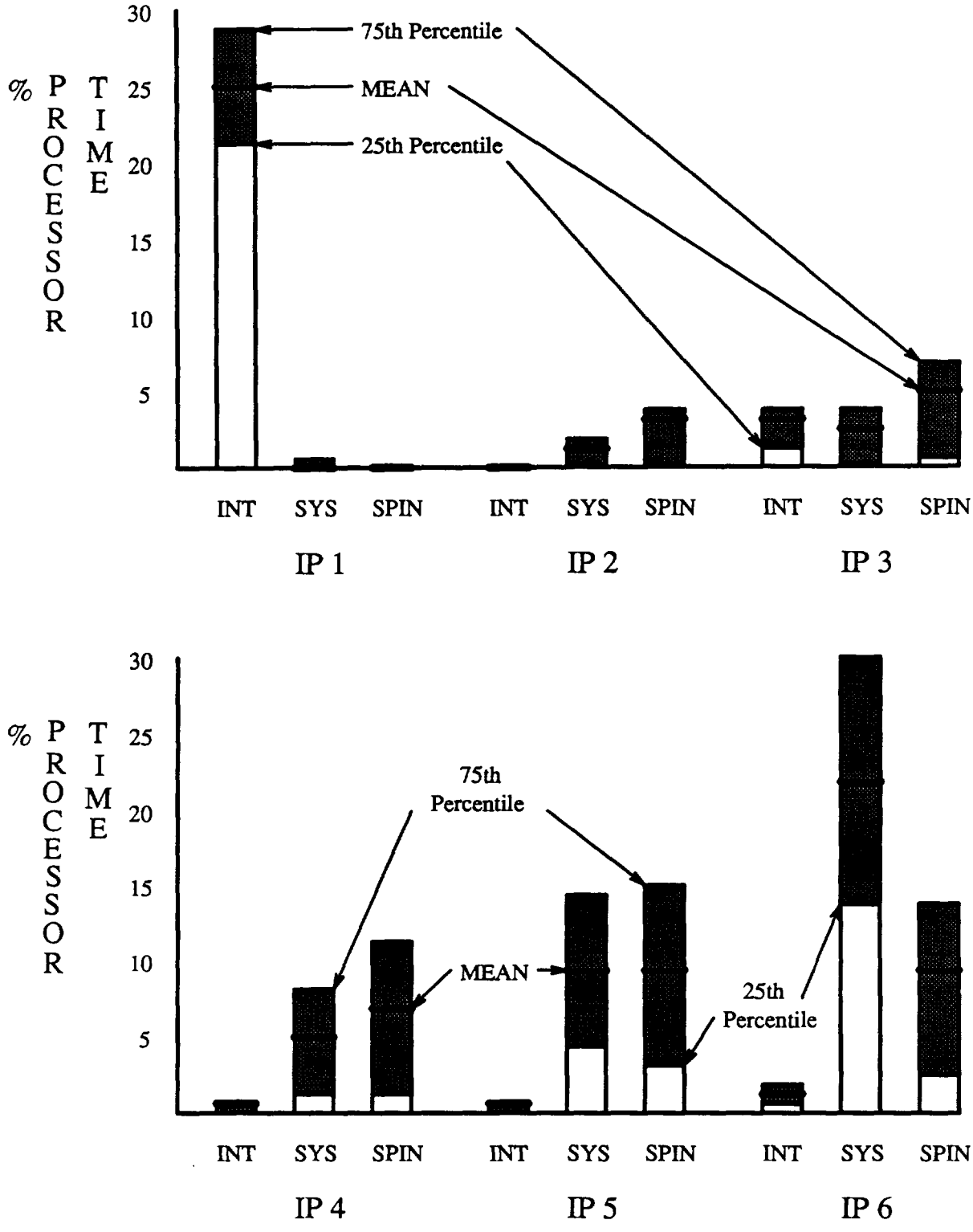



Figure 5.6 IP System Overhead Components

The 30 s sampling period reflects the workload least accurately, but is also least likely to affect the workload. Conversely, sampling frequently may represent the workload well but also perturb it. Due to physical limitations, the smallest sampling period possible was approximately 1.3 s. This was achieved by requeueing the measuring facility as soon as a sample was collected. To reduce the interference with other CE jobs, the sampling facility was executed on the IPs. Unfortunately, the shell program which managed the sampling facility could not be completely regulated to the IPs. Therefore, whenever the workload was not being sampled, there was an extra IP/CE job queued in the system. The overall effect was that the workload was slightly modified by the measuring facility. To account for this, Equation (5.3) was slightly modified to ensure that the MPO estimate remains correct.

To determine the accuracy of the estimate of completion time without the multiprogramming overhead, $\hat{CT}_{w/o\ MPO}$, obtained with each sampling period, the estimate is modeled as the true but unobtainable measure, $CT_{w/o\ MPO}$, plus a Gaussian white noise error term, Z, (Equation (5.5)).[2] The noise term accounts for differences between the true workload and the one represented by the samples. This difference is illustrated by the solid and dotted lines of Figure 5.1. The variance of the noise term (the mean is zero) is a direct reflection of the accuracy of the sampling period. The higher the variance, the larger the possible error in estimating $CT_{w/o\ MPO}$.

$$\hat{CT}_{w/o\ MPO} = CT_{w/o\ MPO} + Z \tag{5.5}$$

This problem of estimating a signal from a signal with noise is a classic problem in signal processing. For the current problem though, difficulties arise in the construction of the filters because the signal and the noise may have the same frequencies. In addition, even if the noise attributed to each queue signal could be accurately obtained, determining the noise in $\hat{CT}_{w/o\ MPO}$ is not straightforward. Because of these difficulties, signal processing methods were abandoned and two statistical techniques were developed to estimate the standard deviation of the noise term, and hence the accuracy of the estimation.

---

[2]For simplicity, all other sections have and will refer to the estimate $\hat{CT}_{w/o\ MPO}$ as $CT_{w/o\ MPO}$.

The first technique consists of determining $\hat{CT}_{w/o\ MPO}$ a number of times in a slowly changing workload (i.e., a workload in which the number and types of jobs remain fairly constant). In other words, for a workload which did not change drastically, a target application was executed a number of times sequentially. Each execution of the target application produced a value for $\hat{CT}_{w/o\ MPO}$. It is assumed that the $CT_{w/o\ MPO}$ (not the estimate) over the entire period of the workload is constant because the workload itself is nearly constant. Therefore, the deviation in the multiple estimates of $CT_{w/o\ MPO}$ can be completely attributed to the noise term, and the standard deviation of the multiple estimates of $CT_{w/o\ MPO}$ can be used as an estimate of the standard deviation of Z.

The above procedure was performed using a number of slowly changing workloads for each of the three sampling periods. Each different slowly changing workload produced multiple estimates of $CT_{w/o\ MPO}$ and a single estimate of Z. The standard deviation for the noise term of a particular sampling period was then determined using Equation (5.6). In the equation, $n_i$ is the number of times $\hat{CT}_{w/o\ MPO}$ was determined for the $i^{th}$ workload, #WL is the number of workloads used, and $\hat{\sigma}_i^2$ is the estimate of Z's variance determined from the $i^{th}$ workload.

$$\hat{\sigma}^2_{overall} = \frac{\sum_{i=1}^{\#WL} n_i \hat{\sigma} subi^2}{\sum_{i=1}^{\#WL} n_i} \tag{5.6}$$

Table 5.5 shows, for each sampling period, the number of workloads used, the number of times $\hat{CT}_{w/o\ MPO}$ was determined, and the estimate of the noise term standard deviation. Notice that the larger sampling periods have larger noise standard deviations.

Table 5.5
Noise Term Variances: Technique One

| Sampling Period (s) | # Work Loads Used | # Samples Taken | Noise Term Std. Dev. (s) |
|---|---|---|---|
| ~1.3 | 4 | 15 | 4.82 |
| 10 | 4 | 14 | 18.80 |
| 30 | 4 | 20 | 28.22 |

The second technique used to estimate the standard deviation of the noise term assumed that $\hat{CT}_{w/o\ MPO}$ determined from the data collected while sampling as fast as possible was the actual $CT_{w/o\ MPO}$ (i.e., there was no noise with the fastest sampling). These data were then pruned to reflect the data that would be collected if a 10 or 30 s sampling were used. For instance, if sampling as fast as possible actually resulted in sampling every 2 s, then every fifth sample would be kept to approximate sampling with a 10 s interval. The $\hat{CT}_{w/o\ MPO}$ was then calculated with the pruned data. The difference between the $\hat{CT}_{w/o\ MPO}$ determined from all of the data and the $\hat{CT}_{w/o\ MPO}$ determined from the pruned data is a single measurement of the noise term. Using this technique, a large number of individual noise terms were determined and used to estimate the distribution of the noise term for each sampling period.

Table 5.6 provides the number of samples analyzed, the estimated mean value of Z, and the estimated standard deviation of Z for the two sampling periods.

The estimates of noise term standard deviation using technique two confirm the estimates using the first technique. The estimates of the noise term means are not statistically different from zero. This complies with the model proposed in Equation (5.5). How the noise term is reflected in the MPO estimate is dependent on the actual completion time of the target application (Equation (5.1)).

Note that the noise term associated with 30 s sampling may at times be large and thus $\hat{CT}_{w/o\ MPO}$ and the MPO estimate may be highly skewed. However, This does not mean that results obtained with a 30 s sampling period are useless. The noise term has a mean of zero (substantiated by the data), so that if the MPO estimate is computed for a number of runs, the mean of these will converge to the

Table 5.6
Noise Term Variances: Technique Two

| Sampling Period (s) | # Samples Used Mean | Noise Term | Noise Term Std. Dev. |
|---|---|---|---|
| 10 | 102 | 0.25 | 15.42 |
| 30 | 102 | 6.94 | 27.66 |

actual MPO mean. Therefore, the means provided in Table 5.4 for sampling periods of 30 s are reliable. The individual measurements may be suspect though.

To summarize, for large sampling periods, many test runs must be collected and averaged before the results are meaningful. For small sampling periods, estimates from single test runs have meaning.

### 5.6. Overhead/Workload Relations

The previous results show that both multiprogramming and system overhead are dependent on the workload. In this section, statistical techniques will identify underlying relations between overhead and workload characteristics. The relation among overhead measurements will be investigated first. The goal is to understand the causes of overhead and how the overhead affects the completion time of parallel applications.

#### 5.6.1. Overhead — overhead correlations

The fraction of total system overhead which is attributable to kernel lock spinning and interrupt handling has been directly measured (Figure 5.5). The fraction of the MPO attributable to these overhead components cannot be measured. To gauge the impact of these components on MPO, as well as the relationships among the overhead components, correlation analysis was conducted with the real workload overhead measurements.

Correlation coefficients between all pairs of overhead measurements (for example, IP 2 SPIN time and CE 4 INT time) were calculated. The relationships between multiprogramming overhead and other system overheads are discussed first. The MPO was moderately correlated with cluster total system overhead (0.45), IP 1 INT time (0.51), and cluster SPIN time (0.47). All other correlations between MPO and overhead parameters were negligible. The correlation with total system overhead is expected because MPO is one of its major components. The correlation with IP 1 INT time is due to general increases in system activity and not suggestive of anything crucial. The correlation between

cluster SPIN time and MPO is suggestive. It indicates that kernel lock spinning (which is not found on a uniprocessor) is a major component of MPO on a multiprocessor. Due to the prominence of kernel lock spinning (Figure 5.5), it is easily deduced that multiprogramming is more costly to implement on a multiprocessor than a uniprocessor. This indicates that measurements of uniprocessor multiprogramming overhead will not provide sufficient understanding of multiprogramming on a multiprocessor. Also, measurements of the individual components of MPO (such as context switch time) do not provide adequate understanding of MPO. Real workload measurements are the only means to acquire the whole picture of multiprogramming overhead on a multiprocessor.

Correlation analysis of kernel lock spinning among the processing resources reveals another interesting fact. The amount of time each processor spends spinning is highly correlated with each other (0.65 - 0.95). This suggests that there are times at which access to critical sections of the kernel degrades not just one, but most processors of the machine. Clearly kernel lock spinning is a major, unique culprit of MPO on a multiprocessor.

### 5.6.2. Overhead — workload characteristics correlations

A representative sample of correlation coefficients between overhead measurements and workload parameters is given in Table 5.7. The table shows workload correlations with overheads from the clustered CEs, two detached CEs (one heavily and one normally loaded), and an IP.

Moderate correlations were found between processor utilizations and cluster MPO, and between processor utilizations and cluster system overhead. This suggests that when the system is heavily loaded, a larger percentage of parallel processing time is consumed by overhead. Paging shows little correlation with overhead since little is captured by the experiments. The CE jobs were also not prevalent, and, correspondingly, their correlations with overhead were small.

More interesting are the correlations between the overheads and the number of IP/CE jobs in the system. It was found that an increase in the number of IP/CE jobs was generally accompanied by an

Table 5.7
Correlation Coefficients

| Parameters | Cluster Job Queue Length | CE Job Queue Length | IP/CE Job Queue Length | CE Util. | IP Util. | Paging |
|---|---|---|---|---|---|---|
| MPO | 0.10 | 0.22 | 0.43 | 0.48 | 0.37 | 0.31 |
| Cluster System Overhead | 0.33 | -0.13 | 0.46 | 0.62 | 0.47 | 0.44 |
| Cluster SPIN | 0.29 | -0.01 | 0.57 | -0.20 | 0.80 | 0.22 |
| Cluster SYS | 0.30 | -0.23 | 0.04 | 0.21 | 0.09 | 0.11 |
| Cluster INT | 0.17 | 0.29 | 0.20 | -0.13 | 0.24 | 0.35 |
| CE 1 SPIN | 0.06 | -0.13 | 0.53 | 0.08 | 0.76 | 0.19 |
| CE 1 SYS | -0.11 | -0.44 | -0.02 | 0.14 | 0.28 | -0.09 |
| CE 1 INT | -0.10 | 0.02 | 0.04 | 0.35 | -0.16 | 0.16 |
| CE 5 SPIN | 0.12 | 0.09 | 0.63 | -0.03 | 0.76 | 0.20 |
| CE 5 SYS | 0.14 | 0.08 | 0.64 | -0.17 | 0.80 | 0.09 |
| CE 5 INT | 0.14 | 0.76 | 0.43 | -0.25 | 0.34 | 0.24 |
| IP 5 SPIN | 0.10 | 0.02 | 0.60 | -0.21 | 0.92 | 0.20 |
| IP 5 SYS | 0.09 | -0.08 | 0.54 | -0.40 | 0.86 | 0.10 |
| IP 5 INT | 0.02 | 0.13 | 0.25 | -0.13 | 0.36 | 0.05 |

increase in processor overheads. While this may be expected for the IPs or detached CEs, it is disconcerting to see the correlation between the number of IP/CE jobs and clustered CE overhead (both multiprogramming and system). A quick survey of the table indicates that the overhead component most responsible for this correlation was kernel lock spinning. It is postulated that through increased kernel contention, IP/CE jobs increase overhead on the clustered CEs and degrade the performance of parallel applications.

Surprisingly, the correlation between the number of parallel (cluster) jobs and MPO present in the parallel environment is negligible. This implies that when more parallel jobs are multiprogrammed, the MPO does not necessarily increase.

To investigate this relationship further, the Hoeffding test of independence was conducted on these parameters. The Hoeffding independence test is a nonparametric test which uses observations from two random variables to test the hypothesis that the two random variables are independent. Equation (5.7) illustrates the hypothesis for the random variables X and Y.

$$H_0 : P(X \leq x \text{ and } Y \leq y) = P(X \leq x) \times P(Y \leq y) \tag{5.7}$$

Using this test, the hypothesis that the MPO and the number of parallel jobs in the system are independent could not be rejected at any reasonable level (p-value = 0.2). On the other hand, the same test soundly rejected ($\alpha = 0.01$) the hypothesis that total system overhead is independent of the number of parallel jobs in the system. To summarize, while the number of parallel jobs may increase total system overhead, there is no statistical evidence to indicate that MPO is dependent on the number of parallel jobs in the system.

This result is probably due to the nature of MPO and the types of jobs in the workloads studied. A major component of multiprogramming overhead is context switching which consumes nearly the same amount of time regardless of the number of jobs in the queue. In addition, for most of the workloads tested, the number of parallel jobs on the system ranged from two to four. Given this limited number and the memory requirements of the applications, the memory was large enough to effectively eliminate paging. For these reasons, MPO was less sensitive than total system overhead to the number of parallel jobs in the system.

To summarize, correlation analysis determined that for the workloads studied, the number of parallel jobs multiprogrammed does not greatly affect the amount of MPO in the parallel environment. However, through increased kernel contention, serial jobs executing on peripheral processors have a great effect on the MPO.

# CHAPTER 6.

# MULTIPROGRAMMING AND SYSTEM OVERHEADS: CEDAR

Workloads encountered during Cedar's normal operation are quite different from those found on the Alliant. There are two reasons for this. First, the purpose of each machine is different. Cedar is concerned mainly with executing parallel programs efficiently, while the Alliant tries to provide speed up for parallel applications and high throughput for serial and interactive jobs. In *real* Cedar workloads there will be few (if any) serial jobs, while, as seen, there are many in Alliant workloads. The second reason for the different workload characteristics is that Cedar is a new experimental prototype, while Alliant is a reliable, established mini-supercomputer. Workloads on Cedar are more experimental. Because of this, users tend to avoid using Cedar in multiuser mode, opting for single-user mode in which they can monitor the execution of their application without outside contentions and interactions.

For the reasons listed above, multiuser workloads encountered during normal Cedar operation were not interesting for monitoring or modeling purposes. For the purposes of this study a number of multiuser workloads were constructed. The workloads were constructed using the target applications introduced in Chapter 3 (Table 3.4); they are real in the sense that they consist of actual applications. No serial jobs were included in the workloads because it was felt that Cedar was not constructed for this type of work.

The workloads were constructed to study the performance effects of multiuser workloads executing on Cedar. In the first section of this chapter, characteristics of the target applications used to construct the workloads are detailed. Section 6.2 presents system overhead measurements for the constructed workloads. The multiprogramming overhead incurred by certain target applications in these workloads is also given. It is found that multiprogramming greatly degrades the performance of some applications, while for others, multiprogramming actually benefits total system throughput by I/O overlapping. Section 6.3 presents empirical experiments which highlight these effects and identifies

application characteristics which cause them. It is found that synchronization overhead for SDOALL loops is exacerbated in multiprogrammed workloads which can lead to highly degraded performance.

### 6.1. Cedar Target Applications

The test applications used to construct the workloads were introduced in Section 3.3. In this section their basic characteristics will be studied in greater detail. For each application, the resident page size as a function of time and the processor utilization will be presented.

#### 6.1.1. TFS

The TFS application is the Perfect benchmark Flo52 written for Cedar using loop concurrency (SDOALL) constructs. The application alternately switches between sections of code that are run on a single cluster and loops that are executed on all four clusters through the use of helper tasks. The resident page set size for each type of memory as a function of application execution is shown in Figure 6.1. These results were obtained by periodically interrupting the execution of the program and using the *getvmetc* system call. The size of each page is 4 K. In the figure, the measurements of

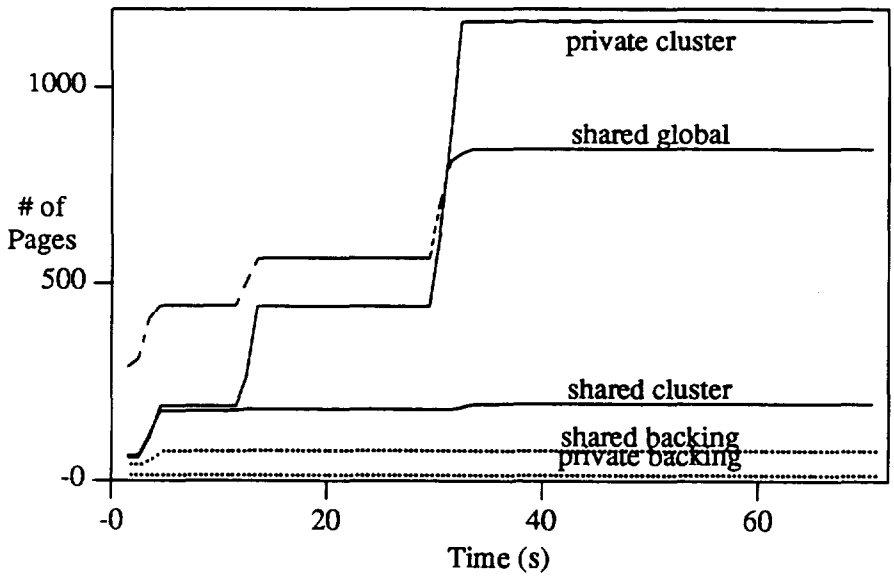



Figure 6.1 Resident Page Set Sizes: TFS

cluster memory usage are for all four cluster memories combined. Shared backing and private backing pages both reside in global memory. They are copies of certain pages in shared and private memory, respectively.

The figure shows that the most extensively used memory is private cluster memory, with shared global memory being used second most frequently. Private global memory is never used by this application. The figure shows that memory usage occurs in three waves, at the beginning of the program, at 15 s, and at 35 s. Correspondingly, examination of the application's algorithm, reveals that the application itself consists of three phases of execution.

Cluster utilization as a function of application execution is pictured in Figure 6.2. Only the first 5 s of execution are shown. The Y-axis shows the number of clusters employed to execute the application. Notice that only two situations exist. The application uses either just one cluster or all four clusters. The one-cluster situation corresponds to serial code or loops with dependency which must be executed on a single cluster. The four-cluster situation corresponds to SDOALL loops. For instance, from program inception until approximately 1 s, TFS executes on a single cluster. At this point an SDOALL loop of approximately 10 ms is executed.

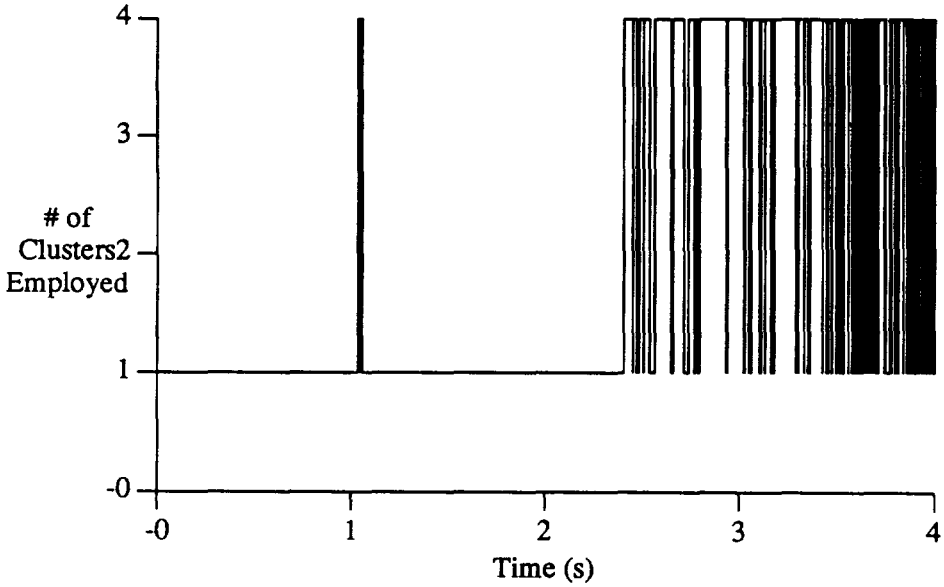



Figure 6.2 TFS Cluster Utilization

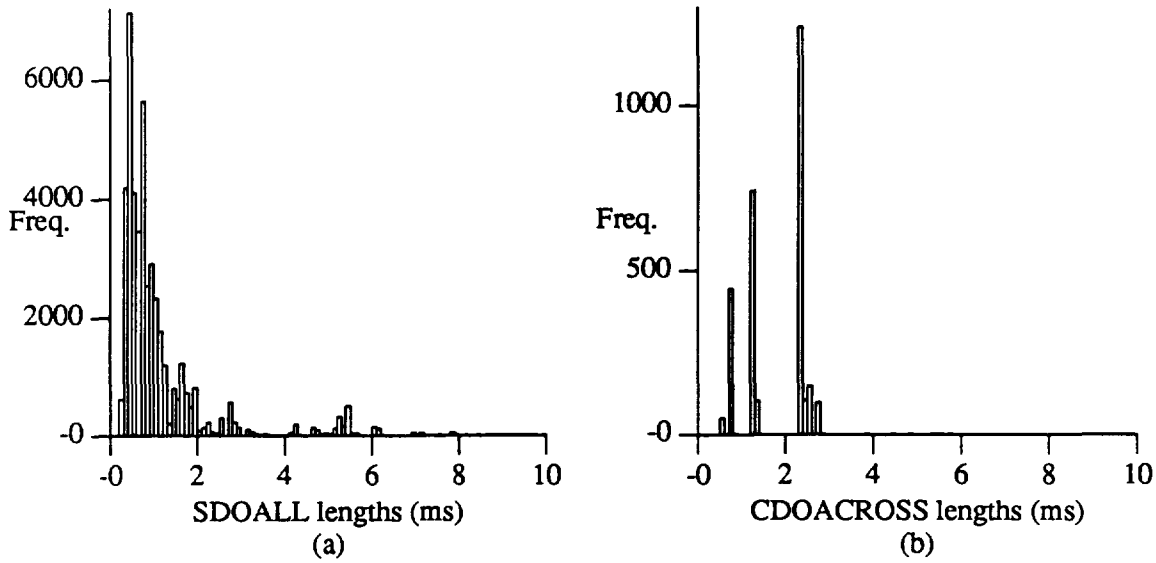



Figure 6.3 Length of SDOALL and CDOACROSS Loops: TFS

Figure 6.3(a) shows the frequency distribution of execution times for SDOALL loops of TFS on a dedicated machine. For the sake of clarity, the figure shows only those loops which execute in under 10 ms. In general, TFS is made up of very short SDOALL loops. The average length is only 1.44 ms, and there is a large mass of SDOALL execution times between 0.5 and 1 ms. However, even with the short lengths, SDOALL loops account for 75.5% of the execution time of TFS on a dedicated machine.

