2003

Distributed dispatchers for partially clairvoyant schedulers

Kiran S. Yellajyosula
West Virginia University

Distributed Dispatchers for Partially Clairvoyant Schedulers

by

Kiran S Yellajyosula

Thesis submitted to the College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Electrical Engineering

Approved by

Dr. K. Subramani, Committee Chairperson
Dr. Supratik Mukhophadyay
Dr. Bojan Cukic
Dr. Hany H. Ammar

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia.
2003

Keywords: Partially Clairvoyant schedules, online dispatchers, loss of dispatchability, distributed memory, PRAM models.

Abstract

Distributed Dispatchers for Partially Clairvoyant Schedulers.

Kiran S Yellajyosula.

This work focuses on the empirical evaluation of distributed dispatching strategies on shared and distributed memory architectures for hard real-time systems. The dispatching model accommodates process parameter variability and analyzes the effect of variable execution times.

Hard real-time systems are modeled in the E-T-C scheduling framework and dispatched if a valid schedule exists. We examine the dispatchability of Partially Clairvoyant schedules of different sizes and varying deadlines under reasonable assumptions. The effect of scaling up the number of processors used by the dispatcher is also studied. The results validate the superiority of the distributed strategies over sequential dispatching and scalability of the distributed strategies. Certain system limitations which lead to Loss of Dispatchability in the experiments were pointed out.

The model finds applications in diverse areas like safety critical systems, robotics and machine control, real-time data management, and this approach is targeted at powering up the controllers.

Acknowledgments

I thank my advisor, Dr. K. Subramani for his patience, guidance and encouragement, without which I would not have reached my goals.

I would like to thank Dr. Supratik Mukhopadhyay for being in my committee and providing me with his valuable suggestions. I would also like to thank Dr. Bojan Cukic and Dr. Hany Ammar, for their support and serving on my committee.

This work was partially supported by National Computational Science Alliance under [ASC30006N] and utilized the account [kirany] and also by Pittsburgh Supercomputing Center under [ASC020016P] and utilized the account [yellajyo].

I was helped in numerous occasions by the support staff of the Pittsburgh Super Computing center and National Computational Science Alliance. I am indebted for their time and valuable suggestions.

I would like to thank my friends Ashraf Osman, Rabita Sarker and Kalyan Reddy Kasarla for their valuable suggestions and discussions.

I thank Don McLaughlin, Dr. Vanscoy and her staff for giving me a login on 'Energy' and helping me take my first step in parallel programming.

Contents

List of Figures

List of Tables

Chapter 1

Real-time Systems

1.1 Introduction

Real-time systems are mostly embedded in dynamic environments and need to react to external stimuli within their deadlines. The system samples its inputs from the environment, computes the response and responds to the changes recorded. A response can be the execution of a job or computing of a number or sending a message. Real-time systems are gaining importance with the development of software systems to control and monitor applications like nuclear reactors, robots, satellites, $MP3$ players, data sensing and transmission, automated factory pipelines and other systems.

Hard real-time systems are that subset of real-time systems where failure to compute a result within a deadline could result in the failure of the system. Air craft controllers, life-support systems, nuclear reactors are typical examples of this subset. The failure of such systems have catastrophic results such as destruction or damage to the system or life loss. Both the software and the hardware of the system must function with high reliability and the deadlines in these systems are to be met at any cost.

The functionality of a system is described as a set of jobs with constraints between them. The execution times of jobs was assumed to be constant (worst case execution time) and different scheduling strategies were proposed depending on heuristics, such as earliest deadlines, job execution time, and job frequency [LW73, SRL90]. With the development and availability of preemptive scheduling systems, job scheduling was targeted to increase the throughput and utilization of the processor based on the priorities of the jobs. Jobs of lower priority were suspended till all jobs of higher priority were completed. Several variations were tested to prevent starvation of low priority jobs, such as priority inheritance and aging techniques.

Many embedded real-time operating systems like VRTX, PSoS and Vxworks were developed. These operating systems had preemptive kernels with a small context switch time, multiple watchdog timers of high precision and very small clock cycle. The operating systems used online strategies or look-up tables or job priorities to determine the next action to be taken. There were in-built timers to monitor different actions and interrupt the operating system when an action needs to be taken.

With the availability of multiple processors for executing jobs, multiprocessor real-time operating systems evolved. The jobs were assigned to and scheduled on processors to balance the load while meeting the deadlines. The distribution of data with jobs created data dependencies across processors. The necessity to share data within a small duration has lead to requirement of fast, reliable communication between processors. A few examples of real-time systems in dynamic environments that need to react within small durations are listed as follows:

- Robots have a mission and are equipped with multiple sensors and actuators to complete their mission. Robots are being developed to achieve missions in hostile environments like surveying landscape and searching for survivors [RGH$^+$02]. Clusters of independent robots are being developed to achieve missions in hostile environments like surveying landscape and searching for survivors. The robots control their own motion, communicate with each other and complete jobs distributed among them to complete the mission.

  The Mars Pathfinder had to explore the surface of Mars and transmit information about the surface and topology on the planet. The motion of a robot requires complex modeling and has to consider various kinematic equations which require different computing times [YYM01, HCF03]. It also had to monitor the environment and control its motion. Such robots need to perform multiple tasks concurrently and monitor their components [CJD91]. This justifies the requirement of online controllers; that would control the actions of the robot and maintain the deadlines across them. Since the environment is dynamic, we cannot use any offline strategy for controlling the system. The hardware and software components of a robotic team that surveys a landscape communicating with the central robot is presented in [RSE$^+$00].

- An automobile cruise control maintains the speed of the car by coordinating and monitoring the actions of different components of the engine such as fuel injection, braking and transmission. New cars have adaptive shifting algorithms, modifying shift points based on road conditions, weather, and the driver's individual habits. The cruise control system can vary the car acceleration according to the exact speed of the car provided by the Anti-lock Braking System. These systems require variable times to compute the required torque to drive the car at a safe speed. A 7−series BMW has 63 microprocessors while

a Mercedes S-class has 65 microprocessors. Jaguars and Volvos, use the PowerPC 505 to control the engine and calculate time-angle ratios, which is vital for valve and ignition timing.

- Sound and video synchronization are essential for a person playing video games. The current video games like PlayStation, Dreamcast, GameCube and N64 have multiple embedded processors to record, display and react to a change made by a player within a very small duration. Game controllers have multiple tasks to perform like controlling hardware and computing the response to the player. Computing a response to a move might require considering complex situations and scenarios which the game is animating.

  Massive Multiplayer Online Role-Playing Games (MMORPGs) such as *Star Wars, City of Heroes, World of Warcraft* are growing in the gaming communities. MMORPGs are derived from MUDS and computer games. The gamers log into the host server and create characters or avatars along with the virtual characters in the game. The server coordinates the actions of thousands of players over the internet and stores the changes made to the environments and characters of each person. An interaction or a fight between two people is real-time where the server needs to respond in fractions of a second. Each server has a maximum capacity of the number of people that it can host. Each person has a mission, while there are options allowing people to group and take up collective missions. This adds the complexity of network delays to the problem of scheduling jobs. In a battlefield, the host server is required to keep track of the environment and also respond to the gamers.

With the increasingly dynamic nature of the real-time systems, the execution of a job takes different times in different scenarios. The traditional models use the worst case execution time to model the real-time systems and would declare complex systems to be infeasible. This promotes the requirement of a more flexible modeling technique for real-time systems.

1.2 E-T-C Scheduling Model

Real-time scheduling models have complex relations between the jobs which are to be satisfied at all times. Some dynamic real-time environments, where the execution time changes according to the circumstance, are listed in the previous section. The execution time of a job can vary due to different factors such as:

1. Input dependent loops - The time to run a job containing loops depending on input parameters changes with the parameters.

2. Caching - Modern computers have multi-tier memories for improving the memory latency and page

swapping operating systems to provide larger memory space. The time taken to access a data depends on where the data is located.

3. Compiler-architecture mapping of the machine - Different compilers go to different levels of optimization and produce code with different strategies. The length of code differs and could cause the execution time to change.

4. Processor speed - The execution time of jobs also varies when the processor executes the jobs with different clock speeds. Transmeta's LongRun, AMD's PowerNow, or Intel's SpeedStep technologies vary the processor voltage or clock frequency to decrease the power consumed by the processors. Building or interfacing real-time systems with such processors further complicates the situation. A scheme to decrease energy consumption for real-time systems by readjusting processor speed and reusing the unused processor cycles mustered, when a job finishes before the worst case execution time, is proposed in [AMMMA01]

The `E-T-C` framework is proposed in [Sub02] to formalize problems in real-time systems which takes into account the variability of execution time, complex relationships between jobs and clairvoyance of the system. The scheduling model consists of three sub-models, namely, the Job model, the Constraint model and the Query model. The Job model describes the type and nature of jobs to be scheduled. The Constraint model describes the relationships existing between the start or finish times of the jobs. The Query model specifies what it means for a job set to be schedulable under the imposed constraints. A scheduling model in the E-T-C framework is constructed by specifying the three sub-models.