Figure 6.3(b) shows the frequency distribution of execution times for CDOACROSS loops. Again, only those loops which execute in less than 10 ms are shown. The CDOACROSS loops contain dependencies and are executed on eight processors of a single cluster. There are fewer CDOACROSS loops in TFS than there are SDOALLs, but the average length is slightly longer at 1.86 ms.

### 6.1.2. ARC and MCP

Applications ARC and MCP are written with coarse-grained task parallelism. The ARC application begins execution by spawning three tasks and remains a four-task application throughout its entire

execution. The MCP application executes on a single cluster for 77.5 s and then spawns three tasks which execute with the main task until completion approximately 40 s later.

The resident page set sizes for ARC and MCP are given in Figures 6.4 and 6.5, respectively. Both applications use private cluster memory more extensively than shared global or shared cluster. Like TFS, neither application uses private global. The sharp rise in private cluster memory usage seen at approximately 80 s for MCP corresponds to the spawning of the three tasks. At this point, the private cluster pages of the main task are copied to the other cluster memories. In this way, each task has its own copy of private cluster data. Notice that ARC's sharp rise in private cluster usage also corresponds to the spawning of tasks.

The figures show that of the three applications, ARC uses the least of all types of memory. Focusing solely on Figure 6.4, it is seen that ARC uses private cluster memory almost exclusively. The most memory intensive application is MCP. Shared global memory is heavily used with a noticeable increase at 80 s when the tasks are spawned. This increase cannot be explained by the mechanics of the task spawning as was the case for the private cluster memory.

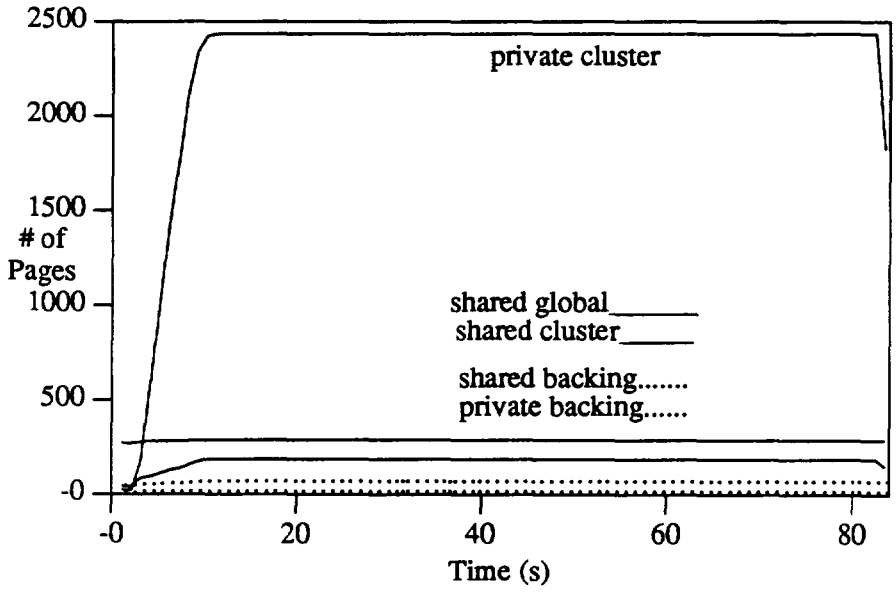



Figure 6.4 Resident Page Set Sizes: ARC

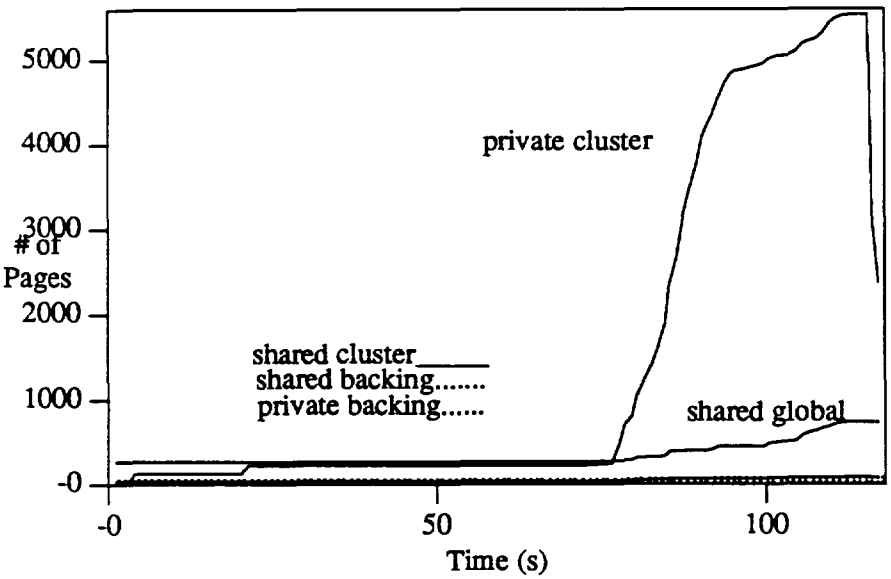



Figure 6.5 Resident Page Set Sizes: MCP

**6.2. Multiprogramming and System Overhead**

Multiuser workloads were created by executing different combinations of the above programs. The workloads were real in the sense that real applications were used. Detached (serial) jobs were not included because Cedar is not the proper machine to run such jobs. The loop and task dummy jobs introduced in Section 4.2 were used in some of the workloads to control the level (number of jobs at a given time) of multiprogramming. Recall that a loop dummy job is an endless job consisting of a large SDOALL loop. The loop spreads CDOALL iterations across all clusters of Cedar. The task dummy job is an endless job which spawns a task for each cluster. Each task performs a CDOALL which uses all eight processors of the cluster. Therefore, both dummy jobs continuously utilize all 32 processors of Cedar. Also, both dummy jobs use a minimum of memory and create a minimum of system overhead.

This section will summarize the workloads created and present measured system overheads for each. The multiprogramming overhead incurred by a target application executing in each of the workloads will also be presented. Unlike the previous chapters, the MPO presented is not necessarily a measure of processor time consumed. This is because the target applications are not computationally

bound (as they were in the Alliant case). Benefits of multiprogramming such as I/O overlapping are expressed in the MPO measurements. Also, MPO in this section is not a subset of system overhead. This is because gang scheduling is not used, and multiprogramming effects may be incurred by user-level spinning. The workloads used to quantify the lower bound on multiprogramming overhead are included in this section for a point of reference.

Table 6.1 summarizes the workloads created and the measurements taken. The first column of the table provides an identification number for easy reference. The second column lists the target application, the application for which the effects of multiprogramming are gauged. The third column lists the applications (in addition to the target application) that make up the specific workload. The fourth column lists the completion time of the target application in the given workload. Following this the percentage of time each cluster is executing system overhead on each cluster is given. The final column is the important one; it lists the multiprogramming overhead incurred by the target application. For instance, Experiment 7c consists of executing TFS on a system with three other copies of TFS executing simultaneously. The completion time of TFS in this environment was 673 s, yielding an MPO of 56.3%. The table also shows that 12% of the processor time of cluster three was spent executing system overhead, while 11% of the other cluster's time was spent executing system overhead. As in all controlled experiments of this thesis, the measurements presented are an average of multiple runs.

Because the workloads were constructed solely from parallel jobs, the CEs on all clusters remained in the clustered configuration at all times. The system overhead for each cluster was measured using the Q facility. A separate log is kept in the kernel memory of each cluster and accessed with a local, independent version of Q. Synchronization of the Q facilities was roughly accomplished using UNIX remote shells. The completion time of the target application executing in the real workloads was measured using the hybrid P3S monitor.

Table 6.1
Multiprogramming and System Overheads of Multijob Workloads

| Exp. ID. | Target Appl. | Workload Description | CT (s) | System Overhead (%) | | | | MPO (%) |
|---|---|---|---|---|---|---|---|---|
| | | | | C1 | C2 | C3 | C4 | |
| 1a | LOOP_CON | 1 loop dummy | 98.6 | 9 | 9 | 10 | 9 | 10.7 |
| 1b | LOOP_CON | 2 loop dummy | 148.5 | 9 | 9 | 10 | 9 | 11.1 |
| 1c | LOOP_CON | 3 loop dummy | 199.4 | 10 | 10 | 10 | 10 | 11.7 |
| 1d | LOOP_CON | 4 loop dummy | 249.5 | 10 | 10 | 10 | 10 | 11.8 |
| 1e | LOOP_CON | 5 loop dummy | 300.0 | 9 | 9 | 10 | 9 | 12.2 |
| 2a | LOOP_CON | 1 task dummy | 98.9 | 9 | 9 | 10 | 9 | 10.7 |
| 2b | LOOP_CON | 2 task dummy | 148.4 | 9 | 9 | 9 | 9 | 11.1 |
| 2c | LOOP_CON | 3 task dummy | 199.5 | 9 | 9 | 10 | 9 | 11.7 |
| 2d | LOOP_CON | 4 task dummy | 249.3 | 10 | 10 | 10 | 10 | 11.8 |
| 2e | LOOP_CON | 5 task dummy | 300.8 | 10 | 10 | 10 | 9 | 12.2 |
| 3a | TASK_CON | 1 loop dummy | 119.3 | 9 | 9 | 9 | 9 | 9.9 |
| 3b | TASK_CON | 2 loop dummy | 179.0 | 9 | 9 | 9 | 9 | 10.2 |
| 3c | TASK_CON | 3 loop dummy | 239.2 | 10 | 10 | 11 | 11 | 10.3 |
| 3d | TASK_CON | 4 loop dummy | 300.5 | 10 | 9 | 10 | 10 | 10.6 |
| 3e | TASK_CON | 5 loop dummy | 363.7 | 10 | 9 | 10 | 9 | 11.6 |
| 4a | TASK_CON | 1 task dummy | 119.7 | 9 | 9 | 9 | 9 | 9.9 |
| 4b | TASK_CON | 2 task dummy | 180.0 | 9 | 9 | 9 | 9 | 10.2 |
| 4c | TASK_CON | 3 task dummy | 240.8 | 9 | 9 | 9 | 9 | 10.3 |
| 4d | TASK_CON | 4 task dummy | 302.1 | 9 | 9 | 10 | 9 | 10.6 |
| 4e | TASK_CON | 5 task dummy | 365.1 | 9 | 9 | 9 | 9 | 11.6 |
| 5a | TFS | 1 loop dummy | 301 | 9 | 9 | 9 | 9 | 51.2 |
| 5b | TFS | 2 loop dummy | 450 | 9 | 9 | 9 | 9 | 51.0 |
| 5c | TFS | 3 loop dummy | 627 | 9 | 9 | 9 | 9 | 53.1 |
| 5d | TFS | 4 loop dummy | 804 | 10 | 10 | 10 | 10 | 54.3 |
| 5e | TFS | 5 loop dummy | 973 | 9 | 10 | 10 | 10 | 54.7 |
| 6a | TFS | 1 task dummy | 320 | 9 | 9 | 9 | 9 | 54.1 |
| 6b | TFS | 2 task dummy | 472 | 9 | 9 | 9 | 9 | 53.3 |
| 6c | TFS | 3 task dummy | 636 | 9 | 9 | 9 | 9 | 53.8 |
| 6d | TFS | 4 task dummy | 822 | 9 | 9 | 10 | 9 | 55.3 |
| 6e | TFS | 5 task dummy | 1008 | 9 | 9 | 10 | 9 | 56.2 |
| 7a | TFS | 1 TFS | 335 | 12 | 12 | 13 | 11 | 56.1 |
| 7b | TFS | 2 TFS | 501 | 11 | 10 | 14 | 10 | 56.0 |
| 7c | TFS | 3 TFS | 673 | 11 | 11 | 12 | 10 | 56.3 |
| 7d | TFS | 4 TFS | 846 | 11 | 11 | 12 | 11 | 56.6 |
| 8a | ARC | 1 loop dummy | 232 | 9 | 9 | 10 | 9 | 36.7 |
| 8b | ARC | 2 loop dummy | 396 | 9 | 9 | 9 | 10 | 44.4 |
| 8c | ARC | 3 loop dummy | 549 | 9 | 9 | 9 | 9 | 46.6 |
| 8d | ARC | 4 loop dummy | 711 | 11 | 11 | 11 | 11 | 48.4 |
| 8e | ARC | 5 loop dummy | 870 | 11 | 11 | 11 | 11 | 49.3 |

Table 6.1 (continued)

| Exp. ID. | Target Appl. | Workload Description | CT (s) | System Overhead (%) | | | | MPO (%) |
|---|---|---|---|---|---|---|---|---|
| | | | | C1 | C2 | C3 | C4 | |
| 9a | ARC | 1 task dummy | 222 | 9 | 9 | 9 | 9 | 33.8 |
| 9b | ARC | 2 task dummy | 396 | 9 | 9 | 9 | 9 | 44.4 |
| 9c | ARC | 3 task dummy | 548 | 9 | 9 | 9 | 9 | 46.6 |
| 9d | ARC | 4 task dummy | 706 | 9 | 9 | 9 | 9 | 48.0 |
| 9e | ARC | 5 task dummy | 869 | 9 | 9 | 9 | 9 | 49.3 |
| 10a | ARC | 1 ARC | 235 | 10 | 10 | 13 | 10 | 37.4 |
| 10b | ARC | 2 ARC | 419 | 10 | 10 | 13 | 9 | 47.4 |
| 10c | ARC | 3 ARC | 593 | 10 | 10 | 11 | 10 | 49.6 |
| 10d | ARC | 4 ARC | 758 | 10 | 10 | 11 | 10 | 51.6 |
| 11a | TFS_SC | 1 loop dummy | 249 | 4 | 4 | 9 | 4 | 13.8 |
| 11b | TFS_SC | 2 loop dummy | 372 | 8 | 8 | 9 | 8 | 13.4 |
| 11c | TFS_SC | 3 loop dummy | 499 | 8 | 8 | 9 | 8 | 14.1 |
| 12a | TFS_SC | 1 task dummy | 247 | 4 | 4 | 9 | 4 | 12.9 |
| 12b | TFS_SC | 2 task dummy | 376 | 8 | 8 | 9 | 8 | 14.5 |
| 12c | TFS_SC | 3 task dummy | 509 | 9 | 8 | 9 | 8 | 15.7 |
| 13a | TFS_SC | 1 TFS_SC | 258 | NA | NA | 9 | NA | 17.0 |
| 13b | TFS_SC | 2 TFS_SC | 393 | NA | NA | 13 | NA | 18.2 |
| 13c | TFS_SC | 3 TFS_SC | 530 | NA | NA | 15 | NA | 19.0 |
| 14a | MCP | 1 loop dummy | 238 | 5 | 7 | 14 | 12 | 1.5 |
| 14b | MCP | 2 loop dummy | 337 | 9 | 10 | 12 | 12 | -3.8 |
| 14c | MCP | 3 loop dummy | 448 | 9 | 10 | 12 | 12 | -4.5 |
| 14d | MCP | 4 loop dummy | 555 | 9 | 9 | 10 | 9 | -5.4 |
| 15a | MCP | 1 task dummy | 236 | 10 | 10 | 13 | 10 | 0.1 |
| 15b | MCP | 2 task dummy | 342 | 10 | 9 | 13 | 9 | -2.6 |
| 15c | MCP | 3 task dummy | 449 | 10 | 10 | 14 | 10 | -4.2 |
| 15d | MCP | 4 task dummy | 552 | 9 | 9 | 10 | 9 | -6.0 |
| 16a | MCP | 1 MCP | 252 | 9 | 9 | 14 | 9 | 7.1 |
| 16b | MCP | 2 MCP | 374 | 9 | 10 | 15 | 10 | 6.1 |
| 16c | MCP | 3 MCP | 498 | 9 | 10 | 15 | 10 | 6.0 |

The MPO for each case is estimated using the techniques introduced in the last chapters. For instance, in the case of Experiment 7, the completion time of TFS executing in workloads with multiple versions of TFS assuming no MPO ($CT_{w/o\ MPO}$) was found using Equation (6.1).

$$CT_{w/o\ MPO} = (X + 1) * CT^{TFS}_{Ded.\ Mac.} \quad (6.1)$$

X: number of extra versions of TFS executing

With $CT_{w/o\ MPO}$ and CT (the actual completion time) determined, Equation (5.1) was used to determine the MPO incurred by the target application. Note that MPO is most accurately understood as being incurred by the target application and not as the percentage of processor time consumed.

The prodigious amount of data of Table 6.1 may at first seem daunting, but there are only a few major conclusions which will be highlighted. The most important observation to be drawn from the experiments is that multiprogramming affects different applications in widely different ways. Some applications execute poorly in a multiprogrammed environment, while others are not adversely affected. This is illustrated by comparing the TFS results (Experiments 5, 6, and 7) with the MCP results (Experiments 14, 15, and 16).

For TFS, MPO is a major performance problem ranging from 51 to 57%. In other words, over half of the execution time for TFS when executed with other applications is dedicated to overhead caused by multiprogramming. When executed alone, TFS takes 73.5 s (Table 3.4); when executed with just one other copy of TFS, the completion time jumps to 335 s. Viewed another way, if multiprogramming were eliminated and the two copies of TFS were executed in a batch mode (one after the other), they would both complete in 146 s instead of 335 s.

In contrast, for MCP the MPO incurred is either very small or negative. Negative values for MPO are not errors. They indicate that some functionality (maybe I/O) is being overlapped in the multiprogrammed workloads. For instance, when a page fault occurs, the processors need not remain idle waiting for it to be serviced, but can execute another application increasing throughput. Further analysis of MCP execution in multijob workloads shows that the MPO incurred for the serial part of the program (first 77 s before the tasks are spawned) is similar to that incurred by TFS_SC. Therefore, the overlapping is not occurring here. However, MPO incurred after the tasks are spawned is negative. Figure 6.5 shows that this period also corresponds to increased global memory use. Therefore, it is postulated that the good performance of MCP in multiple job workloads is due to the overlapping of processing and page faults (global and disk).

The ARC application incurs multiprogramming overheads almost as disconcerting as those of TFS. Measurements of MPO ranged from 33 to 52%. Therefore, high overheads can be incurred by applications written with either task (ARC) or loop (TFS) concurrency. Notice, though, that the type

of workload in which the target application is executed (loop or task dummy jobs) does not affect the MPO measurement significantly. Whether TFS is executed with 3 task dummy or 3 loop dummy jobs does not matter, the MPO incurred is still about 53%.

The MPO tends to be a little higher when the real target applications (TFS, ARC, and MCP) are executed with multiple copies of themselves. This is most likely due to increased memory contentions. The slow increase in MPO as the multiprogramming level grows (more applications are executed) is probably also partially due to increased memory contentions. An interesting case of MPO rising significantly with a higher degree of multiprogramming is Experiment 10. With one extra copy of ARC executing the MPO is 37.4%, while four extra copies of ARC cause the MPO to jump to 51.6. This is possibly due to increased synchronization conflicts when more applications are executing. Also, with more applications executing, there is a longer wait between time quanta assigned to a task. Therefore, if a task is context-switched off a cluster while it is in the middle of a critical section, it will hold the locking variable longer (increasing conflicts) with more applications present.

Note that the MPO incurred by TFS_SC (the single cluster version of TFS) is not as high as that for TFS which uses all four clusters. The MPO for the TFS_SC experiments ranged from 13 to 19%. These values are more in line with those found for the real workloads on the Alliant FX/80 and FX/8. The differences between TFS and TFS_SC point to areas which may be the cause of the high MPO: SDOALL loops and global memory. These will be investigated in more detail in the next section.

The table also shows that system overhead is fairly low and constant across the clusters. The fact that system overhead is much less than multiprogramming overhead for the real applications indicates that MPO is manifesting itself in the form of extra user time. This may be synchronization overhead (user spin locks) exacerbated by the multiprogramming. Again, this may be partially due to the fact that gang scheduling is not being used. This assumption is strengthened by the experiments with target applications that have little inherent synchronization (Exp. 1 - 4). For these experiments, the system and multiprogramming overheads are similar (approximately 10 - 11%).

A point which is not apparent from the table should be made. Like all controlled experiments of this thesis, the measurements are actually the results of multiple tests. For instance, experiment 5c was run five times, the high and low observations were discarded, and the middle three were averaged for the results. For all experiments except the ones with TFS there was not a large variance in the results obtained from the multiple executions. However, for TFS there was a noticeable variation in the completion times. This indicates some indeterminism in the execution of TFS. In parallel programs, a major contributor to indeterminism is synchronization points.

To summarize, the lower bound on MPO (approximately 10%), while not terrible, is still substantial. When real applications are executed, MPO can become a significant performance problem. However, for some applications, multiprogramming can overlap operations providing increased throughput. In the next section, experiments will be conducted to identify characteristics of programs which cause large overhead and those which perform well in multiprogrammed workloads.

### 6.3. Multiprogramming Overhead: Causes

Because TFS incurred the largest MPO measures, this section will concentrate on this application and applications written with loop concurrent constructs (SDOALLs). Each subsection will postulate an effect of multiprogramming and then conduct an experiment to test the hypotheses. Some hypotheses are proven correct, while others are shown to be false. The effects of the following in multiple job workloads will be investigated:

1) Helper tasks contentions.

2) Synchronization overhead in spreading loop iterations across the processors of a cluster.

3) Synchronization overhead in spreading iterations across clusters.

4) Global memory accesses.

It is found that a major cause of the observed multiprogramming overhead is the synchronization of helper tasks so that they can execute iterations of loops across all clusters. It is postulated that the

synchronization found in task concurrent jobs (e.g., ARC) also degrades performance in multiprogramming workloads. This may be avoided if a gang scheduling paradigm were used. However, the independent nature of the clusters on Cedar preclude such an arrangement. On the positive side, it is postulated that page faults can often be overlapped in multiprogrammed workloads to increase throughput.

### 6.3.1. Helper task contentions: the dawdle

As explained earlier, an SDOALL loop is physically implemented with the use of helper tasks. A helper task is bound to a cluster and does nothing except execute the iterations of the SDOALL loops. The iterations are assigned to the helper tasks in a self-scheduled manner. Normally, a CDOALL is nested in an SDOALL so that each iteration uses all eight processors of a cluster. Figure 6.6 shows the typical source code of an SDOALL loop.

Once created by the main task, the helper tasks constantly execute on the separate clusters. If there is no useful work (SDOALL iteration) for the helper task to do, it spins on the cluster checking the queue for incoming iterations. On a dedicated machine this is not a problem. If the main task does not have work for the helper task (it is not in an SDOALL), the cluster is idle so it may as well spin. However, in a multiapplication environment, the time spent by the "helper tasks" spinning could be used to execute a different application and increase throughput.

```
      SDOALL 12 I = 1, N
        CDOALL 12 J = 1, M
          ...
          ...
          <WORK>
          ...
          ...
   12 CONTINUE
```

Figure 6.6 Typical Nested SDOALL-CDOALL Loop

The *dawdle* function and the microtasking run-time library have been implemented to address this problem. If the application is compiled with a -M flag, the executable code will maintain software queues on the main task and the dawdle function on the helper tasks. In addition, the microtasking run-time library will be used instead of the nano-tasking library. The software queues allow for the execution of nested SDOALLs. A nested SDOALL can occur if an iteration of an SDOALL calls a subroutine with an SDOALL. As will be seen, the overhead for this functionality is high.

The dawdle function allows the helper tasks to abandon their time quantum and wait for work so that they will not spin on the processor needlessly. Actually, the processor spins for a short time checking the queue for work. If after the preallotted time there is no work present, the cluster is released. The goal is to create a more benign environment for multiapplication execution.

*Hypothesis:*

The MPO incurred by TFS when executed in workloads with other real applications (Experiment 7, Section 6.2) is partially attributable to helper task contentions.

*Experiment and Analysis:*

The TFS application was compiled with the -M flag so that the executable will use the micro-tasking run-time library and the dawdle function. This version of TFS will be called TFSM. TFSM was then executed with multiple versions of itself as TFS was executed with copies of itself in Experiment 7. The completion time of TFSM with multiple versions of TFSM and the corresponding results for TFS (reprinted from Table 6.1) are shown in Table 6.2.

Table 6.2
MPO Incurred by TFS and TFSM

| Appl. | # of extra copies of application executing | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | | 2 | | 3 | |
| | CT | MPO | CT | MPO | CT | MPO | CT | MPO |
| TFS | 74 | 0 | 335 | 56% | 501 | 56% | 673 | 56% |
| TFSM | 235 | 0 | 656 | 28% | 911 | 23% | 1173 | 20% |

The first column shows that the overhead of implementing the additional functionality and the dawdle function is high. The TFS code running with the nano-tasking library finishes in 73.5 s. TFSM (micro-tasking library) takes an average of 235 s on a dedicated machine.

The dawdle function appears to be successful. The MPO incurred by TFSM is significantly less than that seen by TFS. It is less than half for all cases. In addition, as more applications are executed (possibly increasing helper task contentions), the MPO drops. This reduction is probably due to a number of interrelated causes, though a major factor is surely the use of the dawdle function to reduce helper task contentions. It is interesting to note that even though the MPO is greatly reduced, the completion time of the TFSM workloads is still greater than that of TFS. The original overhead incurred by the micro-tasking library is not overcome by the reduction in the MPO for the case studied. The micro-tasking library is recommended for applications written with loop concurrency that have large sections of single cluster code.

*Conclusion:*

The hypothesis is accepted. MPO is increased due to contentions of helper tasks that do not have useful work. The situation can be avoided by maintaining queues and having the helper tasks release processors when there is no work. However, the overhead to allow this functionality may be significant.

### 6.3.2. Synchronization of CDOALL loops

When a task (helper or main) finishes the execution of one iteration of a CDOALL loop in a SDOALL-CDOALL nested loop (Figure 6.6) it checks the queue and grabs the next iteration to execute. This requires accessing a critical section of code guarded by mutual exclusion variables. In addition, at the end of the CDOALL loop there is some synchronization which must be done among the eight processors of the cluster. This is handled by Alliant's synchronization bus. To complicate matters, each iteration of the SDOALL loop (CDOALL) may not be of the same length. Each of these

items has the capacity to add overhead which can be heightened in a multiprogrammed environment. This section will investigate the effects of multiprogramming on these issues.

*Hypothesis:*

Overhead in synchronization of the CDOALL loops is exacerbated by multiprogramming.

*Experiment and Analysis:*

Three synthetic loops were constructed to test the above hypothesis. The loops were created to use a minimum of memory so that memory contentions do not add to MPO. The loops— LOOP_A, LOOP_B, and LOOP_C— are pictured in Figure 6.7. LOOP_A and LOOP_B do the same number of iterations of the body. However, the number of times the inner CDOALL loop has to synchronize is much greater for LOOP_B. LOOP_C was created to test the effect of different iteration lengths.

Each of the synthetic loops was executed and monitored in workloads consisting of multiple copies of the dummy loop jobs. The completion times and corresponding MPO incurred by the synthetic loops are shown in Table 6.3. The table shows that there is a slight overhead involved when the iterations of the SDOALL loop is spread. LOOP_A, which consists of 4 large iterations, takes 44 s on a dedicated machine, while LOOP_B, which has many more iterations, takes 100 s. Because both

```
    <LOOP_A>                                  <LOOP_B>

SDOALL 12 J = 1, 4                        SDOALL 12 J = 1, 10000000
     CDOALL 12 K = 1, 250000000                CDOALL 12 K = 1, 100
     X = X + Y                                 X = X + Y
12   CONTINUE                             12   CONTINUE

                          <LOOP_C>

                    SDOALL 12 I =1, 5000000, 10000
                       CDOALL 12 J = 1, I
                       X = X + Y
                    12 CONTINUE
```

Figure 6.7 Synthetic Loops: LOOP_A, LOOP_B, and LOOP_C

Table 6.3
MPO Incurred by LOOP_A, LOOP_B, and LOOP_C

| Appl. | # of dummy loop jobs executing | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | | 2 | | 3 | |
| | CT | MPO | CT | MPO | CT | MPO | CT | MPO |
| LOOP_A | 44.0 | 0 | 91.5 | 3.8% | 138.0 | 4.3% | 184.9 | 4.8% |
| LOOP_B | 100.3 | 0 | 209.6 | 4.3% | 313.4 | 4.0% | 418.3 | 4.1% |
| LOOP_C | 62.6 | 0 | 130.7 | 4.2% | 195.9 | 4.1% | 264.2 | 5.2% |

loops execute the same number of iterations, the extra time is attributable to the assigning of iterations to the tasks (the overhead of a sprading an SDOALL).

This overhead, though, is not adversely affected by multiprogramming. When multiple jobs are executed with the synthetic loops, the MPO incurred is negligible. This is true also for the case in which the iterations are of different lengths.

*Conclusion:*

The hypothesis is rejected. There is overhead in distributing the iterations of the SDOALL loop and synchronizing the CDOALL loops. However, this overhead is not exacerbated by multiprogramming.

### 6.3.3. Synchronization of SDOALL loops

Each iteration of the SDOALL loop (usually a CDOALL loop — Figure 6.6) is self-scheduled to a cluster and executed by a helper task. When all of the iterations of an SDOALL loop are complete, the helper tasks must "check-in" with the main task before execution resumes. In other words, at the end of the SDOALL loop, a barrier synchronization must be performed. Once all helper tasks are have checked in, the main task can resume with serial execution.

The overhead of barrier synchronizations on parallel applications in multiuser environments has been studied through simulation [43]. Hot spots in memory and the context switching of a task while executing a critical section are both adverse effects of multiprogramming on synchronization. In this

subsection, it will be determined if the synchronization of the SDOALL construct on Cedar is detrimentally affected by multiprogramming.

*Hypothesis:*

Overhead caused by synchronizing SDOALL loops across clusters is exacerbated in a multiprogrammed environment.

*Experiment and Analysis:*

Two separate experiments were conducted to test the hypothesis. First, the execution time of the SDOALL loops in TFS were measured for all of the workloads of Experiment 5. The distribution of SDOALL lengths for the experiments of Experiment 5a and 5d are shown in Figure 6.8(a) and 6.8(b), respectively. To allow for comparison with Figure 6.3(a), only those values under 10 ms are shown. Therefore, there are many instances missing from both distributions.

The average length of an SDOALL in TFS when executing with one dummy loop job on the system is 8.11 ms; with four dummy loop jobs executing the average is 18.84 ms. Comparing these averages and distributions with each other and those found on a dedicated machine (Figure 6.3, 1.44 ms) indicates that, as expected, the SDOALL loop takes longer as the multiprogramming level increases.

A more telling metric is the percentage of the execution time in which the application was inside an SDOALL loop. On a dedicated machine, the SDOALLs account for 75.5% of the execution time. However, for experiment 5a this parameter jumps to 92.2%; and for Experiment 5d the percentage of completion time in which the application was either executing code in an SDOALL or context-switched off the machine in the SDOALL loop was 94.5%. As more jobs are added, a larger percentage of time is spent executing the SDOALLs. Therefore, the observed performance degradation of TFS in multiapplication workloads has its root somewhere in the SDOALL loops. However, it is not clear if the cause is the code of the SDOALL loops (the iterations) or the implementation of the SDOALL itself.

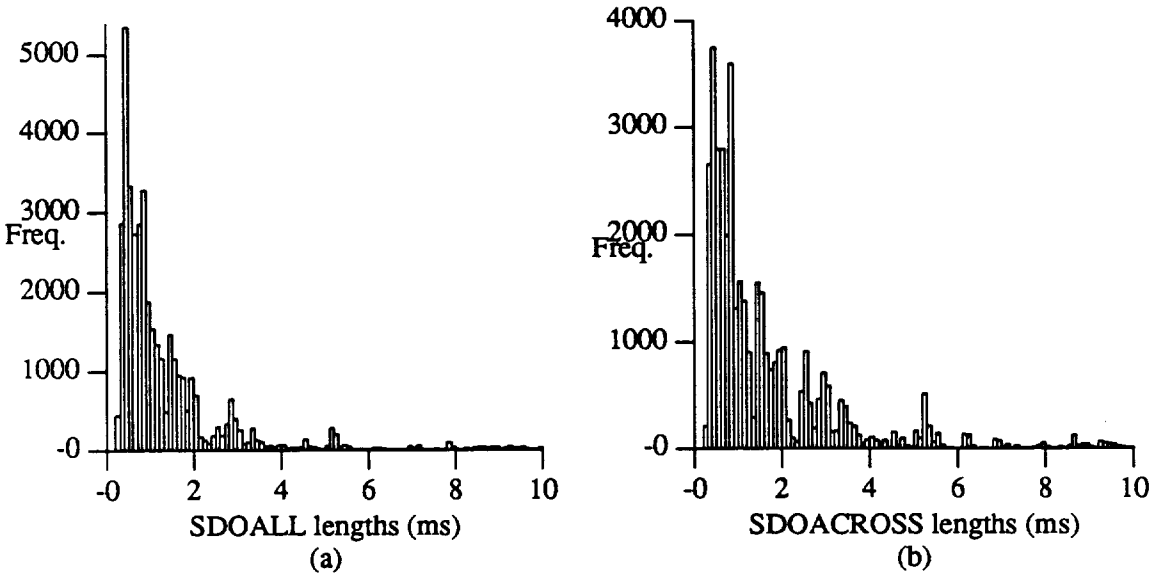



Figure 6.8 Length of TFS SDOALLs: Experiment 5a and 5d

The second experiment determines if the cause of the degradation found in the previous experiment is caused by the SDOALL mechanisms themselves. Two artificial loops were constructed: LOOP_D and LOOP_E (Figure 6.9). LOOP_D forces the mechanisms of the SDOALL to be executed repeatedly. LOOP_E does the same work as LOOP_D except that only one SDOALL synchronization must be done. It is used as the control application. Both loops were executed in workloads consisting of multiple dummy loop jobs. The completion times and MPO values are listed in Table 6.4.

On a dedicated machine, the overhead for executing and synchronizing an SDOALL is small. The section of code with repeated SDOALL (LOOP_D has 10,000 loops) takes just 10 s longer than the loop with only one SDOALL (LOOP_E). Therefore, the actual processing power consumed to

```
            <LOOP_D>                          <LOOP_E>

DO 12 I = 1,10000                     DO 12 I = 1,1
      SDOALL 12 J = 1, 100            SDOALL 12 J = 1, 100
            CDOALL 12 K = 1, 1000               CDOALL 12 K = 1, 10000000
            X = X + Y                           X = X + Y
12  CONTINUE                          12  CONTINUE
```

Figure 6.9 LOOP_D and LOOP_E

Table 6.4
MPO Incurred by LOOP_D and LOOP_E

| Appl. | # of dummy loop jobs executing | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | | 2 | | 3 | |
| | CT | MPO | CT | MPO | CT | MPO | CT | MPO |
| LOOP_D | 54.5 | 0 | 300 | 64% | 447 | 63% | 626 | 65% |
| LOOP_E | 44.3 | 0 | 91.5 | 3.2% | 138 | 3.7% | 188 | 5.7% |

execute the overhead associated with an SDOALL is small. However, in multiple job workloads, the overhead caused by the SDOALL is quite high. For LOOP_D executing with one dummy loop job the MPO is 64%. In other words, 64% of the completion for LOOP_D is attributed to MPO. This may be a result of tasks being contexted switched off the machine while executing critical sections. In this situation, other tasks which may be trying to access the critical section or lock a mutual exclusion variable can only spin helplessly. The overhead may also be partially due to hot-spots in memory.

A fact not shown in the table supports this conclusion. The completion times for the multiple runs of each LOOP_E experiment were fairly consistent. Each execution of LOOP_E in a given workload produced a similar completion time. For instance, for the case with one dummy job executing the completion times for five experimental runs were: 91.4, 91.4, 91.5, 91.6, and 91.6 s. LOOP_D was not as deterministic. The case with one dummy loop job executing produced completion times of 263.9, 237.8, 284.2, 299.5, 365.3, 342.8, and 307.1 s. This indeterminism indicates chance activity in execution such as the locking and unlocking of queue locks.

*Conclusion:*

The hypothesis is strongly accepted. Overhead caused by synchronizing the helper tasks of an SDOALL loop is exacerbated in multiapplication workloads and significantly degrades performance. The problem appears to occur in the barrier synchronization at the end of the SDOALL loops. A program containing many short SDOALL loops (such as TFS or LOOP_D) will perform poorly in a multiapplication workload.

### 6.3.4. Removal of SDOALL loops from TFS

The last subsection presented strong empirical results suggesting that applications written with the SDOALL construct are highly susceptible to multiprogramming overhead. Because TFS is such an application, the obvious conclusion is that the poor performance of TFS in multiuser workloads is largely due to this construct. This assertion was corroborated by the increase in the fraction of completion time TFS spent executing SDOALL loops when it was run in multiprogrammed environments. In this section this claim is further tested. TFS is modified by replacing all SDOALL loops with CDOALLs. The new application (called TFS_ser) is a single cluster version of TFS with the same memory reference patterns as TFS.

This experiment is conducted for two purposes: First, to validate the conclusion of the last subsection, and second, to investigate the effect accesses to global memory may have on MPO.

*Hypotheses:*

A) SDOALL synchronization contentions is a major contributor to MPO for TFS. B) Global memory accesses are major contributors to MPO.

*Experiment and Analysis:*

The TFS application was rewritten (TFS_ser) with all SDOALLs converted to CDOALLs. Functionally, converting SDOALL-CDOALL nested loop (Figure 6.6) to a CDOALL-CDOALL nested loop serializes the inner CDOALL loop. The memory accesses (cluster or global) remain unchanged, but the application executes on a single cluster.

The completion time of TFS_ser on a dedicated machine is 157.4 s. The increase in completion time over TFS (73.5) is caused by executing all loops on a single cluster instead of four. The TFS_ser application was then executed and monitored in workloads consisting of multiple copies of dummy loop jobs. The completion times and MPO incurred by TFS_ser in the multiple dummy jobs workloads are summarized in Table 6.5. The same numbers for TFS and TFS_SC are also given.

Table 6.5
MPO Incurred by TFS, TFS_SC, and TFS_ser

| Appl. | # of dummy loop jobs executing | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | | 2 | | 3 | |
| | CT | MPO | CT | MPO | CT | MPO | CT | MPO |
| TFS | 74 | 0 | 301 | 51% | 450 | 51% | 627 | 53% |
| TFS_ser | 157 | 0 | 390 | 19% | 507 | 6.8% | 659 | 4.5% |
| TFS_SC | 108 | 0 | 249 | 14% | 372 | 13% | 499 | 14% |

The MPO experienced by TFS_ser is small compared to that of TFS and TFS_SC for the same workloads. This corroborates the conclusion of the last subsection: synchronization overhead of SDOALL loops is exacerbated by multiprogrammed workloads. It should also be noted that the completion time of TFS_ser in the multiple job workloads were more consistent than those of TFS. This again points to the synchronization problem.

The difference in MPO between TFS_SC and TFS_ser is probably due to a more subtle cause. Functionally, the major difference between the two is that TFS_SC stores all of its data in cluster memory, while TFS_ser stores a good deal of data in global memory. The difference in completion times on a dedicated machine (TFS_SC: 107, TFS_ser: 157) can be partially attributed to this fact. Also, remember that the source code for the two are drastically different.

The TFS_ser application incurs much smaller MPO then TFS_SC. This could possibly be due to I/O overlapping similar to that found in the MCP experiments. Page faults to disk, or soft page faults to global memory, can be serviced in tandem with work done on other applications. Therefore, it appears that accesses to global memory are not adversely affected by multiprogramming.

*Conclusions:*

Hypothesis A is accepted, while hypothesis B is rejected. When executed in multiuser workloads, SDOALL loops cause performance degradation due to synchronization contentions. Global memory accesses do not appear to be a major contributor to MPO.

C-2

### 6.4. Conclusions: Cedar Multiprogramming

This chapter has illustrated the importance of evaluating the performance of multiuser workloads. On a dedicated machine, the Cedar system performs admirably. However, the performance of certain applications in multiprogrammed environments is poor. The MPO estimation technique identified a few applications whose performance was significantly degraded. Further analysis identified a synchronization problem with SDOALL loops.

In general, the analysis has shown that care must be taken with fine-grained parallelism on cluster-based multiprocessors. The synchronization involved when coupled with multiprogramming may reduce performance. For this type of parallelism to be successful on this architecture, large loops should be parallelized. Also, gang scheduling may reduce the effect of multiple applications.

# CHAPTER 7.

# MODELING APPLICATION EXECUTION

Chapter 5 introduced a methodology capable of quantifying application performance degradation due to multiple job interactions in real workloads. The methodology was illustrated on the Alliant FX/8 and FX/80. The next step is to use this information in constructing realistic, accurate models of application execution in real workloads. The model must capture both the resource sharing of multiple real applications and the added overhead caused by this sharing. With such a model, application performance could be predicted and tuned, and system design changes could be accurately evaluated.

This chapter presents a methodology which uses real workload measurements to construct such a model. More specifically, a technique is introduced which models the behavior of an application from a given domain executing in real workloads on a particular machine. The model is a Markov reward model capable of predicting the completion time distribution for an application executing under real workloads. More importantly, the model can also predict the effects of system design changes. For instance, the model is capable of predicting the completion time distribution of an application in real measured workloads under different scheduling paradigms without implementing the new schedulers.

To build the Markov reward model, applications representing the given job domain are executed numerous times during the normal operation of the machine. System parameters such as job queue lengths and multiprogramming overhead are monitored during the executions of the applications. Statistical clustering is then used on the collected data to identify a finite-state, discrete-time Markov model. The final step involves assigning a reward to each state to quantify the actual system resource available to an application in that state.

Monte Carlo simulation is used to solve the model and predict the completion time distribution of a specific application under the measured workload. System design changes are modeled by modifying the reward function of each state in the Markov model.

This chapter is divided into two sections. In the first section, the model building and solution methodology are presented in a machine-independent way. In the second section, the methodology is illustrated by modeling the execution of computationally bound, parallel applications executing in real workloads on the Alliant FX/80. In the chapter following this, the constructed model will be used to investigate a large number of scheduling paradigms and machine configurations for the Alliant FX/80.

### 7.1. Model Construction (Machine Independent)

The objective of the methodology is to build a model of the system and workload as it would be seen by an application of the targeted domain. It is important to define precisely the application domain for which the model is constructed. Given a workload, a parallel application will view the available resources much differently than would a serial job. Therefore, the model built for a serial job must be entirely different than that for a parallel job even if the same workloads are measured. As would be expected, the constructed model provides reliable results only for applications of the correct domain.

The five steps of model construction and solution are:

1) Monitor system/workload parameters while a target application executes during normal machine operation.

2) Statistically cluster measured data to identify key states of system/workload operation.

3) Convert the identified cluster model into a Markov model.

4) Define reward and cost functions for each state of the Markov model.

5) Given base resource requirements of the application in question, solve for completion time distribution.

*STEP ONE —*

The first step is similar to the monitoring step introduced in the real workload MPO methodology. The data necessary to build the model are obtained by monitoring the system while an application of the targeted class (referred to as a *target application*) executes in a normal workload. The periods in which a target application is executed are chosen randomly over an extended period of time. By monitoring a large number of different types of real workloads, the model becomes robust enough to represent accurately the system/workload that would be seen by an application (similar to the target application) in many situations.

Choosing the workload parameters to monitor while the target application executes is entirely dependent on the reward functions. The parameters monitored are those needed to calculate, using the reward function, the amount of resources an application of the specified domain will receive. For instance, parameters determining the processor power or the amount of main memory available may be monitored. The resource or resources chosen to be used as the reward are those which are most crucial to an application's execution. Therefore, choosing the parameters to be monitored is a direct consequence of what resource or resources are used as the reward.

The execution time of the target application is split into intervals of nearly equal length called *observations*. Each observation describes a system/workload state with the parameters measured during that time period. The observation's length is chosen by trial and error to reflect the granularity of the system/workload states. The result of the monitoring step is a large number of observations each defined by parameters used to determine reward and cost.

*STEP TWO —*

The next step in model construction is summarizing the abundance of measured data by identifying the distinct system/workload states in which the machine has operated. This is done with statisti-

cal clustering,[3] a common statistical technique commonly used in computer analysis [67]-[70]. Statistical clustering treats each observation as a point in n-dimensional space (one dimension for each parameter defining the observation) and groups the points into clusters that maximize the Euclidean distances between the clusters while minimizing the Euclidean distances among the observations of a cluster. Statistical clustering identifies groups of similar observations. Here, statistical clustering is used to identify common system/workload states of operation.

There are many statistical clustering algorithms available. In this thesis, the goal is to identify distinct states of machine operation; thus the clustering algorithm used should delineate nonoverlapping clusters. For this thesis, the K-means algorithm was chosen [71]. However, in general, any clustering method which identifies distinct groups could be used to accomplish this step of the methodology.

Before clustering, the measured data should be standardized so that each parameter has a mean of zero and a standard deviation of one. This is done so that the parameters with the largest range of values do not dominate the clustering procedure. Choosing the correct number of clusters to capture the real workload behavior accurately has been previously discussed [67][69]. For the sake of standardization, it is suggested that the number of clusters used be the least number for which 90% of the cluster radii (maximum distance from the cluster centroid to an observation in the cluster) are less than 1.5. After the original clustering, observations in small clusters should be absorbed into the nearest large cluster. This is to guard against bias due to outliers.

At the conclusion of Step Two, there are a number of groups or clusters of observations. Each group identifies a unique, naturally occurring state of the observed workload. The state and the observations contained in that state can be summarized by the centroid values (geometric center) of the cluster.

---

[3]Further apologies for the overloading of meanings on the term *cluster*. Statistical clustering and physical clustering of the CEs are completely unrelated, just as a cluster on Cedar is unrelated to the clustering of the CEs. The context in which the term *cluster* is used should make its intended meaning clear.