The jobs are ordered and non-preemptable and the constraints imposed on the jobs are strict difference constraints between start and finish times of jobs. We use the algorithm proposed in [Sub03] to decide the schedulability of a Partially Clairvoyant system and a set of dispatch functions are generated when a schedule exists.

1.3 Strategy

In this thesis, we focus on "the dispatching analysis and implementation of distributed dispatchers" for Partially Clairvoyant systems. We provide additional processors to relax constraints and obtain the dispatch interval of jobs while one processor executes a job. The data required for relaxing constraints is either transmitted as messages between the processors or stored at a shared location. The temporal deadlines imposed on the system are met by increasing the processing capacity of the controller.

We implement and test the sequential and parallel versions of the online dispatcher with schedules of different sizes and constraints. We study the effect of increasing the number of processors on the update time for schedules of different sizes. We also study the dispatchability of schedules of different execution time intervals and spacing time intervals and suggest the factors that can improve dispatchability of schedules.

1.4 Summary of Contributions

The main contributions in the thesis are as follows:

- Algorithms : This work extends the parallel online dispatcher for Partially Clairvoyant systems proposed in [Sub00] to a fixed number of processors; much less than the number of jobs. The original algorithm requires n processors and assumes that the cost of transmitting data to all the processors is constant, while the extended algorithm allows the number of processors to be variable and has constant transmission time. The job set was divided into mutually exclusive and exhaustive sets and assigned to processors. The complexity of the online dispatching algorithm is analyzed assuming that the execution time of a job is greater than the update time. The algorithms were modified to suite two different memory architectures, namely:

  1. Shared Memory: Two flags synchronize the execution of the jobs and the updating of constraints. Memory is flushed to read or write the shared data from the central memory based on the values of these flags. The algorithm uses one dedicated processor for executing jobs while the other processors compute the time interval within which the next job can be started without violating the constraints.
  2. Distributed Memory: Processors share data by passing messages to the others. The receiving of a message is blocking and synchronizes the execution of jobs and updating of constraints. The dispatching algorithm in [Sub00] was extended to propose and analyze two algorithms using fixed number of processors. The first algorithm uses a single processor to execute the jobs and the rest of the processors are used to update constraints. The effect of using a communicating processor to transmit data is analyzed through an analytic model. The second algorithm distributes the execution of jobs on multiple processors and analyzes the complexity of dispatching jobs.

- Empirical Analysis - We identify the parameters that effect dispatching and show their effects through experiments. We show the superiority and scalability of the distributed dispatching strategies through experimentation. In our experiments, the distributed dispatchers dispatched schedules of different sizes where the sequential dispatcher failed, and increasing the number of processors helped in dispatching

larger schedules. We show the effect of the execution time and spacing time on the dispatchability of schedules and identify and analyze various parameters leading to failure while dispatching.

1.5 Organization

The rest of the thesis is organized as follows: Chapter 2 provides a discussion of the Partially Clairvoyant scheduling model and describes how to decide if a Partially Clairvoyant schedule exists. Chapter 3 describes dispatching Partially Clairvoyant schedules on a single and multiple processors. It also explains the experimental setup, simulation of the job execution and the test suites created. Chapter 4 describes the algorithms proposed for distributed memory machines and analyzes their complexity and this chapter also analyzes the effect of a communicating processor in the single controller model and lists the results obtained on experimentation. Chapter 5 proposes the dispatching algorithm for shared memory machines and lists the results obtained on experimentation. Chapter 6 concludes by summarizing the results obtained and suggesting pointers for future research.

Chapter 2

Partially Clairvoyant Scheduling

2.1 Introduction

This chapter focuses on determining the existence of a schedule for Partially Clairvoyant real-time systems, wherein the dispatch time of the current job may depend upon the start and execution times of the jobs sequenced before it. Partially Clairvoyant scheduling was introduced in [Sak94] to reduce the inflexibility of static scheduling in hard real-time systems. The scheduling problem for Partially Clairvoyant systems has two stages:

1. Schedulability - Given an instance of the problem in the E-T-C model, to determine if the system is schedulable under the given constraints (Section §2.2).
2. Dispatchability - Given a schedule for a Partially Clairvoyant system, to determine the start time interval of jobs and dispatch all the jobs such that none of the constraints are violated (Chapter 3).

The problem of dispatching a Partially Clairvoyant system exists only if the instance of the problem is decided to be schedulable. A constraint graph is built from the difference constraints imposed on the system and the algorithm checks for the existence of a negative cycle in the graph by relaxing and removing redundant edges and contracting vertices. When a negative cycle is found in the graph, the system is declared to be infeasible. Schedulability of a Partially Clairvoyant system is determined offline while deciding the dispatchability of a schedule is done at run-time with the job execution. Online dispatching techniques for Partially Clairvoyant systems are introduced in Chapter 3 and discussed in detail in Chapters 4 and 5.

This chapter reviews and describes contributions towards Partially Clairvoyant scheduling made in [Sub00, Sub03].

The rest of this chapter is organized as follows: Section §2.2 formally introduces the problem of Partially Clairvoyant scheduling in the E-T-C scheduling framework. Section §2.3 describes the procedure for constructing a constraint graph and states the approach and complexity of the algorithm used to decide the Partially Clairvoyant schedulability of a problem for the special case in which all constraints are strictly relative. Section §2.4 provides an example of a real-time system modeled in the E-T-C framework and determines the schedulability of the system. Section §3.5 describes related work done in Partially Clairvoyant scheduling.

2.2 The Scheduling Problem

2.2.1 Job Model

Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of non-preemptive, ordered hard real-time jobs to be scheduled in time windows of length L. At the start of each scheduling window, the time is set to zero.

2.2.2 Constraint Model

The constraints on the jobs are described by System (2.1):

$$\mathbf{A}.[\vec{\mathbf{s}}\ \ \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}}, \quad \vec{\mathbf{e}} \in \mathbf{E}, \tag{2.1}$$

where,

- $\mathbf{A}$ is an $m \times 2.n$ rational matrix; the constraint set comprises only of standard constraints between two jobs. Standard constraints express the difference relationships between the start times or finish times of two jobs and are explained in Section §2.3.1.
- $\mathbf{E}$ is an axis-parallel rectangle `aph` represented by:

$$\mathbf{E} = [l_1,\, u_1] \times [l_2,\, u_2] \times \ldots [l_n,\, u_n] \tag{2.2}$$

  The `aph` $\mathbf{E}$ models the fact that the execution time of job J_i can assume any value in the range $[l_i,\, u_i]$ i.e., it is not constant.

- $\vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n]$ is the start time vector of the jobs, and
- $\vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n] \in \mathbf{E}$ is the execution time vector of the jobs.

The jobs are ordered, i.e., $s_i + e_i \leq s_{i+1}, i = 1, 2, \ldots, n-1$; the ordering constraints are part of the constraint matrix $\mathbf{A}$.

2.2.3 Query Model

Suppose that job $J_a, 1 \leq a \leq n$, has to be dispatched. The dispatcher has access to the start times $\{s_1, s_2, \ldots, s_{a-1}\}$ and execution times $\{e_1, e_2, \ldots, e_{a-1}\}$ of the jobs $\{J_1, J_2, \ldots, J_{a-1}\}$.

Definition: 2.2.1 *A Partially Clairvoyant Schedule of an ordered set of jobs, in a scheduling window, is a vector $\vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n]$, where each s_i, $1 \leq i \leq n$, is a function of the start time and execution time variables of jobs sequenced prior to job J_i, i.e., $\{s_1, e_1, s_2, e_2, \ldots, s_{i-1}, e_{i-1}\}$.*

Note that s_1 is numeric, since J_1 is the first job in the sequence.