*STEP THREE—*

The next step is to convert the cluster model to a Markov model. Here the methodology relies on the memoryless property of Markovian analysis— the next workload/system state of the machine is totally dependent on the previous state. The validity of such an assumption depends on the parameters defining the states. For instance, it has been postulated that job length queues obey the Markov property.

The identified clusters translate directly to states in a discrete Markov model. Transitions between the observed states are garnered from the measured data. The probability of moving to state i after being in state k is estimated by dividing the observed transitions from state i to state k by the number of transitions out of state k. The Markov model constructed can be either discrete or continuous in time. The starting state probabilities are estimated using the first observations of each target application execution.

At the end of this step, there exists a Markov model with states summarizing observed system/workload states and transitions summarizing observed transitions between these states.

*STEP FOUR —*

Now a reward and a cost is associated with each state of the Markov model. The reward quantifies the amount of a resource that an application of the class being modeled would be given if it were submitted to the system while the system was in the state defined by the Markov model state. The reward for each state is calculated using the centroids of the clusters. In general, the reward can quantify anything from CPU time to main or cache memory. However, the resource chosen should be the one that most directly affects the completion time of the application domain modeled. If there is more than one such resource, the model can be extended to associate multiple reward functions to each Markov state. The cost is the wall clock time needed to obtain the reward. It is monitored along with the parameters in Step One.

*STEP FIVE —*

The model is capable of predicting the completion time distribution of an application from the modeled domain if it were to execute in workloads such as those measured on the given machine. In other words, given an application, the model can determine the probability that it will finish by a given time if it were executed in workloads similar to those measured. To do this, the amount of reward resource the application requires to complete is first determined. This can be done by monitoring the execution of the application on a dedicated machine or through analytical techniques.

Let the amount of resource needed by the application to complete be represented by $P$. Let $W_k$ be the state the Markov model is in at time k, $r_W$ the reward for Markov state W, and $c_W$ the cost for Markov state W. Then the accumulated reward at time i, $Y_i$, is calculated with Equation (7.1).

$$Y_i = \sum_{j=0}^{i} r_{W_j} \tag{7.1}$$

The accumulated cost at time i, $Z_i$, is calculated with Equation (7.2).

$$Z_i = \sum_{j=0}^{i} c_{W_j} \tag{7.2}$$

The probability that an application needing P reward will finish by time x, $F_P(x)$, is then given by Equation (7.3).

$$F_P(x) = Prob\{Z_i \le x \mid Y_i \ge P, Y_{i-1} < P\} \tag{7.3}$$

A closed form solution to a problem similar to this one relying on double Laplace transforms has been proposed [72]. However, the transform solution relies on complicated numerical techniques and the result is difficult to simplify. For the sake of simplicity, Monte Carlo techniques can be used to solve the model and estimate the completion time distribution of a given application. The Monte Carlo method relies on multiple simulations of application execution. The states of the model are stepped through accumulating both reward and cost. When enough reward is collected, the cost is a single estimate of completion time. This technique will be presented formally when solving the model constructed for the Alliant (Section 7.1).

The constructed model facilitates two powerful performance analysis techniques. First, it can be used to determine how well a given application will run during the normal operation of the machine. Second, the model can predict the effect on performance of certain system design changes. This is useful for tuning the machine and making future generations of the machine more responsive to actual work being done.

The first goal is obtained by simply solving the model as stated in step 5 for the completion time distribution. To predict the effect on performance of a system change, the reward function is modified to model the appropriate change. The completion time distribution for an application is then determined with the new reward functions. The distribution computed under the modified reward is compared to that under the normal reward to evaluate the modification. Using the completion time of a test application as the metric of comparison, the model can successfully determine which system changes are beneficial.

It should also be emphasized that some of the steps in the methodology, most notably determining the reward (step 4), are highly machine dependent and require detailed knowledge of the system under investigation. It should also be noted that the model allows only applications of the chosen class to be analyzed (e.g., computationally bound, parallel applications). This is not overly restrictive though, because a class of applications is generally quite broad. However, the effects of the system design change on other types of applications (not just those from the modeled domain) should also be considered before coming to any general conclusions. This can be done by building a new model to investigate the different classes of jobs (e.g., I/O bound jobs).

## 7.2. Alliant FX/80 — Model Construction

### 7.2.1. Monitoring and measuring the system

The goal is to build a model of computationally intensive, parallel jobs executing in real workloads on the Alliant FX/80. The data necessary to build the model are obtained by monitoring the

system while an application of the targeted class executes in a normal workload. Three applications were chosen from the Perfect Club benchmark suite to be used as target applications. The three— Dyfesm, Flo52, and Track— are listed along with their base processing requirements in Table 3.3. All three applications were compiled as type A cluster jobs and execute on all eight CEs in the clustered configuration.

The data for the model were collected by monitoring the Alliant during 100 separate target application executions distributed over a 3 month period. The execution times were randomly chosen during the machine's normal operation. The 100 executions included 35 executions of Dyfesm, 35 of Flo52, and 30 of Track.

Because the applications are computationally intensive, the most critical resource for completion is clustered processing time. Therefore, for this model, reward was defined as the amount of clustered CE time given to a Type A cluster job. Hence, parameters that allow for the calculation of deliverable clustered CE time were monitored. In addition to the parameters necessary to determine the reward, parameters which affect the amount of clustered CE time an application would receive under the system design changes under investigation were also measured. In addition, the parameter necessary for the cost function was monitored.

In the next chapter, the system design changes investigated will be detailed, and the reasoning behind some of the parameters monitored will become clear. From the scheduling algorithm (Table 3.1), and past work [73], it was determined that the first five parameters of Table 7.1 were necessary to calculate the amount of clustered CE time (reward) which would be granted to a parallel job. The fifth parameter is also necessary because it will be the cost of the state. The last two parameters in the table are needed to model certain system design modifications. For these reasons, the seven parameters listed in the table were monitored while the target application was executed.

The parameters of Table 7.1 were obtained from measurements taken using the Q and HRTIME facilities. The MPO was determined using the technique of Chapter 5.

Table 7.1
Observation Parameters

| Parameter | Symbol | Description |
|---|---|---|
| 1 | CLA | avg. number of type A cluster jobs |
| 2 | %CLC | % of time at least one type C cluster job |
| 3 | CLUSP | % of time CEs execute cluster jobs |
| 4 | MPO | multiprogramming overhead |
| 5 | TIME | Length of observation |
| 6 | CLC | avg. number of type C cluster jobs |
| 7 | %CLA | % of time at least one type A cluster job |

The monitoring procedure is illustrated by Figure 7.1. The target application is run under a normal workload. Both HRTIME and Q are invoked once at the inception and once at the completion of execution. In addition, the Q facility is invoked approximately once every 1.2 s to measure the job queue lengths (short arrows in Figure 7.1 represent Q invocations). Because the Q facility must be submitted as a software job, the time between measurements varied (standard deviation = 0.28 s). Most of the work required to execute Q was done by the IPs thus the perturbation of the workload on the CEs was negligible.

Figure 7.1 also illustrates the division of target application execution into observations. Each observation is made up of five consecutive samples, resulting in an average observation length of 6.094 s (standard deviation = 1.37 s). A variety of observation sizes were tested. It was determined that five samples was a decent granularity to reflect a single system/workload state.

For each observation, the parameters of Table 7.1 were estimated. Equations (7.4) - (7.10) use the nomenclature of Figure 7.1 to detail how the parameters were calculated.

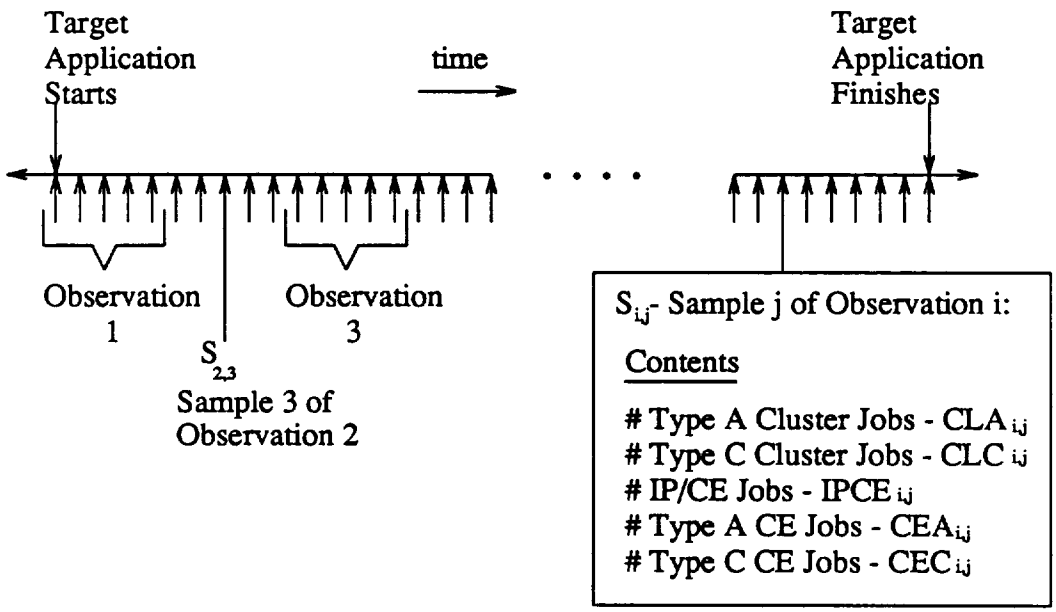



Figure 7.1 Monitoring System

$$CLA_i = \frac{1}{5} \times \sum_{j=1}^{5} CLA_{i,j} \tag{7.4}$$

$$\%CLC_i = \frac{1}{5} \times \sum_{j=1}^{5} IND\,[CLC_{i,j} > 0] \qquad IND\,[arg] = \begin{cases} 1 & \text{if } arg \text{ is } True \\ 0 & \text{if } arg \text{ is } False \end{cases} \tag{7.5}$$

$$CLUSP_i = \frac{1}{5} \times \sum_{j=1}^{5} IP_{i,j} \qquad IP_{i,j} = \begin{cases} 1 & \text{if } IPCE_{i,j} = CEA_{i,j} = CEC_{i,j} = 0 \\ \frac{7}{13} & otherwise \end{cases} \tag{7.6}$$

$$MPO_i = MPO \text{ for } execution\ of\ entire\ program \tag{7.7}$$

$$TIME_i = \frac{Completion\ Time\ of\ Target\ Application}{Number\ of\ Samples\ Taken} \tag{7.8}$$

$$CLC_i = \frac{1}{5} \times \sum_{j=1}^{5} CLC_{i,j} \tag{7.9}$$

$$\%CLA_i = \frac{1}{5} \times \sum_{j=1}^{5} IND\,[CLA_{i,j} > 0] \tag{7.10}$$

The average number of type A cluster jobs in the system during observation i, $CLA_i$, was estimated by averaging the five sampled measurements of cluster A job queue length making up the

observation (Equation (7.4)). Parameter 2 was estimated by the fraction of the five samples in which there was at least one type C cluster job (Equation (7.5)).[4] The percentage of observation time in which cluster jobs were executing ($CLUSP_i$) was estimated using the sampled queue lengths and the scheduling information of Table 3.1 (Equation (7.6)). The equation assumes that if only cluster jobs were present in a sample, then the CEs executed cluster jobs the entire time. If, however, there were CE or IP/CE jobs present, the clustered jobs were given the CEs $\frac{7}{13}$ of the time. The percentage of time the clustered CEs were executing multiprogramming overhead ($MPO_i$) was determined for the entire run of the application and this number was assigned to each observation in that execution. The length of each observation period ($TIME_i$) was estimated by dividing the completion time of the target application by the number of samples taken during the execution (Equation (7.8)). The calculation of $CLC_i$ and $\%CLA_i$ is shown by Equations (7.9) and (7.10).

The entire Alliant FX/80 time monitored (100 target application executions) produced 21,046 observations over approximately 36 h of real computer time. Each of the 21,046 observations is defined by the parameters listed above, and each can be seen to capture a certain state of system/workload execution.

### 7.2.2. Statistical clustering

Statistical clustering of the 21,046 observations was done in the five-dimensional space defined by the first five parameters of Table 7.1. The measured data were first standardized so that each parameter had a mean of zero and a standard deviation of one. This was done so that the parameters with the largest range of values did not dominate the clustering procedure. Clustering of the standardized data was accomplished with the FASTCLUS procedure of the SAS software package; FASTCLUS is based on the K-Means algorithm [71].

---

[4]The equation introduces the indicator function IND[] which is used throughout this paper.

Using the criteria set out in Section 7.1, the observations were grouped into 80 distinct clusters ($r^2 = 0.94$). Clusters of less than 10 observations were then dissolved by moving their observations into neighboring clusters. This operation resolved bias due to outliers and reduced the number of clusters from 80 to 72. Therefore, with regard to the five parameters clustered upon, the real machine operation was described by 72 system/workload states.

Choosing the number of clusters must balance the opposing goals of tractability and reduction of outlier bias (few clusters) with that of accuracy (many clusters). Grouping real workload machine operation into 72 distinct states may appear arbitrary and it is to a certain extent. However, as will be shown later, the methodology is not highly dependent on the number of clusters. Similar results would be obtained if any number of clusters near 80 were originally used.

This question is addressed in more detail in Figures 7.2. Figure 7.2(a) shows the $r^2$ value (a measure of goodness of fit) as a function of the number of clusters. Usually values above 0.9 indicate that the number of clusters chosen has accurately identified groups of similar data. Using this criterion, at least 44 clusters should be used to summarize the data of this experiment.

Figure 7.2(b) plots the number of clusters left after absorbing outliers (clusters with less than 10 observations) against the number of groups originally clustered into. For instance, the (x,y) pair (80,72) is on the line in the figure because when the data are clustered into 80 groups, there are 8 clusters with less than 10 observations, leaving 72 clusters after resolving outlier bias. The figure shows that when too many clusters are used, there are many groups with few observations. This is a problem because states with only a few observations may not be identifying actual workload/system states. With this in mind, the number of clusters used should probably be less than 110. Fortunately, the criterion proposed for choosing the correct number of clusters falls between these loose bounds (44 — 110).

After clustering, the data points were returned to their original values and the cluster centroids were calculated. Cluster centroids are the geometric centers of the clusters (i.e., the average of all the

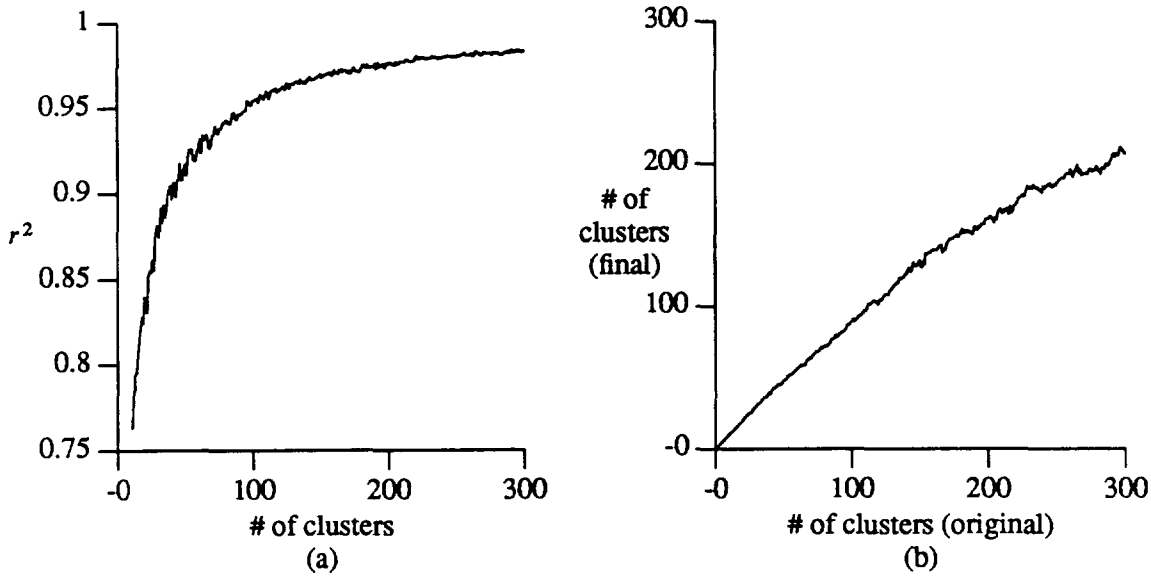



Figure 7.2 Varying the Number of Clusters

observations in the cluster). The superscript C will indicate a centroid value. For example, $CLA_i^C$ refers to the centroid value of cluster $i$ corresponding to the average number of cluster A jobs present. Table 7.2 lists the size and centroid values of a number of representative clusters.

### 7.2.3. Discrete-time Markov model

Once the system/workload states are identified, the model is enhanced to account for observed temporal relations between these states. The cluster model is transformed into a discrete-time Markov

Table 7.2
Centroids of Representative Clusters

| Cluster Number | number of Observations | Centroid Values | | | | |
|---|---|---|---|---|---|---|
| | | $CLA^C$ (# jobs) | $\%CLC^C$ (% time) | $CLUSP^C$ (% time) | $MPO^C$ (% time) | $TIME^C$ (s) |
| 1 | 147 | 5.66 | 0.97 | 0.54 | 0.24 | 6.85 |
| 7 | 168 | 7.44 | 0.01 | 0.54 | 0.16 | 6.62 |
| 20 | 43 | 2.94 | 0.99 | 0.54 | 0.09 | 9.19 |
| 26 | 803 | 1.15 | 0.99 | 0.56 | 0.15 | 4.97 |
| 33 | 139 | 7.83 | 0.98 | 0.54 | 0.17 | 6.64 |
| 49 | 465 | 1.14 | 0.00 | 0.54 | 0.35 | 5.28 |
| 52 | 125 | 0.91 | 0.02 | 0.76 | 0.14 | 5.58 |
| 68 | 1400 | 2.62 | 0.01 | 0.54 | 0.15 | 5.45 |

model whose transition probabilities represent observed transitions among system/workload states. The 72 states of the cluster model map directly to Markov model states. The elements of the transition matrix, $\Phi = [P_{i,j}]$, are estimated using Equation (7.11). The probability $P_{i,j}$ is determined by dividing the number of times an observation in cluster i ($C_i$) was followed with an observation in cluster j ($C_j$) by the number of observations in cluster i that have successors.

$$P_{i,j} = \frac{\sum_{k=1}^{100} \sum_{l=1}^{OBS(k)-1} IND\,[(O_{k,l} \in C_i) \wedge (O_{k,l+1} \in C_j)]}{\sum_{k=1}^{100} \sum_{l=1}^{OBS(k)-1} IND\,[O_{k,l} \in C_i]} \tag{7.11}$$

$C_i$ : *Cluster i* ; $O_{k,l}$ : *Observation l of* $k^{th}$ *target appl. execution*

$OBS(k)$ : *# of Obs. in* $k^{th}$ *target appl. execution*; $1 \le i,j \le 72$, $1 \le k \le 100$

The starting states' probabilities, $\pi(0)=[\pi_i(0)]$, are estimated with Equation (7.12). The probability of starting in state i, $\pi_i(0)$, is estimated by dividing the number of times the first observation of a target application execution is in state i by the total number of target application executions.

$$\pi_i(0) = \frac{\sum_{k=1}^{100} IND\,[O_{k,1} \in C_i]}{100} \tag{7.12}$$

### 7.2.4. Reward and cost functions

The cluster centroids are now used to associate a reward with each state of the Markov model. The reward quantifies the amount of resources that a computationally intensive, parallel job would receive if it were submitted to the system. Because the resource with the greatest impact on the completion time of computationally bound, parallel applications is clustered processing time, this is the resource used for the reward. Therefore, the reward function calculates the amount of clustered time that would be given to a type A cluster application if it were submitted to the system while in the system/workload state described by that Markov model state.

The parameters necessary to determine the reward are the five parameters on which the model was built. The reward for each state, $R_i$, is determined using Equation (7.13). The equation multiplies

the average time spent in the state ($TIME^C$) by the fraction of time the system would process a type A cluster job if it were submitted.

$$R_i = (TIME_i^C) \times CLUSP_i^C \times (\frac{1}{1+CLA_i^C}) \times (1 - (\%CLC_i^C \times NORMC_i)) \times (1 - MPO_i^C) \quad 1 \le i \le 72$$

$$NORMC_i = \frac{1}{CLUSP_i^C} \times [(\frac{7}{13} \times \frac{4}{7}) + ((CLUSP_i^C - \frac{7}{13}) \times \frac{2}{3})] \tag{7.13}$$

The equation is best explained one term at a time. $CLUSP^C$ is the fraction of time for which cluster jobs (type A or C) are executed. Of the $CLUSP^C$ fraction, a certain percentage is given to type A jobs and a certain percentage to type C jobs. The term $(1 - (\%CLC^C \times NORMC))$, which is determined from the scheduling algorithm (Table 3.1), estimates the fraction of $CLUSP^C$ given to type A cluster jobs. The fraction of time granted to Type A cluster jobs is shared equally among them. This is accounted for by the $\frac{1}{1+CLA^C}$ term. The last term accounts for multiprogramming overhead.

Figure 7.3 graphically shows the division of an observation into its components isolating the amount of clustered CE time available to a type A cluster job. The top line shows the total observation time divided into a clustered and detached component. Of the clustered time, a fraction is given to type A and a certain fraction to type C cluster jobs (second line in figure). The third line shows the equal sharing of the remaining time among the type A jobs. Finally, the last line accounts for MPO.

The cost associated with each state ($COST_i$) is the wall clock time needed to achieve the reward of the state (Equation (7.14)). The rewards and costs associated with the clusters detailed in Table 7.2 are given in Table 7.3. For instance, when the system is in the state represented by State 7, an application will receive 0.351 s of clustered time for each 6.62 s in the system. Consulting Table 7.2 it is seen that this low reward is due to the abundance of type A jobs in the observations of that state.

$$COST_i = TIME_i^C \tag{7.14}$$

Figure 7.4 shows part of the constructed model: a discrete-time, discrete-state Markov reward/cost model. The states correspond to observed system/workload states and the transitions

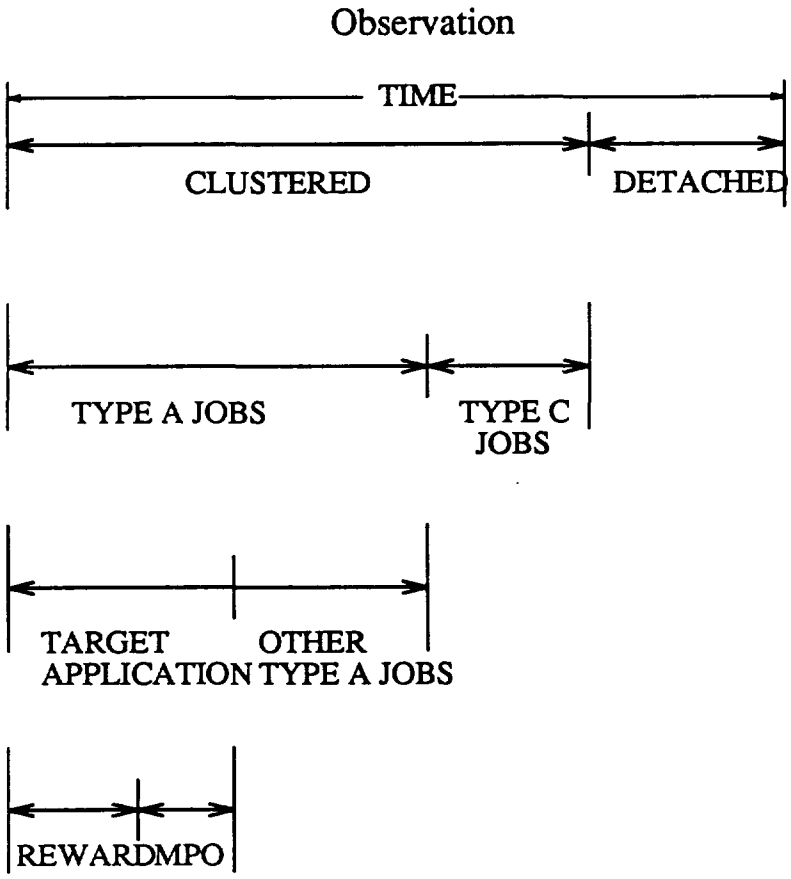



Figure 7.3 Reward Determination

Table 7.3
Rewards and Costs of Selected States

| Cluster Number | Reward (s) | Cost (s) |
|---|---|---|
| 1 | 0.188 | 6.85 |
| 7 | 0.351 | 6.62 |
| 20 | 0.495 | 9.19 |
| 26 | 0.471 | 4.97 |
| 33 | 0.148 | 6.65 |
| 49 | 0.864 | 5.29 |
| 52 | 1.880 | 5.58 |
| 68 | 0.686 | 5.45 |

model real, observed transitions. Each state has a reward associated with it that quantifies the processing resources available to an application if it were submitted to the system while in that state.

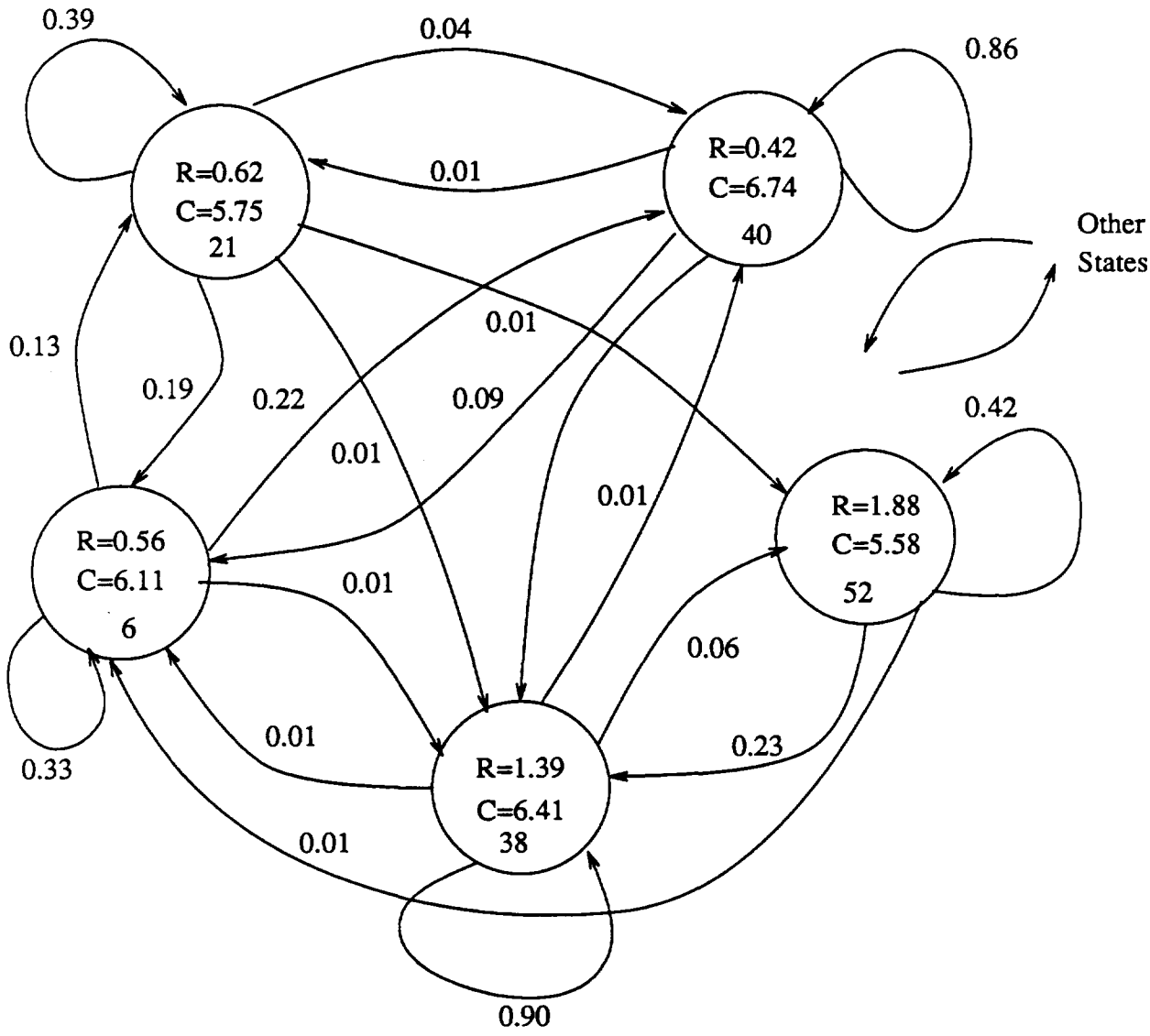



Figure 7.4 Constructed Markov Reward Model

### 7.2.5. Model solution

The reward/cost Markov model is capable of predicting the completion time distribution of a cluster type A job application running in real, multiprogrammed workloads. First, the amount of clustered CE time an application requires is determined. This can be done by measuring the application on a dedicated machine as was done with the target applications (Table 3.3).

With the reward determined, Monte Carlo simulation is used to solve the Markov model and generate completion times for the application. A random number is generated and used to select a starting state according to the starting state probability vector, $\pi(0)$. Random numbers are then generated to determine a path through the Markov model according to the transition matrix, $\Phi$. The chosen states are stepped through accumulating both reward and cost. When the amount of reward accumulated is equal to or greater than that required by the application, the simulation is complete and the cost accumulated is an estimation of the wall clock completion time for the application.

To present this procedure more formally, assume $\mathbf{X} = \{X_0, X_1, X_2, \cdots X_n, \cdots\}$ is a path through the Markov model generated by Monte Carlo simulation, where $X_n$ is the state of the Markov process after the $n^{th}$ transition. Let $Y_k$ and $Z_k$ be the accumulated reward and cost after the process has visited state $X_k$ (Equations (7.15) and (7.16)). If $t$ is the first time in which the accumulated reward is greater than the required reward (Equation (7.17)), then $Z_t$ the completion time of the application (Equation (7.18)).

$$Y_k = \sum_{i=0}^{k} r_{X_i} \tag{7.15}$$

$$Z_k = \sum_{i=0}^{k} c_{X_i} \tag{7.16}$$

$$t: \quad Y_{t-1} < \textit{Resources Needed} \leq Y_t \tag{7.17}$$

$$\textit{Completion Time} = Z_t \tag{1.18}$$

The above procedure provides a single measure of completion time for the given application. The distribution of completion time is estimated by repeating the above procedure to generate a large number of independent measures. For instance, Figure 7.5 shows the predicted completion time distribution of BDNA using the constructed model and 5000 Monte Carlo simulations. Recall that BDNA is a perfect benchmark requiring 131 s of dedicated machine time to complete (Table 3.3).

This illustrates the use of the methodology to predict the performance of an application. To predict the effect on performance of a system change, the reward function is modified to model the

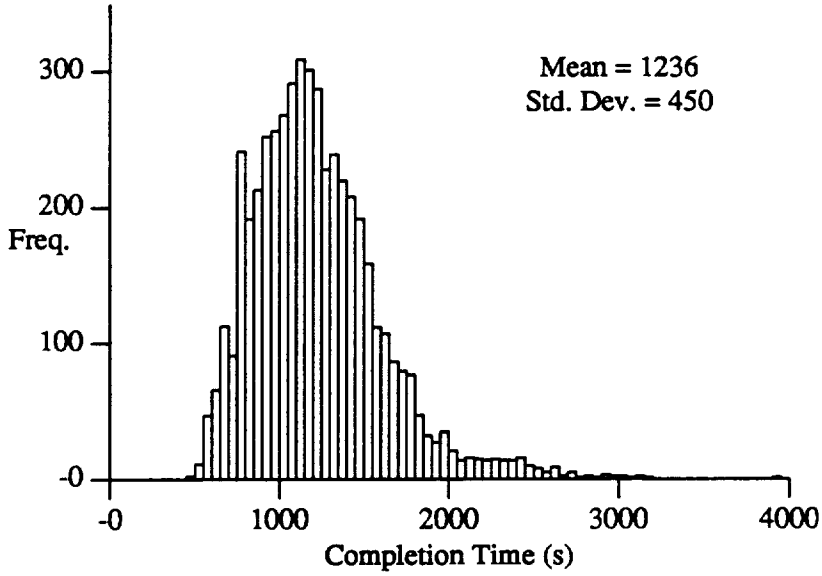



Figure 7.5 Predicted Completion Time: BDNA

appropriate change. The completion time distribution for an application is then determined with the new reward functions. The distribution computed under the modified reward is compared to that under the normal reward to evaluate the modification. The next chapter will use this procedure in an extensive study of Alliant schedulers and configurations.

# CHAPTER 8.

# MODEL USAGE — PREDICTING EFFECTS OF SYSTEM CHANGES

The previous chapter detailed the model building methodology and solution. That chapter hinted at the use of the model in predicting application performance and the effects of system design changes. In this chapter, the model constructed for the Alliant FX/80 will be used in an extensive study of possible scheduling paradigms; multiprogramming overhead will be revisisted; the performance effects of adding processors to the system will be investigated; and the constructed model will be critically evaluated. It is found that the model is flexible enough to model all of these situations. In addition, the evaluation uncovers a number of paradigms and configurations that will perform better than the current system setup.

## 8.1. Alliant FX/80 Scheduler

The model built for the Alliant FX/80 is used in this section to evaluate various scheduling policies and CE complex modes of operation. The majority of the paradigms and processor configurations studied are implementable on the current Alliant system. However, a few non-supported configurations are evaluated. The results are interesting on two levels. First, the evaluation of a real machine's possible scheduling paradigms in real workloads is always valuable. Second, and more importantly, the experiments demonstrate the power and flexibility of the model in evaluating and comparing different system configurations.

In Section 8.1.1, three different scheduling policies with the CEs in the dynamic mode are compared. In Section 8.1.2, the traditional complex mode is investigated. Section 8.1.3 evaluates the static detached mode. In Section 8.1.4, a new CE complex mode of operation is introduced and evaluated. Finally, in the last subsection, the results of the extensive scheduling evaluation are summarized. In all, 18 scheduling/CE-complex-modes are investigated (Table 8.1).

Table 8.1
Scheduling/CE-Complex-Modes

| Experiment ID | CE Complex Mode | Job Class | Scheduling Policy | Flo52 Completion Time (s) Mean | Std. Dev. |
|---|---|---|---|---|---|
| 1 | Dynamic | A | A | 839.7 | 342.8 |
| 2 | Dynamic | C | A | 413.3 | 130.0 |
| 3 | Dynamic | A or C | B | 716.3 | 234.7 |
| 4 | Dynamic | A | C | 808.5 | 326.6 |
| 5 | Dynamic | C | C | 475.8 | 142.5 |
| 6 | Traditional | A | A′ | 510.8 | 227.9 |
| 7 | Traditional | C | A′ | 259.8 | 88.6 |
| 8 | Traditional | A or C | B′ | 439.0 | 144.5 |
| 9 | Traditional | A | C′ | 490.8 | 208.8 |
| 10 | Traditional | C | C′ | 293.0 | 98.3 |
| 11 | 7/1 Static Detached | A or C | B′ | 478.6 | 156.0 |
| 12 | 6/2 Static Detached | A or C | B′ | 536.9 | 177.0 |
| 13 | 5/3 Static Detached | A or C | B′ | 617.4 | 195.3 |
| 14 | 4/4 Static Detached | A or C | B′ | 741.9 | 224.2 |
| 15 | 7/1 Dynamic Detached | A or C | B, B′ | 459.7 | 152.8 |
| 16 | 6/2 Dynamic Detached | A or C | B, B′ | 486.9 | 159.5 |
| 17 | 5/3 Dynamic Detached | A or C | B, B′ | 515.4 | 172.7 |
| 18 | 4/4 Dynamic Detached | A or C | B, B′ | 541.0 | 176.6 |

Each experiment in Table 8.1 consists of modeling the given change with a new reward function and solving the model for the completion time distribution of Flo52. Each distribution consists of 5,000 independent completion time simulations. The completion time distributions predicted for each system modification are then compared to evaluate the effects of the schedulers and complex configurations. The reward required by Flo52 was obtained by executing the application on a dedicated machine (Table 3.3).

### 8.1.1. Dynamic mode

The dynamic CE complex mode (Section 3.1.1) was evaluated under three scheduling policies: A, B, and C. Each policy can be easily implemented on the Alliant. Policy A (Table 3.1) was the scheduling policy in use while the data with which the model was built were collected. For this reason, it will be used as the basis for comparison. This policy gives type C jobs longer time quanta than type A jobs. Policy B (Table 8.2) groups type A and type C jobs into the same class, effectively

eliminating any sense of class. Policy C (Table 8.3) enforces the class structure but removes the priority given to class C jobs (equal time quanta).

The reward functions used to model policies A, B, and C are given by Equations (7.13), (8.1), and (8.2), respectively. The superscript on the reward term, $R_i$, indicates the experiment ID number. The reward function for policy B uses the number of type C cluster jobs ($CLC_i^{\zeta}$) instead of the percentage of time for which there is at least one type C cluster job ($\%CLC_i$), because all clusters jobs (type A and C) are scheduled in a round-robin fashion to the same time block. It is no longer true that a time block is reserved solely for type C jobs as it is under Policy A (Level 2, Table 3.1). Policy C's reward function modifies the evaluation of $NORMC$ to $NORMC2$ to reflect the equal time quanta given to type A and C jobs.

$$R_i^{exp3} = (TIME_i^{\zeta}) \times CLUSP_i^{\zeta} \times \left(\frac{1}{1 + CLA_i^{\zeta} + CLC_i^{\zeta}}\right) \times (1 - MPO_i^{\zeta}) \tag{8.1}$$

$$R_i^{exp4} = (TIME_i^{\zeta}) \times CLUSP_i^{\zeta} \times (1 - (\%CLC_i^{\zeta} \times NORMC2_i)) \times \left(\frac{1}{1 + CLA_i^{\zeta}}\right) \times (1 - MPO_i^{\zeta}) \tag{8.2}$$

$$NORMC2_i = \frac{1}{CLUSP_i^{\zeta}} \times \left[\left(\frac{7}{13} \times \frac{3.5}{7}\right) + \left(\left(CLUSP_i^{\zeta} - \frac{7}{13}\right) \times \frac{1.5}{3}\right)\right]$$

Table 8.2
Scheduling Policy B

| Lev. | Quant. | Choice 1 | Choice 2 | Choice 3 |
|---|---|---|---|---|
| 1 | 300 ms | cluster | IP/CE | CE |
| 2 | 400 ms | cluster | IP/CE | CE |
| 3 | 200 ms | CE | IP/CE | cluster |
| 4 | 200 ms | CE | IP/CE | cluster |
| 5 | 200 ms | IP/CE | CE | cluster |

Table 8.3
Scheduling Policy C

| Level | Quantum | Choice 1 | Choice 2 | Choice 3 | Choice 4 | Choice 5 |
|---|---|---|---|---|---|---|
| 1 | 350 ms | cluster (A) | cluster (C) | IP/CE | CE (A) | CE (C) |
| 2 | 350 ms | cluster (C) | cluster (A) | IP/CE | CE (C) | CE (A) |
| 3 | 150 ms | CE (C) | CE (A) | IP/CE | cluster (C) | cluster (A) |
| 4 | 150 ms | CE (A) | CE (C) | IP/CE | cluster (A) | cluster (C) |
| 5 | 150 ms | IP/CE | CE (C) | CE (A) | cluster (C) | cluster (A) |
| 6 | 150 ms | IP/CE | CE (A) | CE (C) | cluster (A) | cluster (C) |

Completion time distributions for Flo52 scheduled as both type A and type C cluster jobs were determined for scheduling policies A and C. For policy B the type of job is irrelevant thus only one distribution was determined. Type C cluster jobs were modeled by modifying the reward function accordingly. For instance, the reward for a type C job under scheduling policy A is shown by Equation (8.3). The reward function for a type C cluster job is arrived at symmetrically to the function for a type A job (Figure 7.4). However, in the second line of the figure, the right side of the time period (type C jobs) is used instead of the left side.

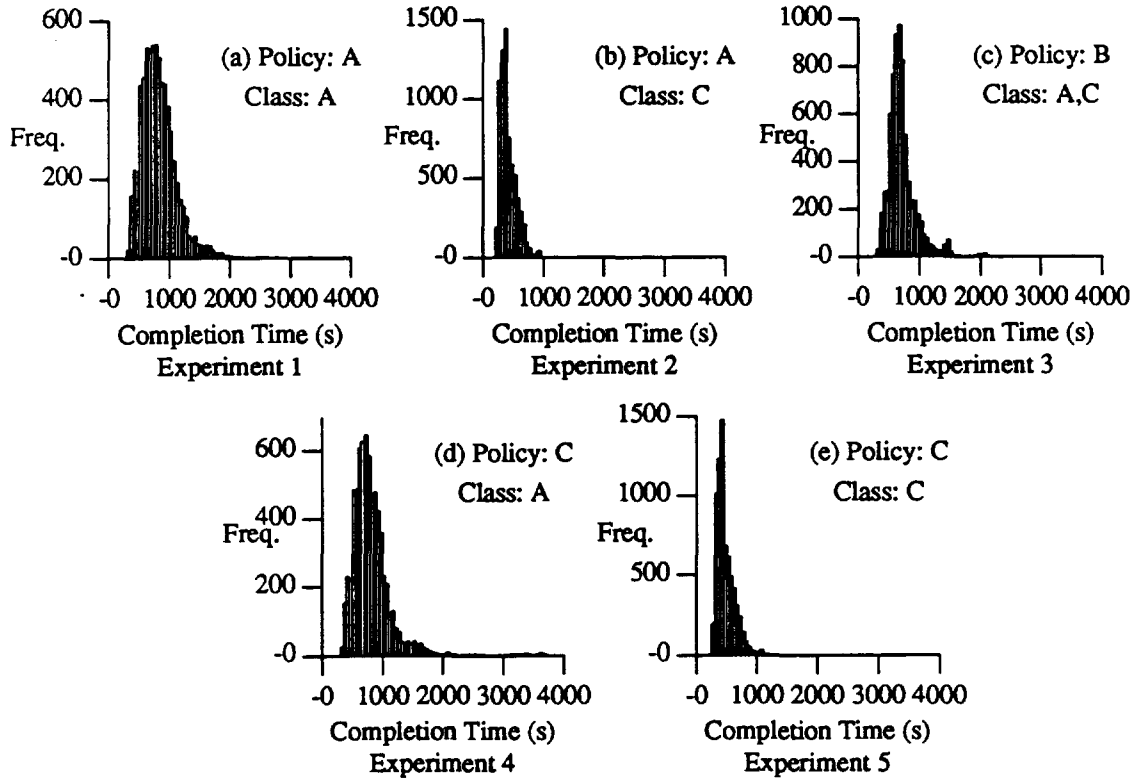
$$R_i^{exp2} = (TIME_i^C) \times CLUSP_i^C \times (1 - (\%CLA_i^C \times NORMA_i)) \times (\frac{1}{1 + CLC_i^C}) \times (1 - MPO_i^C) \qquad (8.3)$$

$$NORMA_i = \frac{1}{CLUSP_i^C} \times [(\frac{7}{13} \times \frac{3}{7}) + ((CLUSP_i^C - \frac{7}{13}) \times \frac{1}{3})]$$

The distributions of completion times for Flo52 under the different schedulers are shown in Figures 8.1(a) - (e). Only one graph is shown for policy B because the completion time distribution for a type A job is identical to that of a type C job (no class system enforced). The means and standard deviations for the experiments are shown in Table 8.1 (Experiments 1-5).

The results show that policy A is the worst of the three for cluster A jobs (Experiment 1) and the best for cluster C jobs (Experiment 2). This is due to the different size time quanta granted to the different job classes. Removing the slight (50 ms) time quantum difference between type A and C jobs (Experiments 4 - 5) reduces the average completion time of type A cluster jobs by only 3.7%, while increasing the average completion time of type C cluster jobs by 15.1%. Therefore, this policy is not recommended. Under policy B (Experiment 3), type A cluster jobs finish, on average, 14.7% more quickly than under Policy A; however, type C cluster jobs take an average of 73.3% longer. The large difference in the vicissitudes is attributable to the abundance of type A cluster jobs in the workloads studied.

Consulting the distributions, it is seen that under policies A and C, a small number of simulations placed the completion time of Flo52— when classified as a type A cluster job




Figures 8.1 Completion Time: Dynamic Mode

(Experiments 1,4)— well over 2000 s (some as high as 4000 s). The possibility of extremely long completion times is a weakness of these scheduling policies — a weakness not found under policy B (Experiment 3). However, policies A and C have the advantage of nearly guaranteeing that Flo52 will finish within 1000 s if submitted as a type C job (Experiments 2,5). This type of information is extremely useful for real-time systems evaluation.

As a final point, it should be noted that all three scheduling policies treat detached jobs identically. Therefore the performance of detached jobs is assumed to be identical under the different policies.

### 8.1.2. Traditional complex mode

In the traditional complex mode the CEs remain clustered at all times and are used as a single resource to multiprogram cluster jobs (Section 3.1.1). All other jobs are executed by the IPs. Due to

the abundance of IP/CE jobs in the real workloads measured, this mode could result in a processing bottleneck at the IPs. Therefore, the traditional complex mode is eliminated from serious consideration. However, for the sake of completeness, the traditional mode is studied in this subsection.

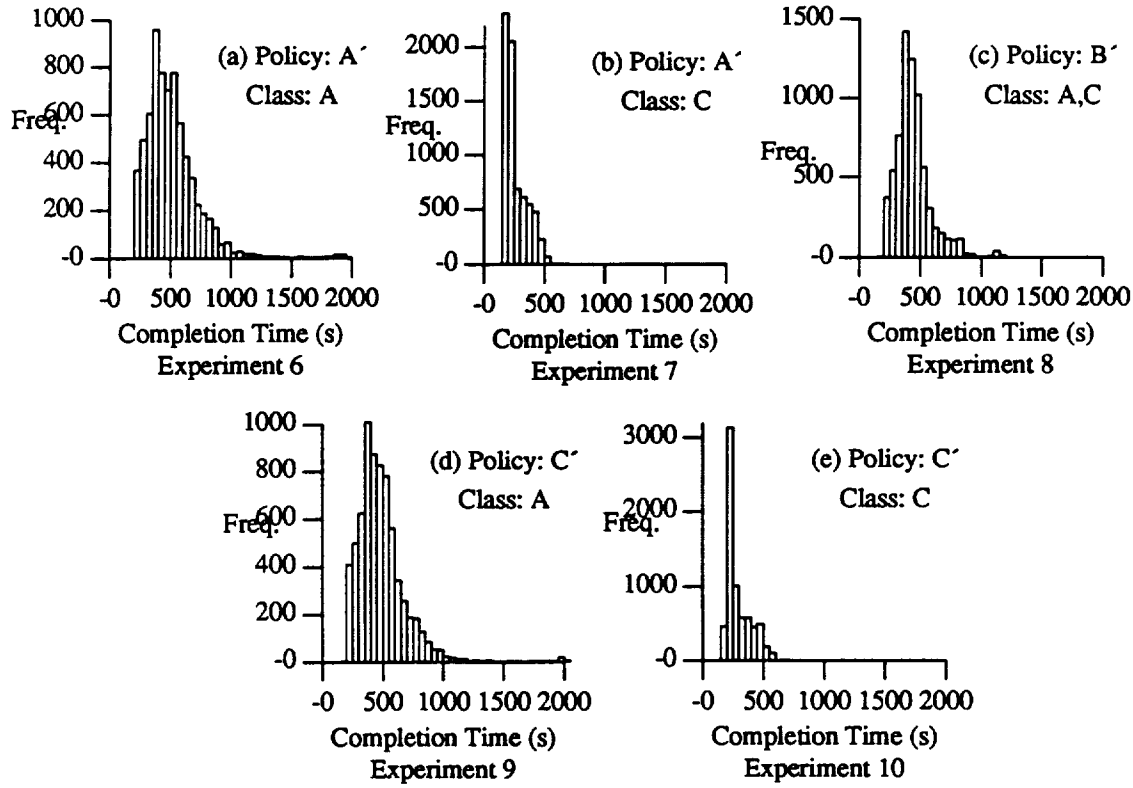
Three scheduling policies (A′, B′, and C′), resembling the three policies introduced in the last subsection, were evaluated. The policies are completely described by the top two levels of the corresponding dynamic scheduling policies. For instance, policy A′ is simply levels 1 and 2 of Table 3.1. Equation (8.4) shows the reward function for type A cluster jobs under policy A′. The $CLUSP^C$ term is not used in this reward function because the CE complex is always clustered in the traditional mode. In other words, the division shown by the first line in Figure 7.4 is not necessary.

$$R_i^{exp6} = (TIME_i^C) \times (1 - (\%CLC_i^C \times \frac{4}{7})) \times (\frac{1}{1 + CLA_i^C}) \times (1 - MPO_i^C) \tag{8.4}$$

The completion time frequency distributions for Flo52 executing under policies A′, B′, and C′ as both type A and type C cluster jobs are shown in Figures 8.2(a) - (e). The means and standard deviations for the traditional mode evaluations are summarized in Table 8.1 (Experiments 6-10).