Definition: 2.2.2 *A Partially Clairvoyant Schedule $\vec{\mathbf{s}}$ for the constraint system (2.1) is said to be feasible, if for all sequences $b_{seq} = < s'_1, e'_1, s'_2, e'_2, \ldots, s'_n, e'_n >$, where s'_i is chosen as per $\vec{\mathbf{s}}$ and $e'_i \in [l_i, u_i]$, we have $\mathbf{A}.[\vec{\mathbf{s}'} \ \vec{\mathbf{e}'}]^{\mathbf{T}} \leq \vec{\mathbf{b}}$, where s'_i and e'_i are numeric vectors, corresponding to the sequence b_{seq}.*

We look for the existence of a start time vector,*where the start time of a job J_i depends on the start and execution times of the jobs scheduled before it*, such that for any duration of job execution within $[l_i, u_i]$, the start and execution time vectors do not violate the constraints imposed on the system. The discussion above directs us to the following formulation of the schedulability query:

$$\exists s_1 \, \forall e_1 \in [l_1, u_1] \, \exists s_2 \, \forall e_2 \in [l_2, u_2] \ldots \exists s_n \, \forall e_n \in [l_n, u_n] \quad \mathbf{A}.[\vec{\mathbf{s}} \ \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}}? \tag{2.3}$$

The combination of the Job model, Constraint model and the Query model constitutes a scheduling problem specification within the `E-T-C` scheduling framework [Sub02].

2.3 Algorithm

2.3.1 Standard Constraints

The class of standard constraints was introduced in [GPS95] to describe strict difference constraints between jobs .

Definition: 2.3.1 *A constraint is said to be standard, if it represents a strict difference constraint between exactly* 2 *jobs.*

As per Definition (2.3.1), the relationships between job J_i and job J_j are standard, if they fall into one of the following categories:

1. A difference constraint between the start time of J_i and the start time of J_j, e.g. $s_i \leq s_j + c$
2. A difference constraint between the start time of J_i and the finish time of J_j, e.g. $s_i \leq s_j + e_j + c$
3. A difference constraint between the finish time of J_i and the start time of J_j, e.g. $s_i + e_i \leq s_j + c$
4. A difference constraint between the finish time of J_i and the finish time of J_j, e.g. $s_i + e_i \leq s_j + e_j + c$

Absolute constraints ($s_i \geq a$ or $s_i \leq b$) are treated as relative constraints through the addition of a dummy job J_0 with start time s_0 and execution time $e_0 \in [0, 0]$.

Observe that standard constraints are in fact difference constraints between jobs; consequently, they do have a constraint graph structure [CLR92]. In Section §2.3.2, the construction of the constraint graph corresponding to a set of standard constraints is shown.

2.3.2 Construction of the Constraint Graph for Standard Constraints

Given a set of n jobs, with standard constraints imposed on their execution, a graph $\mathbf{G} =< \mathbf{V}, \mathbf{E} >$ is constructed, where $\mathbf{V}$ is the set of vertices and $\mathbf{E}$ is the set of edges.

1. $\mathbf{V} = \langle s_0, s_1, s_2, \ldots, s_n \rangle$, i.e., one node for the start times of each job, and node s_0 which is used for handling absolute constraints;
2. For every constraint of the form: $s_i + k \leq s_j$, construct an arc $s_i \rightsquigarrow s_j$, with weight $-k$;
3. For every constraint of the form: $s_i + e_i \leq s_j + k$, construct an arc $s_i \rightsquigarrow s_j$, with weight $k - e_i$;

4. For every constraint of the form: $s_i \leq s_j + e_j + k$, construct an arc $s_i \rightsquigarrow s_j$, with weight $e_j + k$;
5. For every constraint of the form: $s_i + e_i \leq s_j + e_j + k$, construct an arc $s_i \rightsquigarrow s_j$, with weight $e_j - e_i + k$;
6. Finally construct arc $s_o \rightsquigarrow s_1$ with weight 0, since $s_1 \geq 0$ and arc $s_n \rightsquigarrow s_0$ with weight $L - e_n$, since all jobs have to be completed by the end of the current scheduling window.

Given an instance of a scheduling problem with standard constraints and execution time belonging to closed intervals, algorithm proposed in [Sub03] is used to decide the schedulability of the constraint graph generated using the procedure in Section §2.3.2. The algorithm proceeds by eliminating e_i followed by s_i in succession, starting with the last job. The execution time e_i, is eliminated by substituting $[l_i, u_i]$ on the edges depending on e_i and removing redundant edges. l_i is substituted on those edges where e_i has positive sign and u_i where e_i has a negative sign. After substituting, the edges have rational weights and that edge with the least weight is retained. The start time s_i is eliminated by contracting the vertex corresponding to s_i.

2.3.3 Complexity

The complexity of the algorithm used is $O(n^3)$ as discussed in [Sub03]. The time taken to eliminate the execution time variable e_i depends on the degree of the vertex s_i, since e_i is present only in constraints involving s_i. Since there are $n+1$ vertices, the number of edges involving e_i can be at most $4 \cdot (n+1)$. Hence the elimination of the execution time variable takes $O(n)$ time in the worst case. The time to eliminate the variable s_i is the time taken to contract one vertex in the constraint graph. The contraction of a vertex takes time proportional to the product of the in-degree and the out-degree of the vertex, since the relaxing of edges can be done in constant time. In the worst case, there are $O(n)$ edges coming into the vertex and $O(n)$ edges going out. Hence the time taken to contract a vertex is $O(n^2)$.

The time spent in contracting the n vertices is $O(n^3)$. Hence the complexity of algorithm is $O(n^3)$.

2.4 Example

Consider a simple robot trying to move an object from one place to another. The speed of the robot depends on the mass of the object and the surface on which the robot is moving; and the time the robot takes to change direction depends on the angle it has to turn. Consider that the movement is controlled by the following algorithm: The robot checks its speed by sensing the environment and varies its speed according to the requirement. It takes two units to compute the required speed and then adjusts its speed. After

adjusting the speed it checks if it is moving in the right direction towards the destination. If the robot is not moving in the right direction, it adjusts the direction. Suppose this happens once in every forty units of time with the additional constraints that the robot should start finding the direction of motion between five to ten units of finding its speed. Assume that the robot takes around three to seven units of time to find the speed, five to six units of time to adjust its speed, two to seven units of time to find the direction in which it is moving and eight to twelve units of time to adjust the direction.

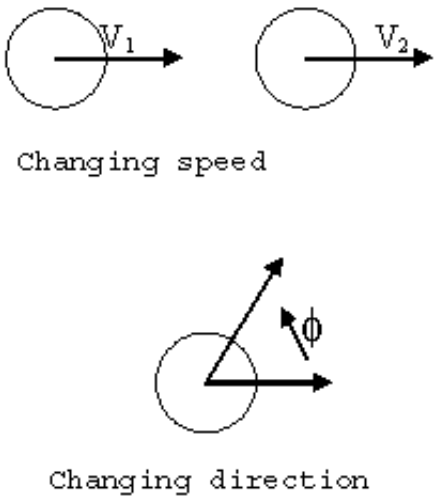

Figure 2.1: A simple robot

The above system can be modeled as:

- $\mathcal{J}$=$\{J_1, J_2, J_3, J_4\}$
- $e_1 \in [3, 7]$
  $e_2 \in [5, 6]$
  $e_3 \in [2, 7]$
  $e_4 \in [8, 12]$
- $s_1 + e_1 + 2 \leq s_2$
  $s_2 + e_2 \leq s_3$
  $s_3 \leq s_1 + e_1 + 10$
  $s_1 + e_1 + 5 \leq s_3$

- $s_3 + e_3 + 5 \leq s_4$
- $s_4 + e_4 \leq 40$

If the length L of each scheduling window is less than 39, i.e., $(s_4 + e_4 < 39)$, then the above constraint system would be infeasible.

2.5 Related Work

The E-T-C scheduling model was introduced and formalied in [Sub02]. The term "Partially clairvoyant scheduling" was first used in [Sub02] while the scheduling was introduced in [Sak94]. A polynomial time algorithm to decide schedulability was proposed in [GPS95], when the constraints imposed on the jobs are standard constraints. The principal technique used in their algorithm was the Fourier- Motzkin elimination procedure to eliminate existentially quantified variables [DE73]. They showed that when the constraints are standard, the elimination procedure does not lead to an exponential increase in the set of resolvent constraints, a phenomenon observed when the constraints are arbitrary [HJLL90].

Chapter 3

Partially Clairvoyant Dispatching

3.1 Introduction

The algorithm in [Sub03] decides the schedulability of a Partially Clairvoyant system and produces a set of dispatch functions when the query (2.3) is satisfied. In general, the dispatch functions produced are as follows:

$$\max(g_0, g_1, \ldots, g_{i-1}) \leq s_i \leq \min(g'_0, g'_1, \ldots, g'_{i-1}).$$

where g_j and g'_j are functions depending on the start and execution times of job J_j ($j < i$). The dispatching algorithm has to compute the time interval during which a job can start and dispatch the job in the computed interval so that none of the constraints imposed on the job are violated.