Not surprisingly, because there is no research sharing between clustered and detached jobs, cluster jobs execute more quickly on a machine in traditional mode than on a machine in dynamic mode. On average, type A cluster jobs execute 39% faster on the traditional complex under policy A′ than on the dynamic cluster under policy A (Experiments 1,6). The standard deviation of completion times is also much lower for all policies on the traditional mode complex. This indicates that cluster job response times are more predictable when executed on CEs in the traditional mode.

As with the dynamic mode, the best scheduling policy for type A jobs in the traditional mode consists of removing the class distinction (Experiment 8); again, this is the worst policy for type C cluster jobs. Comparing policies A′ and C′ (Experiments 6,7,9,10) demonstrates that leaving the class distinction, but providing equal quanta to both A and C cluster jobs, does not significantly improve type A job performance.




Figures 8.2 Completion Times: Traditional Mode

### 8.1.3. Static detached mode

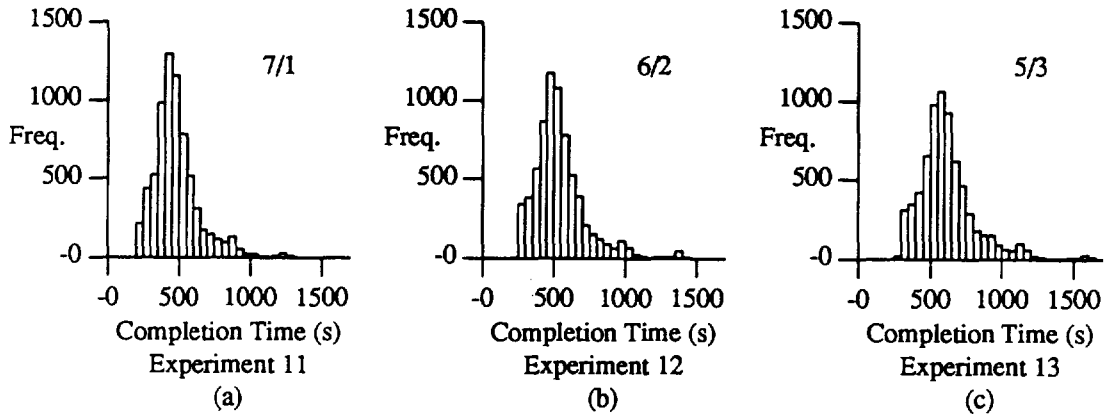
In the static detached mode a fixed number of CEs are clustered executing parallel jobs, while the rest of the CEs remain detached. This subsection investigates four static detached configurations under scheduling policy B´. Scheduling policy B´ ensures that the completion time distribution will be unaffected by the class (A or C) of Flo52.

A static detached complex with seven CEs clustered and one CE detached will be called a 7/1 static detached complex. From the cluster jobs view, the machine in the 7/1 static detached mode appears as it does in the traditional mode, but with one less processor. Therefore, performance will be degraded on sections of an application that require all eight processors. To model this, the reward needed for sections of Flo52 requiring all eight processors was increased by a factor of $\frac{8}{7}$, while the reward needed for sections requiring seven or fewer processors was left unchanged.

Therefore, by increasing the reward required by Flo52, and using the reward function for scheduling policy B′, the completion time distribution of Flo52 executing on a 7/1 static detached complex was estimated. Similar analysis was performed for 6/2, 5/3, and 4/4 static detached configurations. Selected results are shown in Figures 8.3(a) - (c) and Table 8.1 (Experiments 11-14).

The figures show that as the number of statically clustered CEs increases, the completion time of clustered jobs decreases. Comparing Experiments 11 - 14 with Experiment 3, it is seen that the only case in which policy B in dynamic mode performs better than the static detached mode is for the 4/4 static detached configuration. Because it does not outperform the current system environment, the 4/4 static detached configuration is eliminated from further consideration.

Further analysis shows that the 7/1, 6/2, and 5/3 static detached configurations under scheduler B′ (Experiments 11,12,13) execute clustered jobs faster than the dynamic configuration under scheduler B (Experiment 3). It must be remembered though that the model can only predict the performance of clustered jobs. The performance of these jobs must be considered to make the analysis complete. For the 7/1 static detached mode there remains only one CE (along with the 6 IPs) to handle all of the IP/CE and CE jobs. The response time of these jobs may suffer under this configuration. Preliminary analysis indicates that 2 or 3 dedicated CEs along with the 6 IPs are powerful enough to meet the encountered IP/CE and CE job demands. This information, along with




Figures 8.3 Completion Times: Static Detached Mode

the fact that both the 6/2 and the 5/3 configurations perform better on cluster jobs than the dynamic mode, indicates that these configurations should be implemented or evaluated further.

#### 8.1.4. Dynamic detached mode

All scheduling paradigms and processor configurations investigated in the last subsections are supported by the Alliant FX/80. Although prohibitively time consuming, the evaluation could have been accomplished by implementing each change and measuring the corresponding performance. In this subsection, the power of the model will be demonstrated by evaluating a CE complex mode which is not supported by the current Alliant operating system. The mode is called the *dynamic detached mode*. It is a straightforward mix of the dynamic and the static detached modes.

In the dynamic detached mode, the CEs dynamically switch between two configurations: *clustered* and *semi-detached*. In the clustered configuration, all eight CEs are applied to the execution of cluster jobs identically to the clustered configuration in the dynamic mode. In the semi-detached configuration, the CE complex performs as it does in the static detached mode— a fixed number of CEs remain clustered and multiprogram parallel jobs, while the remaining CEs concurrently execute detached (serial) jobs. Notation similar to that introduced in the previous subsection is used to describe the number of CEs that remain clustered in the semi-detached configuration. Figure 8.4 illustrates the 4/4 dynamic detached mode.

Equation (8.5) shows the reward for cluster jobs executing in this environment. The equation calculates separately and then adds the reward for the CEs in the clustered configuration ($CLUSP_i^C$ fraction of the time) and the reward for the CEs in the semi-detached configuration. The term $\frac{\#CE}{8}$ accounts for there being fewer than 8 processors available to the application while in the semi-detached configuration. The term $FILL_i$ accounts for multiprogramming overhead, competing jobs, and the time of the observation.

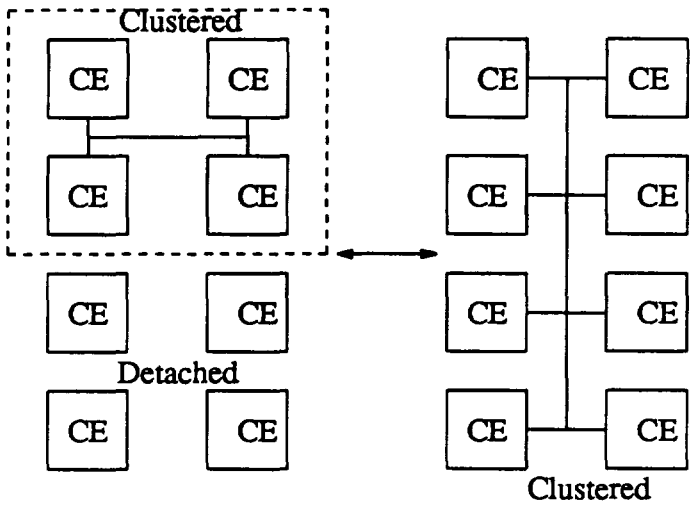



Figure 8.4 4/4 Dynamic Detached Mode

$$R_i^{exp\,15\text{–}18} = [(1 - CLUSP_i^C) \times \frac{\#CE}{8} \times FILL_i] + [CLUSP_i^C \times FILL_i] \qquad (8.5)$$

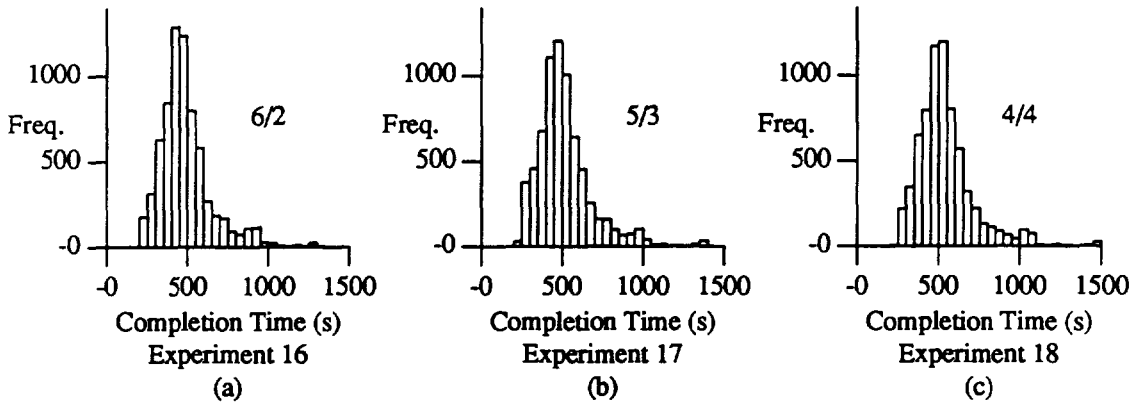
$$FILL_i = (TIME_i^C) \times (\frac{1}{1 + CLA_i^C + CLC_i^C}) \times (1 - MPO_i^C)$$

#CE = CEs executing parallel jobs while semi-detached

Four dynamic detached modes were investigated: 7/1, 6/2, 5/3, and 4/4. Selected results are shown in Figures 8.5(a) - (c) and Table 8.1 (Experiments 15-18).

The performance of the dynamic detached mode is bounded from above by the performance of the traditional mode, and from below by the performance of the dynamic mode. As can be seen, the dynamic detached mode executes cluster jobs more efficiently than the corresponding configurations in the static detached mode (Experiments 11 and 15; 12 and 16; 13 and 17; 14 and 18). In addition, preliminary analysis indicates that the 5/3 and 4/4 dynamic detached modes provide adequate processing power for the IP/CE and CE jobs encountered in the workloads measured; the 7/1 configuration would lead to a bottleneck at the IPs.

Therefore, the dynamic detached mode, while currently unavailable, can be an effective mode which should be investigated further. The model predicts that for the workloads measured, the 5/3




Figures 8.5 Completion Times: Dynamic Detached Mode

dynamic detached mode (Experiment 17) will execute cluster jobs more efficiently than both the dynamic mode (Experiments 1, 3, and 4) and the 6/2 static detached mode (Experiment 12).

#### 8.1.5. Scheduling summary

In the last subsections, available and speculative scheduling paradigms and processor configurations were investigated. Once the model was constructed, the cost and time required to evaluate all of the policies were minimal. The model allowed the policies to be evaluated not only for a single application (although a single completion time distribution is used) but for a class of applications executing in real measured workloads. In essence, the analysis evaluates the system design change with respect to all of the workloads which were measured.

The results demonstrated that the CE configuration and scheduling policy could be modified to improve performance for parallel jobs without degrading the performance of serial jobs. First, if the CE complex remained in the dynamic mode, the study indicated that a new scheduling policy (policy B) would provide more consistent service to all parallel jobs. If the CE complex were operated in the static detached mode, the investigation suggested that a 6/2 or 5/3 configuration would provide better service to the parallel jobs.

Finally, a new mode of operation (dynamic detached mode) was introduced and shown to outperform the current configuration and also the static detached configuration. The investigation suggested further experimentation with either a 4/4 or 5/3 dynamic detached mode.

### 8.2. Predicting Effect of Multiprogramming Overhead — Alliant FX/80

In Section 5.4.1 the percentage of processing power consumed by multiprogramming overhead in real workloads was quantified. In this section, MPO is not approached from a processor utilization perspective but from an application performance degradation point of view. In other words, in this section, the increase in the completion time of an application due to MPO is quantified. Also, the effects of reducing MPO are quantified. The results are useful in determining whether the reduction of MPO is a cost effective goal.

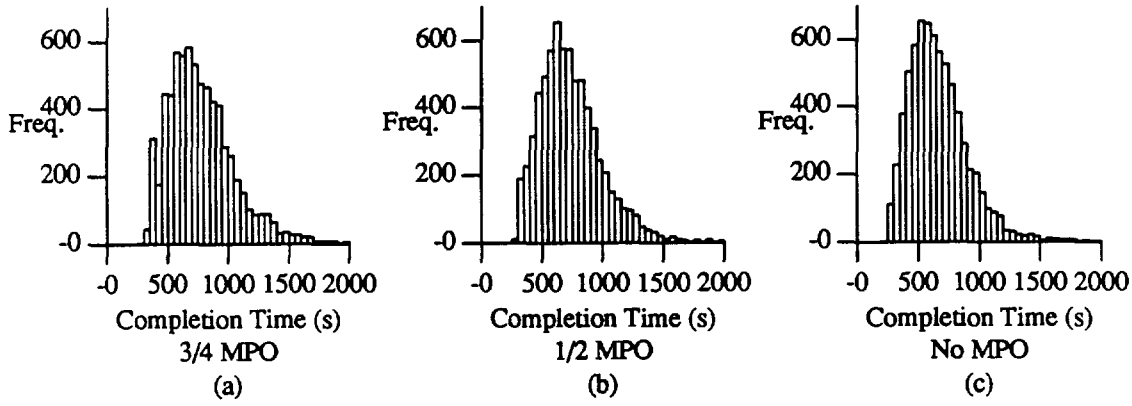
The completion time distribution of a parallel application was first predicted in the workloads without any system changes (full MPO). The reward function was then modified to model a 25% reduction in MPO (Equation (8.6)), a 50% reduction in MPO (equation not shown), and the ideal case of complete MPO removal (Equation (8.7)).

$$R_i = (TIME_i^C) \times CLUSP_i^C \times (1 - (\%CLC_i^C \times NORMC_i)) \times (\frac{1}{1 + CLA_i^C}) \times (1 - \frac{MPO_i^C}{4}) \qquad (8.6)$$

$$R_i = (TIME_i^C) \times CLUSP_i^C \times (1 - (\%CLC_i^C \times NORMC_i)) \times (\frac{1}{1 + CLA_i^C}) \qquad (8.7)$$

The completion time distribution of Flo52 in normal workloads was previously determined and shown in Figure 8.1(a). The distributions with the MPO reduced 25%, 50%, and completely eliminated are shown in Figures 8.6(a) - (c). The figures show only completion times less than 2000 s because there were few times greater than 2000 s and their inclusion reduced the figures' precision. The means and standard deviations for the distributions are shown in Table 8.4.

By reducing the MPO by 25%, the average completion time is reduced by 6%; reducing the MPO by 50%, reduces the completion time by 11%; completely eliminating the MPO reduces the average completion time by 20%. Notice that the standard deviation is not substantially different




Figures 8.6 Reduction of Multiprogramming Overhead

Table 8.4
Reducing Multiprogramming Overhead

| % MPO | Flo52 Completion Time | |
|---|---|---|
| | Mean | Std. Dev. |
| 100 | 839.7 | 342.8 |
| 75 | 792.7 | 332.8 |
| 50 | 746.8 | 291.8 |
| 0 | 675.8 | 270.4 |

between the 1/2 MPO and the no MPO situations. In other words, once half of the MPO is eliminated, removing more will improve the mean completion time of parallel applications, but will not substantially reduce the wide range of completion times.

### 8.3. Additional Processors: Alliant FX/80

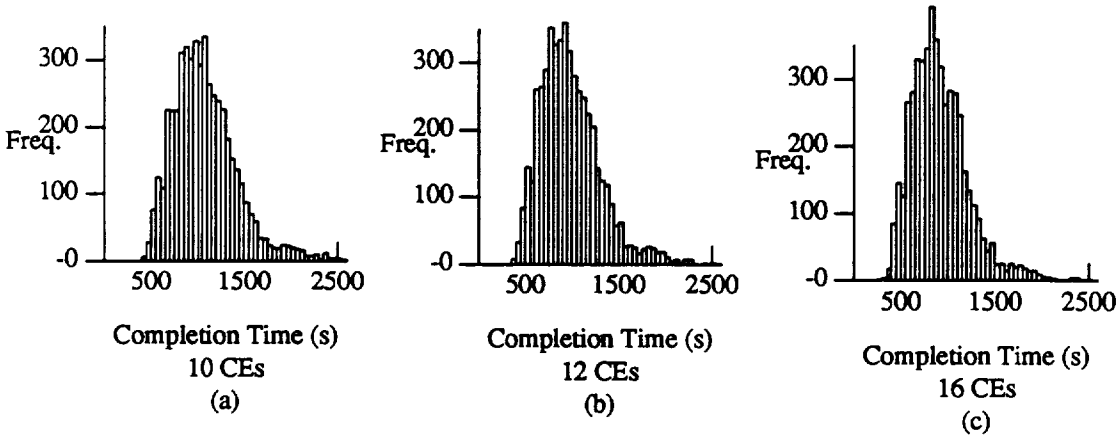
In this section, the performance improvement obtained by adding more CEs to the Alliant is investigated. It is assumed that N additional CEs are added to the Alliant FX/80 and that the CEs remain in the dynamic mode and the jobs are scheduled with Policy A (Table 3.1). Now, for the sake of variety, BDNA will be used as the target application used to gauge the performance improvement with extra processors.

For sections of BDNA where all 8 processors are employed, a linear speedup of $\frac{8+N}{8}$ is assumed. This assumption is valid because these section of codes consist of self-scheduled iterations

of a loop. Sections of BDNA using less than 8 processors will not be sped up when more processors are added to the system. For BDNA, it is found from executing on an eight-processor dedicated machine that 60.5% of the application's execution uses all eight processors. The reward needed was changed according to these assumptions and the model was used to simulate 5000 runs of BDNA. Note that the model solves for the completion time of BDNA executing with the real workloads on a machine with 8 + N CEs.

Systems with N = 2, 4, and 8 extra processors were modeled. The completion time frequency distributions for the extra processor machines are shown in Figure 8.7(a) - (c). The completion time of BDNA on the 8 processor machine is shown in Figure 7.6. The mean and standard deviations for the completion times are shown in Table 8.5.

The results show that for BDNA adding two extra processors to the system will decrease the average completion time by 12%. After this, adding more processors will not decrease the completion




Figures 8.7 Additional Processors

Table 8.5
Additional Processing Elements

| # CEs | BDNA Completion Time Mean | Std. Dev. | % improvement |
|---|---|---|---|
| 8 | 1236 | 450 | - |
| 10 | 1091 | 392 | 12% |
| 12 | 1001 | 395 | 19% |
| 16 | 933 | 381 | 25% |

time as quickly. Obviously, this is a consequence of Ahmdal's law. Also notice that the first two extra processors decrease the standard deviation of completion time for BDNA, but adding more than two does not affect it significantly.

This indicates that adding processors may not be the best way to increase the performance of parallel jobs on the Alliant FX/80 for the workloads studied. In fact, extra processors (past the first two) used in the dynamic mode show negligible performance improvement for real parallel jobs. Instead, extra processors would be better used by changing the CE complex to a static detached mode and using the extra processors to handle all CE and IP/CE jobs. In this way, the original eight CEs would remain clustered at all times, and performance equivalent to the traditional mode (Section 8.1.2) would be achieved.

### 8.4. Model Usage Evaluation

As demonstrated in this chapter, a major strength of the constructed model is its flexibility. All scheduling/CE-complex-mode configurations encountered could be modeled. The model was capable of accounting for changes in multiprogramming overhead and performance with an increased number of processors. Part of the model's flexibility stems from the different methods in which system modifications can be modeled. For instance, changes can be modeled by modifying the reward function (scheduling paradigms), or by modifying the reward requested (extra processors). If the execution of applications in different workloads needs to be predicted, the starting state probabilities can be modified to model the new workloads. This provides a function of the model which will be illustrated in the next chapter.

In addition to the wide range of system modifications which can be modeled, the modeling technique is valuable because it provides more than a single point measurement for comparison; the entire distribution of completion time is provided. The probability that an application will finish by a given time in the workloads provides not only information on the specific application that is being

modeled, but also inherently evaluates how entire workloads will execute under the system design change.

The previous sections have illustrated some of the results available through the completion time distribution. For instance, the peaks in the frequency distributions indicate the most common completion time for an application. The standard deviation indicates how predictably a job will behave under a policy or configuration. This was used in Section 8.2 to see that reducing multiprogramming overhead past a certain point will not improve the predictability of an application's completion time. By analyzing the distribution tails, worst-case behavior can be gauged. This was useful when comparing scheduling paradigms while the CEs were in the dynamic mode (Section 8.1.1). It was found that Policy A occasionally reported terrible performance, while Policy B was much more consistent. The distribution also allows the probability that an application will finish by a given deadline to be estimated. This type of information is invaluable for real-time systems. Of course, single point comparisons can still be made by using the distributions means. However, the entire completion time distribution provides significantly more detail than the means alone.

# CHAPTER 9.

# MODEL VALIDATION

It is often the case that analytical models are built, used to predict system behavior, and never validated with empirical data. This is true for a number of reasons. Frequently models predict system or system design changes which are not available for measurement. Other times, accurate measurements, while available, are hard to obtain and thus are not gathered. Frequently, the models built are based on such broad assumptions of workload, that there do not exist empirical data which can validate them. For instance, it is hard to find workloads following exponential arrival times, and thus models based on these assumptions are difficult to validate with measurements from real machines. In cases in which validation is not performed, the predictions made by the models can be trusted only as far as the assumptions in the modeling technique.

When validation is performed, it is normally a validation of an analytical model with simulation results. While this is beneficial, if assumptions made by the analytical model are also made by the simulation, the validation loses some of its persuasiveness. It is a tenet of this thesis that whenever possible, predictions made by models should be validated with empirical data from real workloads.

In this section the results of four experiments that validate the model constructed for the Alliant FX/80 with empirical data are presented. The first two validations use the completion times of the target applications to validate the model. The conclusion from these tests is that the model accurately represents the workloads measured. The third validation uses the model to predict correctly the effect of operating the system with only seven CEs. In the final experiment, the performance of an application in new workloads under a new scheduling paradigm is successfully predicted. The last two experiments validate the predictive abilities of the constructed model.

The effect of the number of clusters chosen to represent the workload on the accuracy of predictions is then evaluated. The criterion proposed in Chapter 7 is shown to be valid.

### 9.1. Validation A: Predicting CT of Target Applications, 1

The first phase of validation tests how well the constructed Markov model represents the workload measured. The Markov model was slightly modified to include an absorption state corresponding to application completion. The Markov model was then solved for an average number of transitions until absorption (application completion). If the model accurately represented the workload, the average number of states visited before absorption multiplied by the average time spent in a state would be close to the mean completion time of the 100 target applications executed while the data were collected. In other words, the average predicted completion time of the target applications would be close to the actual completion time.