Definition: 3.1.1 *A safety interval($[l_b, r_b]$) for a job is the time interval during which the job can be started such that none of the constraints imposed by the constraint system (2.1) are violated.*

The dispatch algorithm fails to dispatch a job in the computed safety interval due to the delay in starting the job, i.e., the time after computing the safety interval exceeds r_b. Hence the job set cannot be dispatched and the system looses dispatchability. This phenomenon is called *Loss of Dispatchability.*

The dispatch functions generated by the dual algorithm for the example problem in Section §2.4 in Chapter 2 are as follows:

1. $0 \leq s_1 \leq 1$

2. $s_1 + e_1 + 2 \leq s_2 \leq \min(s_1 + e_1 + 4,\ 10)$

3. $\max(s_1 + e_1 + 5, s_2 + e_2) \leq s_3 \leq \min(s_1 + e_1 + 10,\ 16)$

4. $s_3 + e_3 + 5 \leq s_4 \leq 28$

Definition: 3.1.2 *A feasible Partially Clairvoyant schedule is said to be dispatchable on a machine* M*, if for every job* J_i*,* M *can start executing* J_i *within it's safety interval.*

A sequential online dispatcher executes the schedule by executing a job J_i and updating the safety intervals of the jobs depending on J_i. A multiprocessor dispatcher has one processor executing a job while the remaining processors update the safety intervals. The remaining processors update and report the safety interval for the next job, there by consuming less time than the sequential dispatcher.

In this chapter, the requirement of distributed dispatching is motivated by citing examples where sequential dispatching fails. We point out the parameters involved in creating Partially Clairvoyant schedules and identify those which make dispatching difficult. This chapter also explains how the simulation of dispatching the schedules proceeds.

Section §3.2 describes and analyzes the complexity of the sequential dispatching algorithm. Section §3.3 proposes and analyzes the complexity of the parallel dispatching algorithm along with motivating the requirement of the dispatcher. Section §3.4 describes the parameters involved in generating the constraint sets and the procedure of simulating the dispatching of a Partially Clairvoyant schedule. Section §3.5 describes work done in the areas.

3.2 Sequential Dispatching

In sequential dispatching, the dispatch algorithm switches between job execution and updating safety intervals of the remaining jobs. The dispatch algorithm can compute the safety interval using one of the following techniques:

- Update the safety intervals of all the remaining jobs, whose dispatch functions depend on the start and execution times of the completed job.
- Compute the safety interval of the next job only using the start and execution times of all the completed jobs(lazy evaluation).

In the first approach, the algorithm computes the safety intervals of the remaining $(n-i)$ jobs, immediately after a job J_i completes. Hence the update time decreases with the completion of jobs.

In the second approach, the number of constraints to be relaxed to obtain the safety interval is proportional to the i completed jobs. The update time increases as jobs finish.

3.2.1 Algorithm

In this thesis, the online dispatching algorithm (3.2.1) computes and updates the safety intervals of all the remaining jobs depending on the start and execution time upon completing a job.

Function SEQUENTIAL-ONLINE-DISPATCHER-FOR-J_a $(G = \langle V, E \rangle)$

1: Let $[l_{b_i}, r_{b_i}], (l_{b_i} < r_{b_i})$ denote the current safety interval of J_i.
2: set current time to 0.
3: **for** $(i = 1$ **to** $n)$ **do**
4: **if** (current-time $< l_{b_i}$) **then**
5: Sleep (l_{b_i}-current-time)
6: **end if**
7: **if** (current-time $\in [l_{b_i}, r_{b_i}]$) **then**
8: Execute job J_i
9: Update all safety intervals depending on (s_i, e_i)
10: **else**
11: Return (Schedule is not dispatchable)
12: **end if**
13: **end for**

Algorithm 3.2.1: Sequential Dispatcher for `<aph|stan|param>`

3.2.2 Complexity of Sequential Online Dispatching

The list of dispatch functions generated bound the start time of the job as shown in Table 3.1. In the case of standard constraints, the length of these lists is at most $O(n)$. For any schedule, the first job has a start time interval, $[a, b]$, independent of other jobs. Upon termination of job J_1, s_1 and e_1 can be plugged into $f_1()$ and $f_1'()$, thereby providing a safety interval $[a', b']$ for s_2. The same argument can be applied to the following jobs up to J_n. The dispatcher needs to determine the start time of the first job in the sequence,

Lower bound function	$\leq$ Start time $\leq$	Upper bound function
a	s_1	b
$f_1(s_1, e_1)$	s_2	$f'_1(s_1, e_1)$
$f_2(s_1, e_1, s_2, e_2)$	s_3	$f'_2(s_1, e_1, s_2, e_2)$
$\vdots$	$\vdots$	$\vdots$
$f_{n-1}(s_1, e_1, s_2, e_2, \ldots, s_{n-1}, e_{n-1})$	s_n	$f'_{n-1}(s_1, e_1, s_2, e_2, \ldots, s_{n-1}, e_{n-1})$

Table 3.1: List of parametric functions

after which the safety intervals can be computed. Since the length of the list is at most $O(n)$, the complexity of dispatching a job is $O(n)$.

3.3 Multiprocessor Dispatching

The online dispatcher updates safety intervals of the remaining jobs in parallel to job execution. The online dispatchers can be modeled using two control paradigms, viz., a master-slave model or a peer to peer model. In the multiprocessor cases, disjoint job sets are assigned to each processor. In the master-slave model, there is only one processor executing all the jobs while the rest of the processors update and report the safety intervals of job sets assigned to them. In the peer to peer model, each processors executes jobs in its job set and updates the safety intervals of jobs in its job set.

3.3.1 Motivation and Related work

A sequential online dispatching algorithm was proposed in [GPS95], for the schedules generated using the algorithm in [Sak94]. The computing overhead of the online dispatcher may cause Loss of Dispatchability due to the linear dispatch complexity.

The original single controller algorithm proposed in [Sub00] assumes that there are as many processors as the number of jobs n and each processor is assigned one job. The jobs are executed on a central processor which then broadcasts the start and execution time of the completed job to the other processors. The n supporting processors receive the start and execution times of a job J_k and update the safety intervals, by relaxing the 4 constraints between the job completed and the job assigned to them. The satellite processor k sends the safety interval of job J_{k+1}. This algorithm has $O(1)$ dispatch time per job and uses $O(n)$ space per processor. In Chapter 4 and 5, multi-processor dispatch algorithms using fixed number of processors,

much less than the number of jobs, for different memory architectures are proposed.

For the example stated in Section §2.4 in Chapter 2, assume the first two jobs take the worst case time and that the first job starts at time $t = 0$, then the third job has the safety interval [15, 16]. If the dispatcher takes more than one unit of time to compute the safety interval, then the third job cannot be dispatched.

Another factor which promotes the use of single and distributed controllers is the existence of distributed applications and critical systems in complex environments as discussed in Chapter 1. This modeling method allows execution time to vary as a parameter and ensures that the deadlines are met. The algorithms proposed are especially useful in parallel and distributed systems which require very high computing power and for control.

Embedded designers are conservative and use 8, 16 or 32-bit processors in most of their applications, which do not have the sophisticated architecture and instruction set support available in modern processors. NASA still uses the reliable IBM RISC6000 chips in some of its projects.

Automotive designers deploy microprocessors to control many automotive processes and parts such as cruise control, automatic transmission, fuel injection, braking and many more. When the cruise control is set, the controller maintains the speed executing complicated algorithms. The control requires processors to communicate with each other and exchange data. The designers are conservative on the processors they embed and prefer to use 8 bit processors that are reliable and exhaustively tested. A controller will be able to meet its deadlines better if its functions are distributed over some processors or it is provided with a computing cluster to do its computations.

Example (1): Consider another hard real-time system with a job-set $\mathcal{J} = \{J_1, J_2, J_3, J_4\}$. *Let the execution times of the jobs be as follows:*

- $e_1 \in [5, 7]$
- $e_2 \in [3, 5]$
- $e_3 \in [1, 3]$
- $e_4 \in [4, 6]$

Let the constraints imposed on the system be as follows:

- $s_1 + e_1 + 5 \leq s_2$
- $s_2 + e_2 \leq s_3$

- $s_3 + e_3 \leq s_1 + e_1 + 20$
- $s_1 + e_1 + 10 \leq s_3$
- $s_3 + e_3 + 2 \leq s_4$
- $s_4 + e_4 \leq 30$

The query (2.3) for the example above produces the following dispatch functions:

1. $0 \leq s_1 \leq \min(2,\ 9)$
2. $s_1 + e_1 + 5 \leq s_2 \leq \min(14,\ s_1 + e_1 + 10)$
3. $\max(s_1 + e_1 + 10,\ s_2 + e_2) \leq s_3 \leq \min(s_1 + e_1 + 15,\ 19)$
4. $s_3 + e_3 + 2 \leq s_4 \leq 24$

Assume that the first two jobs take the worst case time to complete and that the first job starts at time $t = 1$, then the second job has the safety interval [13, 14]. If the dispatcher takes more than one unit of time to compute the safety interval, then the second job cannot be dispatched. In a situation that J_2 starts at time $t = 13 + c$, where $c \in [0,\ 1]$; then J_3 has a safety interval of $[18 + c,\ 19]$. The example clearly shows the requirement of speed while dispatching jobs.

3.3.2 Parallel Dispatch Algorithm

The algorithm (3.3.1) uses a fixed number of processors much less than the number of jobs. One processor executes the jobs while the other update the safety intervals. However, the algorithm can be modified to distribute job execution across the processors.

In the analysis, we assume that the satellite processors complete relaxing all the constraints depending on the start and execution time of the previous job and are waiting for the next start and execution time before the execution of a job finishes.

3.3.3 Complexity of Multiprocessor Dispatching

After a processor completes executing a job J_i, the constraints which need to be computed to determine the safety interval $[l_{b_{i+1}},\ r_{b_{i+1}}]$ of J_{i+1} are as follows:

Function PARALLEL-ONLINE-DISPATCHER-FOR-J_a $(G = \langle V, E \rangle)$

1: Let $[l_{b_i}, r_{b_i}], (l_{b_i} < r_{b_i})$ denote the current safety interval of J_i.
2: set current time to 0.
3: **for** $(i = 1$ **to** $n)$ **do**
4: **if** (current-time $< l_{b_i}$) **then**
5: Sleep (l_{b_i}-current-time)
6: **end if**
7: **if** (current-time $\in [l_{b_i}, r_{b_i}]$) **then**
8: Execute job J_i
9: Update all safety intervals depending on (s_i, e_i) **in Parallel**
10: **else**
11: Return (Schedule is not dispatchable)
12: **end if**
13: **end for**

Algorithm 3.3.1: Parallel Dispatcher for `<aph|stan|param>`

1. $s_i + c_1 \leq s_{i+1}$
2. $s_i + e_i + c_1 \leq s_{i+1}$
3. $s_{i+1} \leq s_i + c_3$
4. $s_{i+1} \leq s_i + e_i + c_4$

where c_1, c_2, c_3 and c_4 are real numbers.

Since there are at most 4 constraints between job J_i and J_{i+1}, algorithm (3.3.1) takes at most $O(1)$ time, for each job sequenced before it. As stated in [Sub00], relaxing 4 constraints takes at most 4 additions and comparisons, i.e., $4 \cdot (T_{add} + T_{comp})$, where T_{add} and T_{comp} are the times taken to perform an addition and a comparison respectively.

Let w be the cost of communicating a floating point number to other processors. In the master slave model, the cost of communicating the start and execution time of a completed job and receiving the safety interval of the next job is $4 \cdot w$ while in the peer to peer model is $2 \cdot w$; both of which are constant. *The time required to compute the safety interval is $4 \cdot (T_{add} + T_{comp}) + C'$, where C' is a constant. The multiprocessor dispatching algorithms, complexity and implementation details are discussed in detail in Chapters 4 and 5.*

3.4 Experiment Design

3.4.1 Generation of Partially Clairvoyant schedules

A test-case is a set of jobs with execution time belonging to a certain time period and several constraints between the jobs. The duration between two adjoint jobs is capped by creating constraints depending on the execution time of the first job. Test-cases are created by varying the number of jobs or the execution time period or the threshold value of the cap. These test-cases are used to generate Partially Clairvoyant schedules.

A detailed description of the parameters required for the schedule generation is in Section §3.4.1. The procedure followed by the schedule generating algorithm (GA) is described in the Section §3.4.1.

Parameters

The parameters required by GA are as follows:

- Number of jobs n: The number of jobs in the schedule
- Execution time $[l, u]$: The lower and upper limit of the execution time of the jobs.
- Spacing time $[p, q]$: This is used to create constraints which would ensure that the next job would begin between $[p, q]$ seconds after the completion of a job. The value of p will prevent constraints which force the two jobs to be very close to each other while q prevents a large interval between the two jobs. p prevents the degree of closeness from being very small and q prevents degree of separation from being large.
- Number of constraints E: The number of standard constraints between jobs.

where l, u, p and q are real numbers.

Constraint Generation

We specify as inputs, the number of jobs n, the number of constraints E, the execution time $[l, u]$ and the spacing time $[p, q]$ along with a random seed, for generating the constraints. The generating algorithm (GA) does as follows:

- For each job, GA generates and prints two numbers between l and u ($l < u$), which bound the execution time of the job.
- Between every job J_i and $J_{i+1}(1 \leq i \leq n)$, GA generates standard constraints of the form $s_i + e_i \leq s_{i+1}$ and $s_{i+1} \leq s_i + e_i + c$ where c is a random number between p and q. The generator generates at least $2 \cdot n$ constraints.
- If $E > 2 \cdot n$ then $(E - 2 \cdot n)$ constraints between the finish times of two randomly chosen jobs (say J_x and J_y) such that a Partially Clairvoyant schedule would exist. If $x < y$ then a small negative real number $-l \leq c_1 \leq 0$ is generated such that $s_x + e_x \leq s_y + e_y + c_1$ which would be trivially true; and if $x > y$ then a very large real number c_2 is generated such that $s_x + e_x \leq s_y + e_y + c_2$ which also would be trivially true.

A large value of E increases the update time on the satellite processors.

3.4.2 Schedule Generation and Execution

The algorithm described in [Sub03] generates a Partially Clairvoyant schedule from a test-case generated by GA. The schedule is written into a file to be read by the online dispatcher.

The dispatcher reads the number of jobs, execution time periods, a random seed and the dispatch functions from the file. The dispatch functions are stored in a two-dimensional triangular array by each processor. Arrays are maintained to store the start time, execution time and execution time periods of the jobs.

The current time is set to zero when the dispatching starts. If the current time at the start of a job is within the safety interval of a job, the job is started. The job execution is simulated by generating a random number t' in the execution time interval of a job and performing a busy wait for t' seconds.

Further details about implementation on an architecture are discussed in the respective chapters.

3.5 Related Work

A sequential online dispatching algorithm was proposed in [GPS95], for the schedules generated using the algorithm in [Sak94]. The algorithm stores lists of dispatch functions and has dispatch time proportional to the number of jobs. The computing overhead of the online dispatcher may cause Loss of Dispatchability due to the linear dispatch complexity, i.e., the time after computing the safety interval (l_b, r_b) exceeds r_b.

A parallel online algorithm was proposed in [Sub00] for eliminating *Loss of Dispatchability* for Partially Clairvoyant schedules.

Traffic Alert and Collision Avoidance system (TCAS) is used in commercial aircrafts to avoid collisions. An imprecise computation technique to meet the necessary deadlines was proposed in [HFL95]. Each job is broken into a mandatory job and an optional job; the mandatory jobs are to meet strict end to end deadlines while the optional jobs are scheduled in between with intermediatory deadlines. In this situation, our algorithm helps in executing jobs on different processors such that all deadlines are met.

For control problems, [MFFR02] proposes to use flexible sampling and timing intervals to decide when to schedule jobs. The start time of jobs is a variable according to the controller but they use the worst case execution times to decide if a schedule exists. There are constraint sets which do not have a schedule in case the worst case execution time is assumed as shown in [Sub03]. In case the number of parameters in the system increase, the efficiency of the controller decreases due to the heavy computing required to schedule jobs. The algorithms proposed would reduce the computing load on the controller and ensures that the controller functions with high efficiency.

Chapter 4

Distributed Strategy

4.1 Introduction

Jobs are distributed among processors as to balance the load on each processor. The parallel dispatching strategy discussed in Section §3.3.2 can be implemented in many ways, two methods of implementing the algorithm are as follows:

1. Single controller: The jobs execute on one dedicated central processor, while the remaining processors update and report the safety interval of the next job, to be executed, to the central processor. The central processor can either transmit the start and execution time of the completed job directly to all the processors or employ different communicating schemes.

2. Multicast controller: Each processor computes the safety intervals and executes the set of jobs assigned to them. Different communication strategies can be employed as before. In this chapter, the processor that executes a job is made to transmit the start and execution time to all the remaining processors.

The above strategies are tested on two network topologies and the results are explained.