The new Markov model had 73 states in its state space: the 72 system/workload states plus an absorption state (state 73). The new transition matrix ($\Phi'$) was calculated with an additional 100 transitions: one for each target application completion (Equation (9.1)). The last observation of each target application ($O_{k,OBS(k)}$) was assumed to traverse to the absorption state.

$$P'_{i,j} = \frac{\sum_{k=1}^{100} \sum_{l=1}^{OBS(k)-1} IND\,[(O_{k,l} \in C_i) \wedge (O_{k,l+1} \in C_j)]}{\sum_{k=1}^{100} \sum_{l=1}^{OBS(k)} IND\,[O_{k,l} \in C_i]} \qquad 1 \le i,j \le 72$$

$$P'_{i,73} = \frac{\sum_{k=1}^{100} IND\,[O_{k,OBS(k)} \in C_i]}{\sum_{k=1}^{100} \sum_{l=1}^{OBS(k)} IND\,[O_{k,l} \in C_i]} \qquad 1 \le i \le 72 \qquad (9.1)$$

$$P'_{73,i} = \begin{cases} 0 & 1 \le i \le 72 \\ 1 & i = 73 \end{cases}$$

To solve the Markov model, the row and column corresponding to the absorption state were removed from the transition matrix to form the 72x72 matrix **Q**. Equation (9.2) determines the matrix **M** which is made up of elements, $m_{i,j}$, corresponding to the expected number of visits to state j before absorption, given the process started in state i. Expected completion was then estimated using Equation (9.3).

$$\mathbf{M} = (\mathbf{I} - \mathbf{Q})^{-1} \qquad (9.2)$$

$$E[Completion\ \ Time] = 6.094 \times [\pi_1(0)\ \ \pi_2(0)\ \ \pi_3(0)\ ....\ \pi_{20}(0)] \times \begin{bmatrix} \sum_{j=1}^{72} m_{1,j} \\ \sum_{j=1}^{72} m_{2,j} \\ . \\ . \\ \sum_{j=1}^{72} m_{72,j} \end{bmatrix} \quad (9.3)$$

The above evaluation determined that the expected completion time of the applications used to build the model was 1306 s. The actual mean of the 100 target applications was 1303 s. This indicates that the Markov model successfully captured the states and transitions of the workloads measured.

### 9.2. Validation B: Predicting CT of Target Applications, 2

The last section validated the Markov model. In this section, the Markov model as well as the reward and cost function will be validated.

Recall that the model was constructed from measurements obtained during the execution of Dyfesm and Flo52 35 times, and Track 30 times. The frequency distribution of completion time for these 100 executions is shown in Figure 9.1(a). If the model accurately captured the workloads measured, then it should be able to predict the completion time distribution of the target applications. Towards this end, model simulation was performed to generate completion times for 4728 executions each of Dyfesm and Flo52, and 4054 executions of Track (the 35:3 5:30 ratio). The frequency distribution of completion times for these simulations is shown in Figure 9.1(b).

The figures show that the distribution of predicted completion time is extremely close to the actual distribution. Both distributions rise to major peaks at 600 - 750 s and then taper to a second plateau beginning around 1200 s. This second shelf remains fairly constant until 2100 s at which point it steadily diminishes until there are only sporadic occurrences after 2500 s. Using common statistical tests [74], the hypothesis that the means of the two distributions are equal cannot be rejected even with

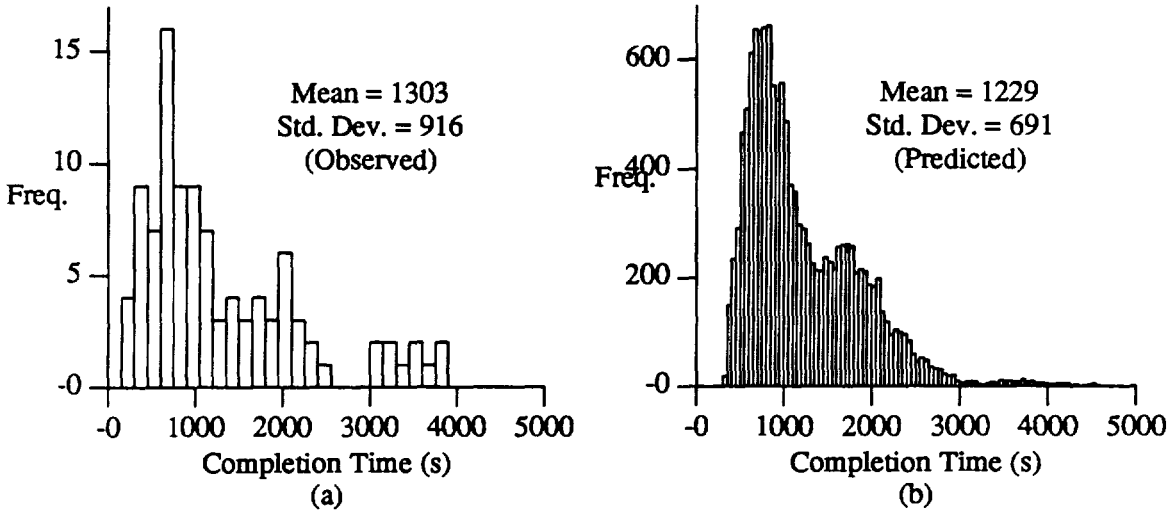



Figure 9.1 Observed and Predicted Target Application Completion Time

a level of significance as high as 0.2. In other words, the mean of the predicted distribution, 1229 s, is statistically equal to the actual mean of 1303 s. The model's capability to reproduce the completion times of the target applications validates the Markov model and the reward function.

### 9.3. Validation C: Predicting the Effect of Fewer Processors

Validations A and B illustrated that the model successfully captured the workload that was measured. The next two experiments will show that the model can predict the performance of applications executing in workloads similar to those measured on systems with different configurations. For the experiment of this subsection, a single CE on the Alliant FX/80 was removed leaving the cluster with only seven processors. The machine continued to handle its normal activities, but with one less CE. Flo52 was executed 48 times in random, uncontrolled workloads over a two-week period on the seven-processor Alliant (Figure 9.2(a)).

The model was used to predict the completion time distribution of Flo52 running on a seven-processor machine. Recall that Flo52 requires 88 s of clustered CE time to complete. It was determined that 57.5 s (65%) of this time requires all eight CEs. Flo52 running on seven processors was modeled by increasing the reward needed for sections of code requiring all eight processors by a

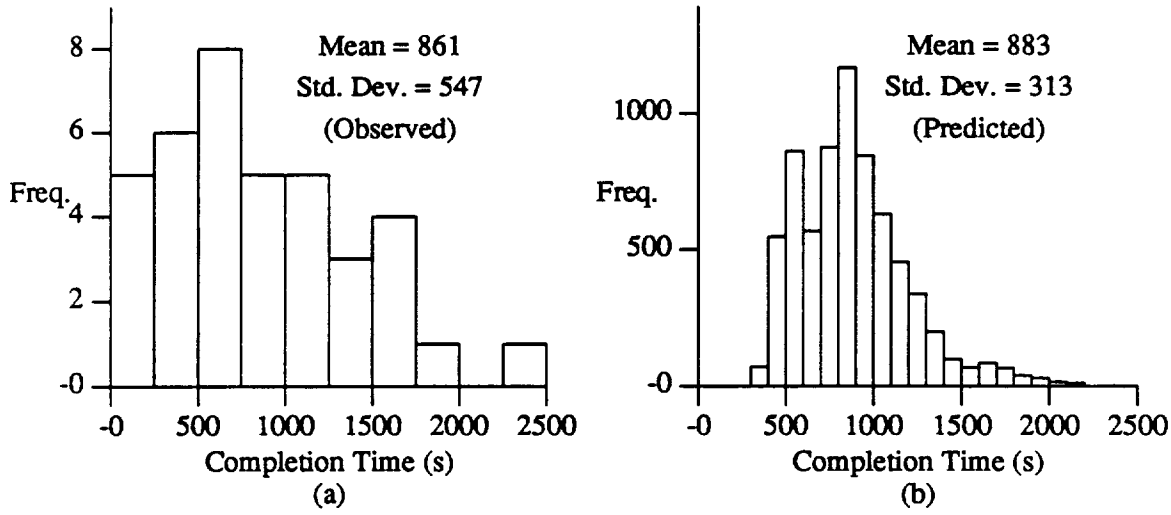



Figure 9.2 Observed and Predicted Completion Time on a Seven Processor Machine

factor of $\frac{8}{7}$ and leaving the reward needed for sections requiring seven or fewer processors unchanged. In addition to increasing the reward required, the starting state probabilities were modified to model the actual 48 starting states of the application executions. With these changes, 5000 completion times were generated. The corresponding frequency distribution is shown in Figure 9.2(b).

The figures show that the model was able to predict the real workload performance of Flo52 on a seven-processor machine. Both distributions show peaks around 750 s, with a tail disappearing around 2000 s. In addition, neither distribution shows values above 2500 s.

However, the model failed to predict the completion times under 350 s. This was caused by low usage workloads present frequently during the test but rarely during the creation of the model. This highlights an important aspect of the modeling methodology; the model does not, nor should it, predict performance in workloads drastically different from those modeled. It should be emphasized, though, that the method models a variety of workloads and is robust enough to predict completion times accurately in workloads that are similar to those modeled.

The average predicted completion time for the seven-processor machine was 883 s. The observed empirical average was 861 s. Using common statistical techniques, the hypothesis that the two population means were equal could not be rejected even with high significance levels. This may be due to the high variances in completion time. With this in mind, the data of this experiment still validate both the model and the methodology used to build the model.

### 9.4. Validation D: Predicting a Scheduling Modification

The ability of the model to predict the effects of a scheduling modification is investigated in this section by implementing scheduling policy B (Table 8.2) on the Alliant FX/80 with the CEs in the dynamic mode. The machine continued to be used by the same community of users for the same purposes but under the new scheduling policy. The Perfect benchmark BDNA (Table 3.3), which is an application of the modeled class (computationally bound, parallel application), was executed 66 times at randomly selected times over a 6 week period in the modified environment. The measured completion time frequency distribution for the 66 runs is shown in Figure 9.3(a).

The model was used to predict the completion time distribution of the 66 executions of BDNA under the new scheduling algorithm. The reward function for each state was modified to model the new scheduling policy (Equation (8.1)) In addition, the Markov model's starting probabilities were changed to reflect the 66 starting states of the BDNA application executions. With these modifications, Monte Carlo simulation was used to generate 7000 completion times of BDNA. The predicted completion time frequency distribution is shown in Figure 9.3(b).

The figures show that the model was able to predict the real workload performance for a completely new application (BDNA) under the revised scheduling policy. Both frequency distributions show their lowest values around 500 s and climb to a peak at 1000 s; the curves diminish to nothingness around 2000 s. The average predicted completion time under the new scheduler was 1087 s compared to the empirical average of 1069 s. Using common statistical techniques, the

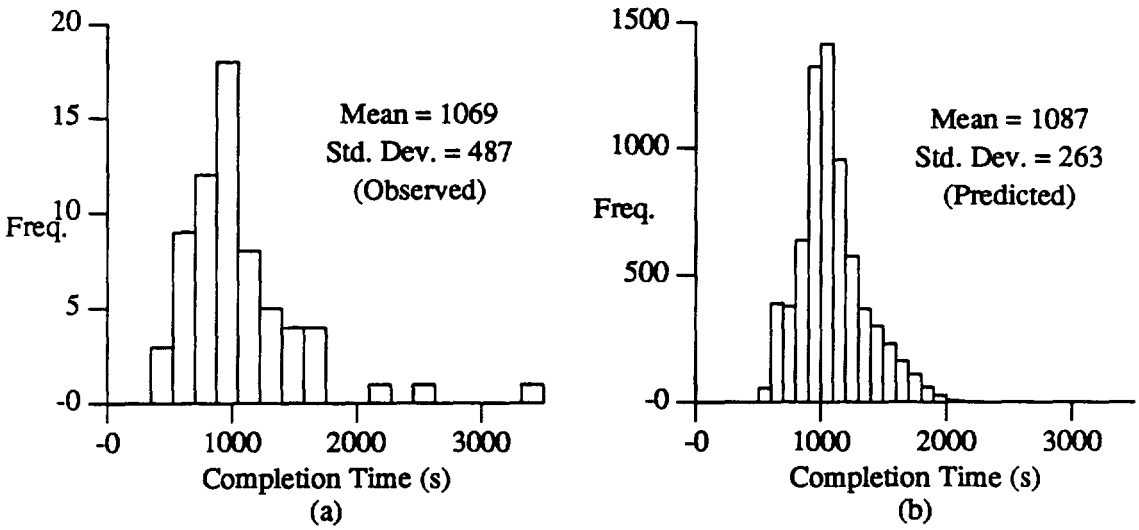



Figure 9.3 Observed and Predicted Completion Time: Scheduling Policy B

hypothesis that the two population means were equal could not be rejected. The data of this experiment validate both the model and its ability to predict the performance effects of scheduling modifications.

### 9.5. Dependence on Number of Clusters

In Section 7.1, the procedure used to choose the number of clusters in the statistical clustering step of the methodology was introduced. In the Alliant example, this procedure yielded an original clustering into 80 groups, with 72 groups left after outlier removal. However, using 80 initial states to characterize workload/system behavior seems a bit arbitrary. Why not start with 20 or 250? In this section, the dependence of the modeling methodology on the number of clusters is investigated. The results will show that the methodology is not highly sensitive to the number of clusters chosen. Similar results would be obtained if 70 or 90 clusters were used instead of 80.

The data used to construct the Alliant FX/80 model were used with varying numbers of clusters to create multiple models. All steps of the model-building methodology were performed as in Section 7.2 except that the number of original clusters used was varied. Models using from 1 to 300 initial clusters were constructed. Absorbing states with less than 10 observations were still performed to

remove outlier bias. The final number of states for each experiment can be determined from Figure 7.3.

The 300 models were used to predict the completion time of the 100 target applications (see Validation B, Section 9.2). Figure 9.4(a) shows the mean predicted completion time as a function of number of clusters used. Figure 9.4(b) shows the corresponding standard deviations.

Figure 9.4(a) shows that the predicted mean completion time changes slowly as a function of the number of clusters. In other words, a model using 40 clusters to represent the workload will make predictions similar to one using 43. This is encouraging because it shows that some randomness in construction will not have a great impact on the accuracy of the model. It is also intuitively pleasing. One would expect a model of 40 or 43 states to provide similar predictions.

Figure 9.4(a) also shows that for all constructed models, the predicted mean was fairly close to the actual mean (1303 s.). While the mean may be accurate for all models, it is unfair to assume that the predicted completion time distribution will be accurate for all of the models studied. In fact, Figures 9.5(a) - (c) show that for a small number of clusters, the prediction becomes centered, and for a large number of clusters the prediction falls victim to outlier bias. The actual distribution should

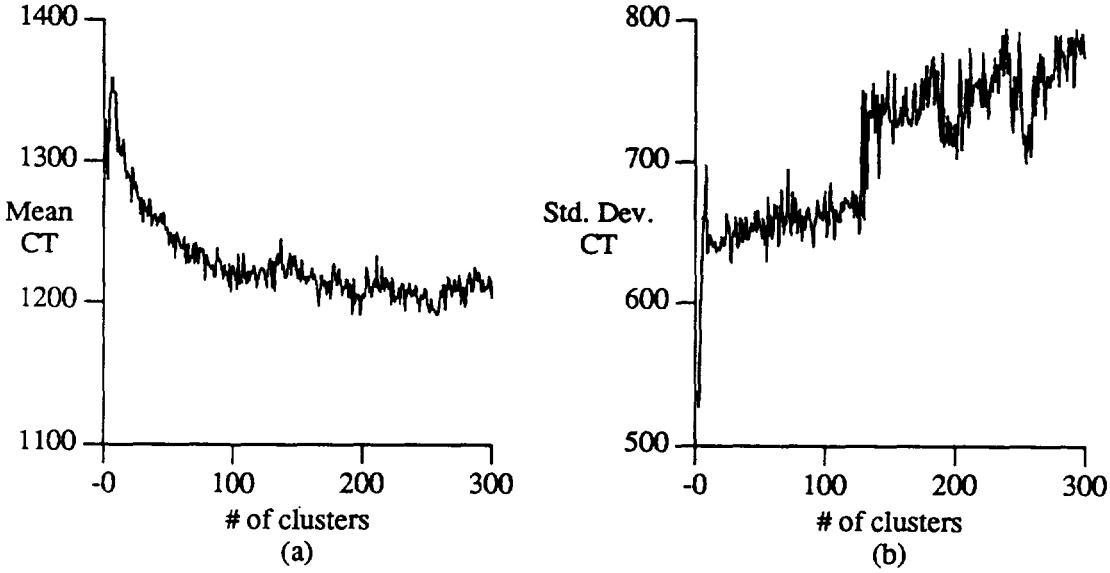
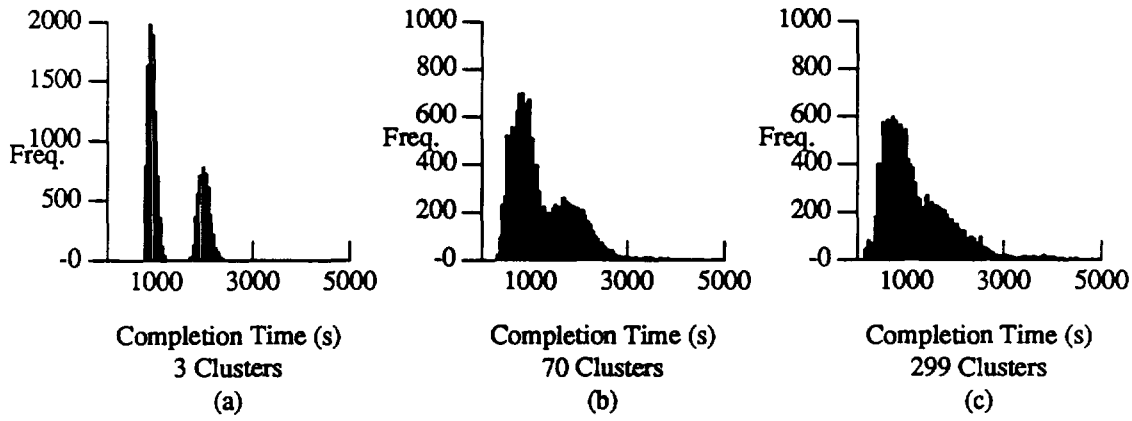



Figure 9.4 Dependence on Number of Clusters




Figures 9.5 Predicted Target Application Completion Times

resemble Figure 9.1(a). Figure 9.4(b) hints at the problems of using too many clusters. Notice that when more than 120 clusters are used, the standard deviation increases dramatically. This increased variability is due to small clusters dominating the analysis.

In summary, the methodology is not highly sensitive to the number of clusters used to represent the workloads. However, care must be taken to choose a fairly reasonable number.

# CHAPTER 10.

# CONCLUSIONS

The contributions of this thesis can be divided conceptually into three categories. First, the thesis introduced evaluation methodologies capable of answering important performance questions which have not been previously addressed. The methodologies are unique and may be used in the evaluation of other computing environments. Second, through the illustrations of these techniques, performance indices from real workloads on parallel processors were obtained. Third, the evaluations themselves identified modes of operation for improved performance in both commercial and experimental multiprocessors. The results from these specific computers may be generalized and may prove useful in future multiprocessor implementations. This section summarizes the important results and contributions of the thesis and proposes future research.

## 10.1. Summaries

### 10.1.1. Multiprogramming overhead: base component

Two techniques were introduced which quantified the base component (lower bound) on multiprogramming overhead for a real machine. The techniques were used to study the Alliant FX/8, FX/80, and Cedar supercomputers. To date it is the only study which isolates multiprogramming overhead using real applications on a real machine. The results are important to understand fully the overheads associated with multiprogramming on a multiprocessor (which are quite different from the multiprogramming overheads found on uniprocessors).

The techniques found that multiprogramming only cluster jobs on the Alliant systems consumed approximately 4% of the cluster time. When two or more detached jobs were introduced to the system, this base amount increases to 5.3%. Results also show that the number of additional cluster jobs does not affect the base amount of multiprogramming overhead on the Alliant systems.

Similarly, when there are two or more detached jobs in the system, the base multiprogramming overhead remains unchanged. Cedar studies indicate that the base component of overhead for parallel jobs is approximately 10.5%. This is over twice as much as that found on the Alliant. It is suspected that the increase is due to added synchronization overhead across the clusters. The high base component for MPO on Cedar hints at performance problems due to multiprogramming when real workloads are executed.

### 10.1.2. Multiprogramming and system overheads: Alliant

A technique capable of quantifying the MPO in real workloads (workloads found during the normal operation of the machine) was then introduced. The technique was illustrated on real workloads of the Alliant FX/8 and FX/80. System overhead, kernel spin lock overhead, and time spent handling interrupts, as well as workload characteristics such as paging, were also measured for the workloads investigated.

It was found that MPO usually consumed between 10 and 23% of the processing power available to parallel programs. Total system overhead was normally measured to be between 12 and 30% of the processing power, but was found to be as high as 82.1%. The mean MPO was determined to be 16% which is well over half of the total system overhead executed on a system (the mean system overhead value was determined to be 24% of processing power).

For all processors, the percentage of processing time spent handling interrupts was minimal and predicatably constant across workloads. Kernel lock spinning, on the other hand, was determined to be more of a degrading factor. It comprised over 1/3 of the total system overhead and was suspected of being a major component of MPO.

To understand the causes of multiprogramming and system overheads, an extensive statistical study was conducted relating the workload characteristics to the overhead values. It was found that MPO, total system overhead, and application completion were all moderately correlated, leading to

the conclusion that the overheads significantly degrade the performance of parallel applications. Relationships among the characteristics of a workload and the overhead measurements indicated that processor utilization and, to a lesser degree, paging were moderately correlated with the overhead present in the workload. It was also found that MPO was independent of the number of parallel jobs in the system, while total system overhead was not. The MPO was more dependent on the number of serial jobs in the system. It was postulated that, through increased kernel lock spinning, serial jobs executing on peripheral processors degraded the performance of parallel jobs.

#### 10.1.3. Multiprogramming and system overheads: Cedar

Workloads were constructed from real applications to investigate multiuser workloads on Cedar. The techniques presented in Chapters 4 and 5 were used to estimate MPO in these constructed workloads. It was found that the multiprogramming affected applications very differently. The performance of some applications was terribly degraded, while others were not adversely affected. For instance, MPO was determined to increase the completion time of some applications by over 100%.

Experiments were conducted to determine what application characteristics were most susceptible to performance degradation in multiapplication workloads. It was determined that synchronization of loops spread across the clusters on Cedar caused an overhead which was exacerbated by multiprogramming. It was postulated that tasks were being context-switched off while executing in critical sections, leaving other tasks spinning idly waiting for access to appropriate code or variables. One solution to this problem would be gang scheduling for the applications. It was also shown that, for the applications studied, accesses to global memory were not adversely affected by multiprogramming.

### 10.1.4. Modeling application execution

The work summarized above provided a basic, thorough understanding of overheads caused by multiple job interactions. Using this knowledge as a springboard, a methodology was developed to model the behavior of applications from a given domain executing in real workloads on a specified machine. The methodology is general and can be used to analyze a myriad of application domains and machine types. For instance, the model is not restricted to multi- or uniprocessor machines. Nor is it limited to either serial or parallel applications. In theory, the methodology can be used to model any application domain on any machine. In practice, the major limiting factor in the model-building methodology is the measurement facilities available to the user.

The constructed model is a measurement-based, Markov model with rewards and costs associated with each state. The states of the Markov model represent observed system/workload states, and the transitions among the states model observed transitions among the workload states. The rewards associated with each state quantify the amount of system resources that an application from the modeled domain would receive if submitted to the system while in that state. The cost associated with each state is the wall-clock time needed to receive the specified reward. Monte Carlo simulation is used to solve the model for a given application's completion time distribution in real workloads.