In this chapter, the original distributed dispatching algorithm proposed in [Sub00] is extended to distributed memory machines using a fixed number of processor for dispatching. The original algorithm required as many processors as the number of jobs. We propose, analyze and empirically test the dispatcher with Partially Clairvoyant schedules of different sizes and constraints. We analyze and explain the results obtained for both the algorithms.

The rest of this chapter is organized as follows: Section §4.2 describes the architecture and the algorithms used for distributed dispatching. It also analyzes the complexity of the algorithms and also explains the existence of a communicating processor through an analytical model. Section §4.3 explains fills in details about communication and timing while dispatching the schedule. Section §4.3 also describes the machine used and the modes of communicating on the machine. Sections §4.4 and Section §4.5 shows and explain the results obtained on dispatching job sets using the single controller and multicast controller algorithms respectively. Section §4.6 describes some work going on in the areas of distributed computing and clock synchronization and §4.7 lists the conclusions reached in our tests.

4.2 Architecture, Algorithm and Analysis

The machine consists of a group of processors which are connected through a high speed switch. Each processor has its own local memory to store its data and this data is communicated between processors through asynchronous messages in a shared network. The parallel programming paradigm employed is Single Program Multiple Data (SPMD).

In the Sections §4.2.2, it is assumed that jobs are distributed equally among the N processors and that a job J_i is assigned to $P_{i \mod N}$. This assumption is not completely unrealistic and the order of communicating (s_i, e_i) can be modified to ensure that the processor that executes the next job (J_{i+1}) receives (s_i, e_i) before the others.

4.2.1 Single Controller

Architecture

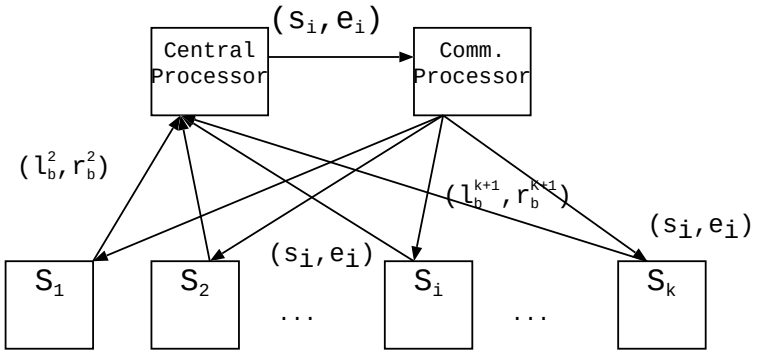

Figure 4.1: The single controller architecture for Partially Clairvoyant dispatching

The proposed architecture is shown in Figure (4.1). The central processor receives the safety interval from the satellite processors. The central processor executes a job and transmits the start and execution time of the executed job to the communicating processor. The communicating processor in turn sends the start and execution time to all the satellite processors. Each satellite processor S updates and reports the safety intervals of a class of jobs C preassigned to them. In case there is only one satellite processor, there will be no communicating processor.

Algorithm

After the central processor completes executing a job J_i, it transmits $(s_i,\ e_i)$ to the satellite processors where relaxing 4 constraints takes $4 \cdot (T_{add} + T_{comp})$ as explained in Section §3.3.3.

Let w_1 be the cost of transmitting a floating point number from one processor to another. The communication cost of transmitting $(s_j,\ e_j)$ is equal to $2 \cdot w_1$ if there is one satellite processor and $4 \cdot w_1$ for multiple satellite processors. Further, $2 \cdot w_1$ is required by a satellite processor to transmit $(l_{b_{j+1}},\ r_{b_{j+1}})$ to the central processor.

Hence algorithm 4.2.1 has a dispatch complexity of $O(1)$.

In the above analysis, the assumption in the Section 3.3.2 is assumed to hold.

Analytical Model

Consider the following analytic model of a distributed controller with processors of low computing speeds:

Let there be k satellite processors and let t_{send} be the time to communicate two floating point numbers to a satellite processor. Let t_{update} be the time taken by the satellite processors to compute the safety interval of the next job.

In case that a communicating processor does not exist, the central processor has to communicate the data to all the k satellite processors before it is ready to receive the safety interval of the next job. The time T' required in the following case is

$$T' = k \times t_{send} + t_{update} + t_{send} = (k+1) \times t_{send} + t_{update}$$

In case a communicating processor exists, the data is transferred to the communicating processor which in turn transmits to the satellite processors beginning with the processor which sends the next safety interval. The time T'' required in the following case is

$$T'' = 2 \times t_{send} + t_{update} + t_{send} = 3 \times t_{send} + t_{update}.$$

Function ONLINE-DISPATCHER-FOR-J_a ($G = \langle V, E \rangle$)

1: Let $[l_{b_i}, r_{b_i}], (l_{b_i} < r_{b_i})$ denote the current safety interval of J_i.
2: Let P denote the number of satellite processors.
3: **for** ($j = 1$ **to** n) **in parallel do**
4: **if** (central processor) **then**
5: **if** (current time $\in [l_{b_j}, r_{b_j}]$) **then**
6: Execute job J_j
7: Transmit (s_j, e_j) to communicating server
8: Compute the satellite processor S_k from which the safety interval is expected
9: Receive ($l_{b_{j+1}}$, $r_{b_{j+1}}$) from S_k
10: **else**
11: Report Schedule is not dispatchable
12: **end if**
13: **end if**
14: **if** (communicating processor) **then**
15: Compute the satellite processor S_k from which the safety interval is expected
16: Receive (s_j, e_j)
17: Transmit (s_j, e_j) to the satellite processors, beginning with S_k to S_{k-1}
18: **end if**
19: **if** (satellite processor S_m) **then**
20: Compute the satellite processor S_k from which the safety interval is expected
21: Receive (s_j, e_j)
22: **if** $S_k = S_m$ **then**
23: Update-constraints(j, $j+1$)
24: Send safety interval to central processor
25: Update-constraints(j, q) $\forall J_q \in C_k$ ($q \geq j+1$)
26: **else**
27: Update-constraints(j, q) $\forall J_q \in C_m$
28: **end if**
29: **end if**
30: **if** i = n **then**
31: Report schedule is dispatchable
32: **end if**
33: **end for**

Algorithm 4.2.1: Dispatcher for single controller

Function UPDATE-CONSTRAINTS(j, k) (s_j, e_j)

Relax constraints between J_j and J_k into absolute constraints.
Compare each absolute constraint with the existing safety interval for J_k
if (new constraint is non-redundant) **then**
  Update Safety Interval
else
  Leave the Safety Interval unchanged
end if

Algorithm 4.2.2: Update constraints function in Single controller dispatcher

It is clear that $T' > T''$ when $b > 2$. With a communicating processor, the load on the satellite processors increases as there is one less processor to update which requires the execution time to be greater than the update time of all the jobs. When the communication cost to transmit to all the satellite processors is large, a communicating processor helps.

4.2.2 Multicast Controllers

Architecture

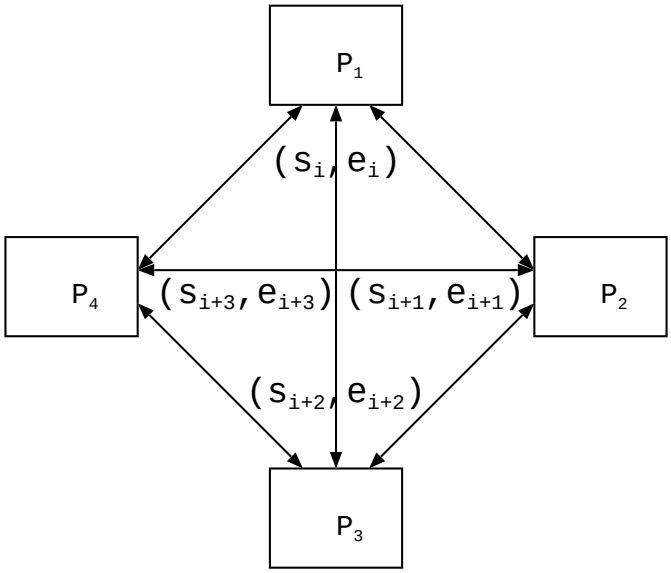

Figure 4.2: Multicasting architecture for Partially Clairvoyant dispatching.

The proposed architecture is shown in Figure (4.2). A processor P completes the job J_i and then transmits

(s_i, e_i) to the other processors. If $J_{i+1} \in C'$ of P', then P' computes the safety interval and immediately starts J_{i+1} in the safety interval. P' then updates the safety intervals of the remaining jobs with the start and execution times of the two jobs, J_i and J_{i+1} . The other processors update the safety intervals for the jobs assigned to them.

Algorithm and Analysis

After completing a job, the processor takes $2 \cdot w_1$ to send (s_i, e_i) to the processor P' that executes the next job. P' relaxes four constraints which takes $4 \cdot (T_{add} + T_{comp})$. Hence, algorithm 4.2.3 takes constant time to compute the dispatch interval for a job.

4.3 Experiment Design

4.3.1 Machine Specifications

Lemieux comprises 750 Compaq Alphaserver $ES - 45$ nodes and two separate front end nodes. Each computational node contains four $1 - GHz$ processors SMP with 4 Gbytes of memory and runs the $Tru64$ Unix operating system. A Quadrics interconnection network connects the nodes.

The Quadrics network has two building blocks, a programmable network interface called Elan and a low-latency high bandwidth communication switch called Elite. The Elan network interface links the high-performance, multi-stage Quadrics network to the nodes. The Elan also provides substantial local processing power to implement high-level message-passing protocols, such as MPI, in addition to generating and accepting packets to and from the network. The Elite switch provides 8 bidirectional links supporting two virtual channels in each direction, an internal 16×8 full crossbar switch and a bandwidth of $400MB/s$ with a latency of $35ns$.

We used MPI libraries in C to implement all the dispatchers.

Schedule Execution

In the single controller, the central processor sends (s_i, e_i) and waits for the safety interval of the next job from the satellite processors. The function, MPI_Wtime(), is invoked before sending the start and execution times to the communicating processor and after receiving the safety interval from a satellite processor. The time difference between the two function calls is added to the finish time of the completed job to obtain the

Function ONLINE-DISPATCHER-FOR-J_a

1: Let the set C_i for each processor P_i.
2: Let $[l_{b_i}, r_{b_i}], (l_{b_i} < r_{b_i})$ denote the current safety interval of J_i.
3: Let N denote the number of processors.
4: **if** (current time $\in [l_{b_1}, r_{b_1}]$) **then**
5: Execute job J_1
6: Send (s_1, e_1) to the other processors.
7: **else**
8: Report Schedule is not dispatchable
9: **end if**
10: **for** $(i = 2$ **to** $n)$ **in parallel do**
11: Determine the processor P on which job J_i has to be executed.
12: Determine the processor B on which job J_{i-1} was executed.
13: **if** (processor $P = P_k$) **then**
14: Receive (s_{i-1}, e_{i-1})
15: Update Constraints between J_{i-1} and J_i to determine the safety interval (l_{b_i}, r_{b_i}).
16: **if** (current time $\in [l_{b_i}, r_{b_i}]$) **then**
17: Execute job J_i
18: Send (s_i, e_i) to the other processors.
19: **else**
20: Report Schedule is not dispatchable
21: **end if**
22: Update-constraints-two(i, q) $\forall J_q \in C_k$
23: **else**
24: Receive (s_{i-1}, e_{i-1})
25: Update-constraints(i, q) $\forall J_q \in C_k$
26: **end if**
27: **end for**

Algorithm 4.2.3: Multicast Dispatcher for distributed controllers

Function UPDATE-CONSTRAINTS(j, k) (s_j, e_j)

Relax constraints between J_j and J_k into absolute constraints.
Compare each absolute constraint with the existing safety interval for J_k
if (new constraint is non-redundant) **then**
  Update Safety Interval
else
  Leave the Safety Interval unchanged
end if

Algorithm 4.2.4: Update constraints between Job J_j and J_k

Function UPDATE-CONSTRAINTS-TWO(j, k) (s_{j-1}, e_{j-1}, s_j, e_j)

Relax constraints between jobs (J_{j-1}, J_k) and jobs (J_j, J_k) into absolute constraints.
Compare each absolute constraint with the existing safety interval for J_k
if (new constraint is non-redundant) **then**
  Update Safety Interval
else
  Leave the Safety Interval unchanged
end if

Algorithm 4.2.5: Update constraints between Jobs (J_{j-1}, J_k) and (J_j, J_k)

start time of the next job. If the start time is within the received safety interval, the next job is started.

In the multi-cast approach, the control passes from one processor to the other with every job. In order to avoid clock synchronization and drift problems of the processors in our implementation, the current time is treated as the global clock time and is sent from one processor to the other with the start and execution times. The communication time is measured by dividing the time required to send a message to and immediately receive a message from the next processor by 2 after completing a job. This approximately simulates the time that would be required by the next processor to receive a message. In case the receiving processor is still updating constraints, then the wait time is automatically added to the communication time.

In the multi-cast approach, the processor which executes a job sends the start and execution times to all the other processors, while the other processors receive and update the safety intervals.

4.3.2 Communication

The cluster has two layers of communicating messages between processors; intra-node communication and inter-node communication. Intra-node communication is communication between processors of the same node through the ELAN interface, while inter-node communication is communication between processors on different nodes through the Quadrics interconnect network. Thus, two sets of experiments are performed for each implementation of the dispatchers. In the first set of experiments, all processors are chosen from the least number of nodes containing them. For example, an experiment with 9 processors requires 3 nodes; *all* 8 *processors of the* 2 *nodes and* 1 *processor of the third node.* There is substantial intra-node communication in this model along with inter-node communication. The second set of experiments chooses one processor per node resulting in inter-node communication only.

4.3.3 Dispatch Variables

In the tests, we observed serious overshoots in the update times taken by satellite processors. These overshoots occur due to other system processes using the system resources or due to uncontrolled traffic over the network that increases the response time taken by the cluster.

We observed the time taken to compute the safety interval in many experiments. Figure (4.3) plots the frequency of the observed update times taken by the single controller dispatcher while Figure (4.4) plots the frequency of the observed update times by the multicast dispatcher. Figures (4.3) and (4.4) show the update times are usually within certain intervals of time and that the overshoots are more than ten times the frequently observed update times. Accordingly, we categorized update times that are ten times greater

than the previous update times as overshoots and such observations were neglected by taking the previous update time. These abnormal overshoots can be safely neglected as real-time systems use dedicated machines with predictable performance. In case there are three consecutive overshoots, the observed update time was considered as the actual time taken to compute the safety interval. The above condition is necessary to check for situations where the concerned satellite processor is still updating safety intervals with the previous start and execution times, i.e., assumption in Section §3.3.2 fails.

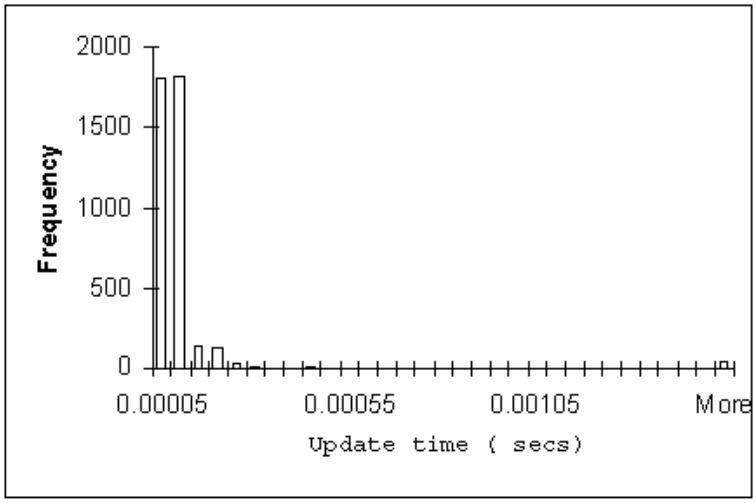

Figure 4.3: The frequency histogram of the observed update times by the single controller dispatcher.

4.4 Empirical Analysis of Single Controller

We generated Partially Clairvoyant schedules with the number of jobs increasing from 1000 to 9750 in steps of 250. The execution time duration of the jobs was ($1\,ms$,$5\,ms$) and the spacing time was chosen to be ($1\,ms$,$5\,ms$), ($0.5\,ms$,$1\,ms$) and ($0.1\,ms$,$0.5\,ms$). In the tests, we selected the processors either by choosing one processor per node or by choosing the processors from the minimum number of nodes containing them.

Figure (4.5) plots the update time taken by the single controller dispatcher for schedules of different size with varying number of processors. It is observed that the update time of the single controller dispatcher is almost constant and in the range of 10^{-5} seconds, while the update time of the sequential dispatcher increases into milliseconds, as the number of jobs increases. This figure show that the single controller dispatcher has very less update time compared to the sequential dispatcher.