Given the base resource requirements of an application, the model can predict the probability that the application will finish by a given time X (for all X) in the real workloads. The model is useful in gauging how well an application will execute, or in predicting the performance impact of a system change. For instance, the performance effects of changing the scheduling paradigm or processor configuration can be easily evaluated. A system design change is modeled by modifying the reward function appropriately.

The methodology was illustrated by modeling the execution of computationally bound, parallel jobs on the Alliant FX/80. The amount of clustered CE time available to type A cluster jobs was used

as the reward. Over 36 hours of normal machine operation were monitored, and the resulting model used 72 distinct states to represent the workloads. Two empirical experiments demonstrated that the constructed model accurately represented the workloads monitored.

#### 10.1.5. System design modification predictions

The model constructed for the Alliant FX/80 was then used to evaluate different scheduling paradigms and processor configurations, as well as the degradation caused by MPO and the performance effects of adding extra processors to the machine. The study was conducted for two reasons: 1) to tune the Alliant to the actual workloads present and 2) to demonstrate the flexibility and power of the model-building methodology.

A number of available scheduling policies which would improve the response time of parallel jobs (e.g., the 6/2 or 5/3 static detached mode under Policy B) were identified. The analysis indicated that the proposed dynamic detached mode, if implemented, would significantly lower the response time of parallel applications. The model predicted that negligible performance improvement would be obtained by adding CEs to the system if the CEs were left in the dynamic mode. A more beneficial option would be to move to the static detached configuration and allow the added CEs to execute all serial jobs, leaving the original 8 CEs continuously clustered. The actual impact of MPO on completion time was then quantified using the model.

The evaluations demonstrated both the flexibility and the power of the modeling technique. The flexibility stems from the different methods in which system modifications can be modeled. For instance, changes can be modeled by modifying the reward function, by modifying the reward requested, or different workloads can be modeled by modifying the starting state probabilities. The model is powerful because it provides more than a single-point measurement for comparison: the entire distribution of completion times is provided. Some of the results available through the completion time distribution were illustrated. For instance, the standard deviation indicated how

predictably a job will behave under a policy or configuration. By analyzing the distribution tails, worst-case behavior can be gauged. The model can also estimate the probability that an application will finish by a given deadline. This type of information is invaluable for real-time systems.

The sensitivity of the modeling methodology to the number of clusters used in the statistical clustering step of the technique was also investigated. It was determined that the predictions made by the model changed slowly as a function of the number of clusters chosen to represent the workload/system states. This demonstrated the robustness of the method.

Finally, two empirical experiments demonstrated the accuracy of the predictions made by the model. The model was shown to successfully predict the completion time of an application executing on a system with one less processor. The model was also able to predict the completion time of a new application executing in a new workload under a new scheduling paradigm.

### 10.2. Future Work

The need for innovative performance evaluation techniques will continue to grow as systems become more complex. More specifically, methods which account for real, multiprogrammed workloads will have to be formulated and perfected. After all, these are the types of workloads which are executed by systems, thus they should be the ones used to evaluate and tune the system. Someday, the use of simple FORTRAN loops will no longer be an acceptable way to judge a system or a system design modification.

*Multiprogramming Overhead Methodology —*

The methodologies presented in this thesis are general enough to expand with changing systems. The techniques which measure the lower bound and real workload component of MPO are complete. No future work needs to be done refining or improving them. However, it would be interesting to use the techniques to measure a number of different platforms. In this way a more thorough understanding of the degradation caused by MPO would be obtained.

*Multiprogramming on Cedar —*

The problems caused by SDOALL synchronization in multiprogrammed workloads have been identified. It is now important to conduct a more detailed analysis to determine exactly which critical sections or synchronization variables are causing the problems. The use of gang scheduling could also be investigated.

*Application Modeling Methodology—*

The application execution modeling methodology (Chapter 7) could be expanded in a number of ways. As it stands, the methodology creates a discrete-time Markov model. It would be interesting to investigate the accuracy of a continuous-time Markov or semi-Markov model in representing the workload. It was found by Hsueh that semi-Markov models were superior to regular Markov models for creating measurement-based reliability/performability models [70]. Work by Devarakonda has shown that continuous-time Markov models can be used to represent process resource usage in UNIX systems [68]. Therefore, the type of Markov model employed makes a difference in the accuracy of the model for measurement-based analysis. For the methodology of this thesis, continuous-time models could easily be created by reducing the sampling period and the observation length.

The solution of the model for application completion time distribution is now estimated using Monte Carlo simulation. Trivedi has been wrestling with the difficult problem of determining the accumulated reward distribution for an irreducible continuous-time Markov chain [72]. If the model is transformed to a continuous-time Markov chain, then the double Laplace transforms proposed by Trivedi may be used to to obtain the closed-form solution for the probability of finishing an application by a given time.

In some cases a single reward function may not be sufficient to model the execution of an application. For instance, an application may depend equally on processor resources and I/O for completion. Therefore, the use of multiple reward and cost functions for each state should be investigated. For instance, for the Cedar supercomputer there may be one reward function for

processing power, and one reward function for each type of memory required by applications. The reward function for shared global memory would quantify the amount of shared global memory available to the executing application. The cost associated with that reward would be a function of the cost for paging memory in and out of the system, the memory available, and the amount of memory available. The completion time of the application would then be determined by the sum of all of the costs at the time the required reward has been collected.

Another interesting modification to the modeling technique would be to allow the type of reward required by the application to be a function of time. For instance, an application on Cedar can be described by the number of tasks or helper tasks that are running at a given time. For instance, Figure 6.2 shows TFS alternating between between using one and four tasks. If the application is running a single task (as in the first 77.5 s of MCP), then even if four processors are available on Cedar, only one can be used. To model this type of situation, the reward accumulated while stepping through the Markov model could be a function of the amount of reward the application can utilize at a given time.

For instance, Figure 10.1 illustrates the number of tasks an application uses as a function of time. The application first requires 20 s of single-cluster time; then it uses 100 s of 4-cluster time, followed by 30 s of two-cluster time. The model created would have a reward function for each situation: applications needing 1, 2, 3, or 4 clusters. Each state in the Markov chain would have four separate rewards. The reward function used at a given time would expand on the part of the application that is being simulated. Solving for the completion time of the application depicted in Figure 10.1 could be accomplished by first solving for a 20 s application using reward function 1. Then, using the state that the 20 s application completed in as the starting state, a 100 s application with reward function 4 would be simulated. This is followed by a 30 s application with reward function 3. The completion time of the application would then be the sum of the completion times of the three parts.

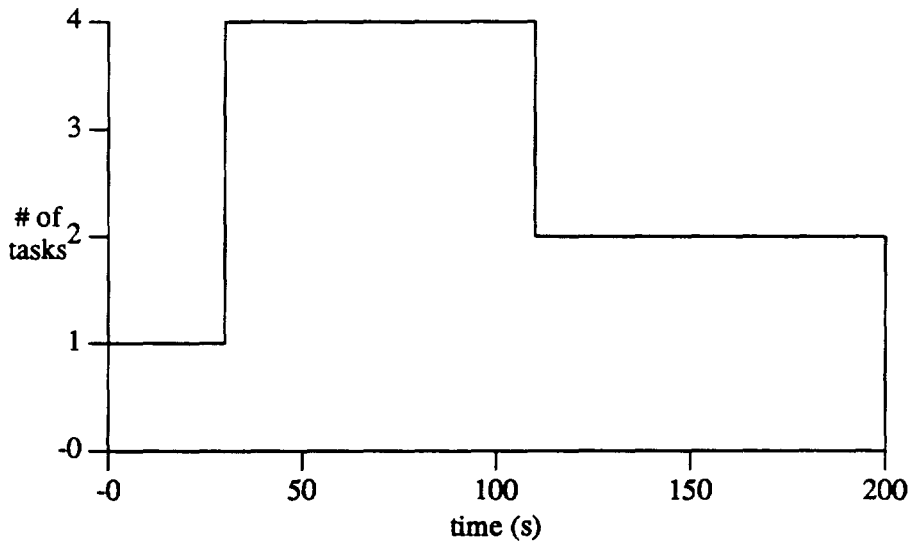



Figure 10.1 Tasks of an Application as a Function of Time

*Empirical Validation—*

Although four empirical validations have already been conducted, there is always need to do more. More specifically, the model should be used to predict the completion time of an application executing on a system without any modifications. Also, it would be beneficial to determine if the model accurately predicted the performance effects of the different system configurations.

Finally, the modeling methodology should be used to model application execution on the Cedar supercomputer. The methods described above could be employed to model the memory and task execution. The model could be used to predict the best memory configuration for the system. Of course, empirical validation should then be done to validate the results.

# REFERENCES


[1] G. Bell, "Ultracomputers a teraflop before its time," *Communications of the ACM*, vol. 35, no. 8, pp. 26 - 47, Aug. 1992.

[2] M. Chiang and G. Sohi, "Evaluating design choices for shared bus multiprocessors in a throughput-oriented environment," *IEEE Transactions on Computers*, vol. 41, no. 3, pp. 297 - 317, Mar. 1992.

[3] A. Ramani, P. Chande, and P. Sharma, "A general model for performance investigations of priority based multiprocessor system," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 747 - 754, June 1992.

[4] P. Heidelberger and K. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Transactions on Computers*, vol. 31, pp. 1099 - 1108, 1982.

[5] U. Herzog, W. Hoffman, and W. Kleinoder, "Performance evaluation modeling and evaluation for hierarchically organized multiprocessor computer systems," *Proceedings of 1979 International Conference on Parallel Processing*, pp. 103 - 114, Aug. 1979.

[6] J. Turek, J. Wolf, K. Pattipati, and P. Yu, "Scheduling parallelizable tasks: Putting it all on the shelf," *Proceedings of 1992 Conference on the Measurement and Modeling of Computer Systems*, vol. 20, no. 1, pp. 225 - 236, June 1992.

[7] R. Iyer and R. Dimpsey, "Evaluation of parallel processors," *Keynote for IEEE Region 10 International Conference on Computers, Communications and Automation*, Nov. 1992.

[8] J. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5 - 44, Mar. 1992.

[9] T. Conte and W. Hwu, "Benchmark Characterization," *IEEE Computer*, vol. 24, no. 1, pp. 48 - 56, Jan. 1991.

[10] R. P. Weicker, "An overview of common benchmarks," *IEEE Computer*, vol. 23, no. 12, pp. 65 - 75, Dec. 1990.

[11] C. Ponder, "Performance variation across benchmark suites," *Performance Evaluation Review*, vol. 18, no. 3, pp. 42 - 47, Nov. 1990.

[12] P. F. Koss, "Application performance on supercomputers," CSRD Rpt. 847, University of Illinois, Jan. 1989.

[13] J. E. Smith, "Characterizing computer performance with a single number," *Communications of the ACM*, vol. 31, no. 10, pp. 1202 - 1206, Oct. 1988.

[14] C. Hall and K. O'Brien, "Performance characteristics of architectural features of the IBM RISC System/6000" *Proceedings of Architectural Support for Programming Languages and Operating Systems*, pp. 303 - 309, Apr. 1991.

[15] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43 - 49, 1976.

[16] J. J. Dongarra, "The Linpack benchmark: An explanation," pp. 1 - 21 in *Evaluating Supercomputers*, London: Chapman and Hall, 1990.

[17] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013 - 1030, Oct. 1984.

[18] J. T. Feo, "An analysis of the computational and parallel complexity of the Livermore loops," *Journal of Parallel Computing*, vol. 7, no. 2, pp. 163 - 185, June 1988.

[19] SPEC Staff, "Benchmark results," *SPEC Newsletter*, vol. 1, no. 1, Fall 1989.

[20] T. Keller, "SPEC benchmarks and competitive results," *Performance Evaluation Review*, vol. 18, no. 3, pp. 19 - 20, Nov. 1990.

[21] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," *International Journal of Supercomputing Applications*, 1989.

[22] Lynn Pointer, Ed., "PERFECT report: 1," CSRD Rpt. 896, University of Illinois, July 1989.

[23] Center for Supercomputer Research and Development Staff, "Perfect report 2: addendum 2," CSRD Rpt. 1168, University of Illinois, Nov. 1991.

[24] A. J. van der Steen and P. P. M. de Rijk, "Guidelines for use of the EuroBen Benchmark," ACCU Tech. Rpt. TR-29, University of Utrecht, Oct. 1990.

[25] A. J. Hey and C. J. Scott, "Report of the state-of-the-art and evaluation work package," Espirit-2 project P2447, Genesis pre-study, June 1989.

[26] O. Lubeck, J. Moore, and R. Mendez, "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2," *IEEE Computer*, vol. 18, pp. 10 - 29, 1985.

[27] H. Wasserman, M. Simmons, and O. Lubeck, "The performance of minisupercomputers: Alliant FX/8, Convex C-1, and SCS-40," *Journal of Parallel Computing*, no. 8, pp. 285 - 293, 1988.

[28] R. Cmelik, S. Kong, D. Ditzel, and E. Kelly, "An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks," *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 303 - 309, Apr. 1991.

[29] M. Simmons, H. Wassermann, O. Lubeck, C. Eoyang, R. Mendez, H. Harada, and M. Ishiguro, "A performance comparison of four supercomputers," *Communications of the ACM*, vol. 35, no. 8, pp. 116 - 124, Aug. 1992.

[30] R. H. Saavedra-Barrera, "Machine characterization and benchmark performance prediction," Tech. Rpt. UCB/CSD 88/437, University of California Berkeley, June 1988.

[31] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Experimentally characterizing the behavior of multiprocessor memory systems," *Proceedings of 1989 Conference on Measurement and Modeling of Computer Systems*, vol. 18, no, 1, June 1989.

[32] R. Fatoohi, "Vector performance analysis of three supercomputers: Cray-2, Cray Y-MP, and ETA-10Q," *Proceedings of Supercomputing 89*, pp. 779 - 788, Nov. 1989.

[33] R. Fatoohi, "Vector performance analysis of the NEC SX-2," *Proceedings ACM International Conference on Supercomputing*, pp. 389 - 400, June 1990.

[34] J. L. Gustafon, G. R. Montry, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM journal of Scientific and Statistical Computing*, vol. 9, no. 4, pp. 609 - 638, July 1988.

[35] F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, and D. Gannon, "Performance evaluation and prediction for parallel algorithms on the BBN GP1000," *Proceedings of ACM International Conference on Supercomputing*, pp. 401 - 413, June 1990.

[36] D. Bradley, G. Cybenko, H. Gao, J. Larson, F. Ahmad, J. Golab, and M. Straka, "Supercomputer workload decomposition and analysis," CSRD Rpt. 1064, University of Illinois, June 1991.

[37] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Performance prediction for parallel numerical algorithms," *International Journal of High Speed Computing*, vol. 3, no. 1, pp. 31 - 62, Feb. 1991.

[38] R. Dimpsey and R. Iyer, "Modeling and measuring multiprogramming and system overheads on a multiprocessor: Case study," *Journal of Parallel and Distributed Computing*, vol. 14, no. 4, pp. 402 - 414, Aug. 1991.

[39] E. Williams, "The effects of operating systems on supercomputer performance," *in Performance Evaluation of Supercomputers*. North Holland: Elsevier Science Publishers B.V., 1988, pp. 69 - 81.

[40] A. Dinning and O. Zajicek, "Efficient mutual exclusion synchronization algorithms," Tech. Rpt. 350, Courant Institute of Mathematical Sciences, Oct. 1988.

[41] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 60 - 69, June 1990.

[42] J. Zahorjan and E. D. Lazowska, "Spinning versus blocking in parallel systems with uncertainty," Tech. Rpt. 88-03-01, University of Washington, Mar. 1988.

[43] A. Gupta, A. Tucker, and S. Urushibara, "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications," *Proceedings of Conference on the Measurement and Modeling of Computer Systems*, vol. 19, no. 1, pp. 120 - 132, May 1991.

[44] Encore Computer Corp., *Multimax Technical Summary*, Mar. 1987.

[45] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh, "Parallel supercomputing today and the Cedar approach," *Science Magazine*, vol. 231, pp. 967 - 974, Feb. 1987.

[46] P. Yew, "Architecture of the Cedar supercomputer," *Proceedings of the IBM Institute of Europe*, pp. 8 - 12, Aug. 1986.

[47] P. Emrath, D. Padua, and P. Yew, "Cedar architecture and its software," *Proceedings of 22nd Hawaii International Conference on System Sciences*, Jan. 1989.

[48] Alliant Computer System Corp., *FX/Series Product Summary*, June 1985.

[49] R. Dimpsey and R. Iyer, "Predicting the impact of scheduling modifications on system performance: Case study," *Proceedings of 25th Hawaii International Conference on System Sciences*, pp. I559 - I568, Jan. 1992.

[50] S. T. Leutenegger and M. K. Vernon, "The performance of multiprogrammed multiprocessor scheduling policies," *Proceedings of Conference on the Measurement and Modeling of Computer Systems*, vol. 18, no. 1, pp. 226 - 236, May 1990.

[51] J. Zahorjan and C. NcCann, "Processor scheduling in shared memory multiprocessors," *Proceedings of Conference on the Measurement and Modeling of Computer Systems*, vol. 18, no. 1, pp. 214 - 225, May 1990.

[52] C. D. Polychronopoulos, "Multiprocessing versus multiprogramming," *Proceedings 1989 International Conference on Parallel Processing*, pp. 223 - 230, Aug. 1989.

[53] J. K. Ousterhoust, "Scheduling techniques for concurrent systems," *Proceedings of Distributed Computing Systems Conference*, pp. 22 - 30, 1982.

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

[55] M. Berry, H. Chen, E. Gallopoulos, U. Meier, A. Tuchman, H. Wijshoff, and G. Yang, "Algorithmic design on the Cedar multiprocessor," CSRD Rpt. 851, University of Illinois, Feb. 1989.

[56] J. Hoeflinger, "Cedar FORTRAN programmer's handbook," CSRD Rpt. 1157, University of Illinois, Oct. 1991.

[57] M. Guzzi, "Cedar FORTRAN programmer's handbook," CSRD Rpt. 601, University of Illinois, June 1987.

[58] R. McGrath and P. Emrath, "Using memory in the cedar system," CSRD Rpt. 655, University of Illinois, June 1987.

[59] R. McGrath, "Memory overheads for xylem programs," CSRD Rpt. 858, University of Illinois, Feb. 1988.

[60] R. McGrath, "Run-time virtual memory statistics for a cedar program," CSRD Rpt. 857, University of Illinois, May 1989.

[61] K. Gallivan, W. Jalby, A. Malony, and P. Yew, "Performance analysis of the Cedar system," *in Performance Evaluation of Supercomputers*. North Holland: Elsevier Science Publishers B.V., 1988.

[62] A. Malony and J. Pickert, "An environment architecture and its use in performance data analysis," CSRD Rpt. 829, University of Illinois, Oct. 1988.

[63] J. Andrews, "A hardware tracing facility for a multiprocessing supercomputer," CSRD Rpt. 1009, University of Illinois, May 1990.

[64] A. Malony, "Virtual high resolution process timing," CSRD Rpt. 616, University of Illinois, Oct. 1986.

[65] A. Malony, "Cedar performance evaluation tools: a status report," CSRD Rpt. 582, University of Illinois, July 1986.

[66] A. Mink, G. Nacht, and J. Roberts, "Multiprocessor performance-measurement instrumentation," *Computer*, vol. 23, no. 9, pp. 63 - 75, Sept. 1990.

[67] R. Dimpsey and R. Iyer, "Performance analysis of a shared memory multiprocessor: Case study," *Proceedings of 1988 International Conference on Parallel Processing*, pp. 174 - 181, Aug. 1988.

[68] M. Devarakonda and R. Iyer, "Predictability of process resource usage: A measurement-based study of UNIX," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Oct. 1987.

[69] H. Artis, "Workload characterization using SAS PROC FASTCLUS," *International Workshop on Workload Characterization of Computer Systems and Computer Networks*, Oct. 1985.

[70] M. Hsueh, R. Iyer, and K. Trivedi, "A measurement-based performability model for a multiprocessor system," *Proceedings of 2nd International Workshop Applied Mathematics and* Performance Reliability Models Computer/Communication Systems," May 1987.

[71] SAS Institute, *SAS User's Guide: Statistics, V5*, SAS Institute, Cary, NC, 1985.

[72] V. G. Kulkarni, V. F. Nicola, R. M. Smith, and K. S. Trivedi, "Numerical evaluation of performability and job completion time in repairable fault-tolerant systems," *Proceedings of International Symposium of Fault-Tolerant Computing Sciences*, pp. 252 - 257, 1986.

[73] R. Dimpsey and R. Iyer, "Performance degradation due to multiprogramming and system overheads in real workloads: Case study on a shared memory multiprocessor," *Proceedings International Conference on Supercomputers*, pp. 227 - 238, June 1990.

[74] K. Trivedi, *Probability & Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

# VITA

Robert Dimpsey was born in [redacted] of the year [redacted] in [redacted] He received the B.S. degree in Computer Engineering in 1986 from the University of Illinois and the M.S. degree in Electrical Engineering in 1988 from the University of Illinois. He received a fellowship from the University of Illinois in 1986. He has held summer positions for the Nuclear Data Corp. and the IBM T.J. Watson Research Center, and has been a teaching assistant and seminar instructor at the University of Illinois. Currently he has published 8 papers in peer reviewed conferences and journals. He is a member of the ACM and IEEE. Upon completion of his Ph.D. he will join IBM in Austin, Texas.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS<br>None |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>UILU-ENG-92-2234 (CHRC-92-18) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Coordinated Science Lab<br>University of Illinois<br>6b. OFFICE SYMBOL (If applicable)<br>N/A | 7a. NAME OF MONITORING ORGANIZATION<br>NASA, ONR |
| 6c. ADDRESS (City, State, and ZIP Code)<br>1101 W. Springfield Avenue<br>Urbana, IL 61801 | 7b. ADDRESS (City, State, and ZIP Code)<br>NASA Langley Res. Ctr. Hampton, VA 23665<br>Office of Naval Researc 800 N. Quincy, Arlingto VA 22217 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>NASA<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>NASA NAG-1-613, N00014-91-J-1116 |
| 8c. ADDRESS (City, State, and ZIP Code)<br>NASA Langley Research Center<br>Hampton, VA 23665 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)

PERFORMANCE EVALUATION AND MODELING TECHNIQUES FOR PARALLEL PROCESSORS

12. PERSONAL AUTHOR(S)

ROBERT TOD DIMPSEY

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Technical | FROM _____ TO _____ | AUGUST 1992 | 149 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Parallel Processing, performance evaluation, measurement-based analysis, multiprogramming, overheads, modeling, markov reward models |
| | | | |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In practice, the performance evaluation of supercomputers is still substantially driven by singlepoint estimates of metrics(eg., MFLOPS) obtained by running characteristic benchmarks or workloads. With the rapid increase in the use of time-shared multiprogramming in these systems such measurements are clearly inadequate. This is because multiprogramming and system overhead, as well as other degradations in performance due to time varying characteristics of workloads, are not taken into account. In multiprogrammed environments, multiple jobs and users can dramatically increase theamount of system overhead and degrade the performance of the machine. Performance techniques, such as benchmarking, which characterize performance on a dedicated machine ignore this major component of true computer performance.

Due to the compexity of analysis ther has been little work done in analyzing, modeling and predicting the performance of applications in multiprogrammed environments. This is especially

| | |
|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ OTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL |

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

true for parallel processors, where the costs and benefits of multi-user workloads are exacerbated. While some may claim that the issue of multiprogramming is not a viable one in the supercomputer market, experience shows otherwise. Even in recent massively parallel machines, multiprogramming is a key component. It has even been claimed that a partial cause of the demise of the CM2 was the fact that it did not efficiently support time-sharing[1]. In the same paper, Gordon Bell postulates that, "Multicomputers will evolve to multiprocessors" in order "to support efficient multiprogramming". Therefore, it is clear that parallel processors of the future will be required to offer the user a time-shared environment with resonable response times for the applications. In this type of environment the most important performance metric is the completion or response time of a given application. However, there are few evaluation efforts addressing this issue.