We conducted experiments to test the dispatchability of various Partially Clairvoyant schedules by varying the number of processors. Each entry in Table 4.1 shows the largest job set successfully dispatched by

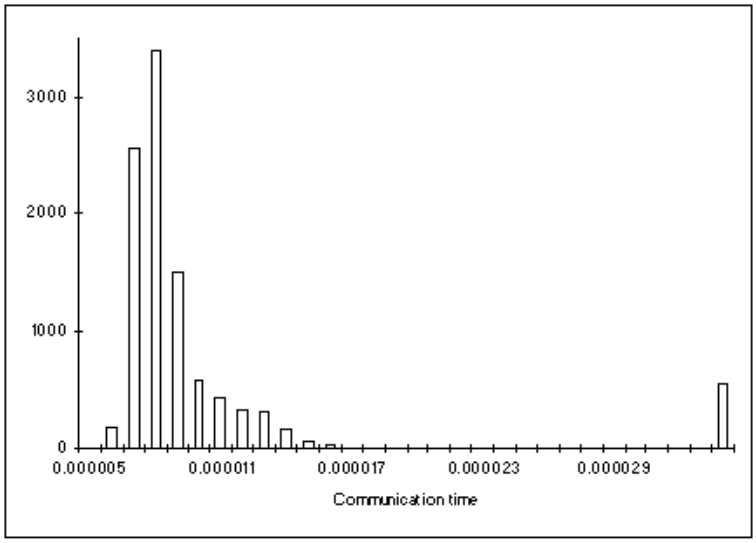

Figure 4.4: The frequency histogram of communication times taken by the multicast dispatcher.

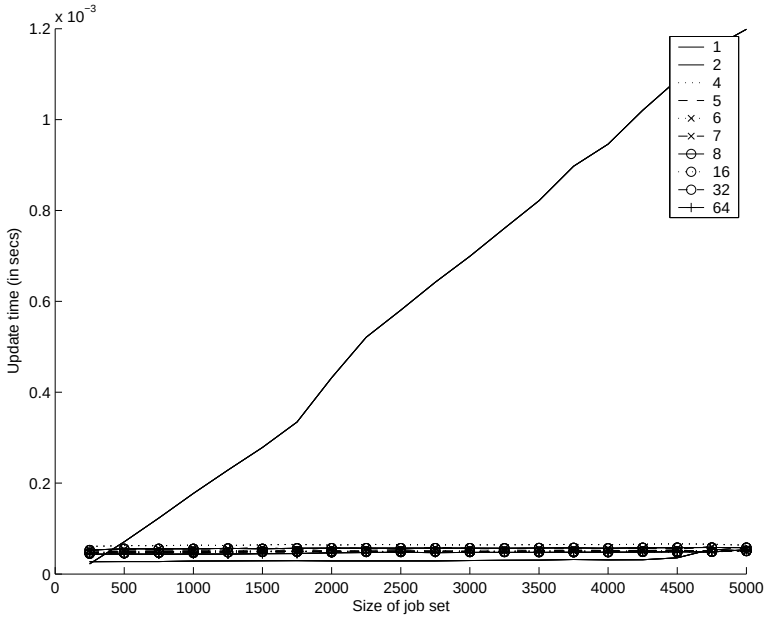

Figure 4.5: Plot of the Update time taken versus the number of jobs as the number of satellite processors are increased for a single controller with a communicating processor.

the dispatcher with the concerned processor allocation for schedules with spacing times of ($1\,ms$, $5\,ms$) and ($0.5\,ms$,$1\,ms$). Figures (4.6) and (4.7) show the results obtained with schedules having a spacing time of ($0.1\,ms$,$0.5\,ms$) as the number of satellite processors are increased. A dot in Figures (4.6) and (4.7) represents that the schedule with the concerned number of jobs was not dispatched by the concerned number of processors.

	Packing processors in nodes		One processor per node	
Processors	$1\,ms - 5\,ms$	$0.5\,ms - 1\,ms$	$1\,ms - 5\,ms$	$0.5\,ms - 1\,ms$
1	3000	1500	3000	1500
2	9750	9750	9750	9750
4	7500	7500	9750	9750
5	8750	8750	9750	9750
6	8750	8500	9750	9750
7	7500	7500	9750	9750
8	7500	6000	9750	9750
9	8500	8250	9750	9750
10	8000	7750	9750	9750
11	7250	7250	9750	9750
12	7500	7250	9750	9750

Table 4.1: Largest job set dispatched for varying number of processors by the single controller dispatcher with a communicating processor.

Table 4.1 shows that the size of job sets dispatched is greater when the spacing time between the jobs is greater. This is observed as the allowed time from the completion of a job to the start of the next job is greater when the spacing time is larger, thereby allowing the satellite processors to compute and report the safety interval to the central processor.

Another interesting observation in Table 4.1 is that the dispatcher with multiple satellite processors does not dispatch large job sets with around 8750 jobs. The satellite processors need to access different memory locations for each update they make as the size of the job set increases. This causes frequent page faults on every processor which slows down the throughput of the node. This explains the dispatching of all the job sets by the single controller dispatcher in Table 4.1 when one processor is chosen per node. Figures (4.6) and (4.7) also show that there are lesser number of job sets not dispatched in case of Figure (4.7) where one processor from one node was chosen. An alternate explanation could be based on the load of the ELAN layer. When a message is to be sent between two processors in a node, the ELAN layer needs to read the

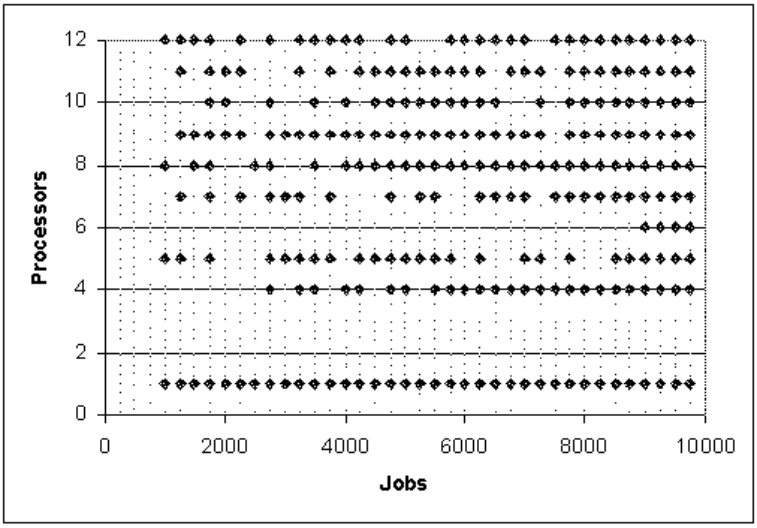

Figure 4.6: In the above tests, the single controller chooses multiple processors per node. For a job set of spacing time $[0.1\,ms, 0.5\,ms]$ and a given number of processors, a dot represents that the job set was **not** dispatched by the single controller with a communicating processor.

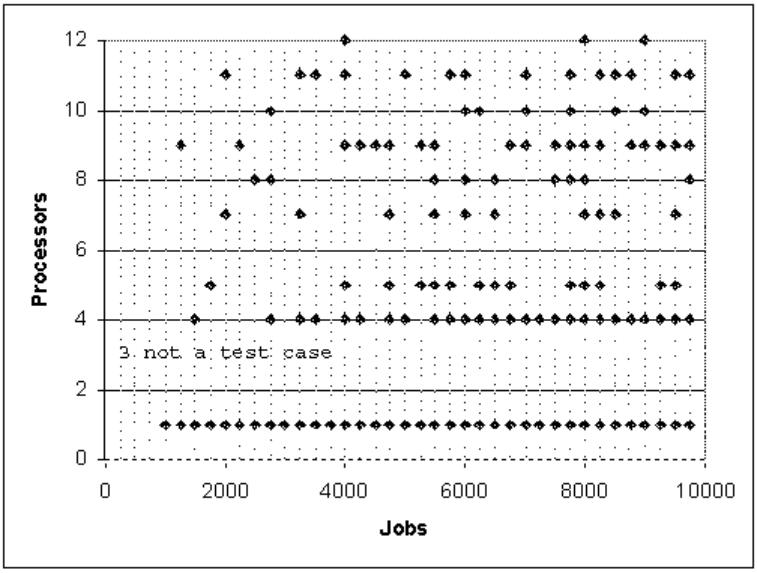

Figure 4.7: In the above tests, the single controller chooses one processor per node. For a job set of spacing time $[0.1\,ms,\ 0.5\,ms]$ and a given number of processors, a dot represents that the job set was **not** dispatched by the single controller with a communicating processor.

data from the memory of the sender and copy it to the memory of the receiver. In situations where data is sent to processors in other processors also, the ELAN sends the data through the Elite switches. In case the concerned satellite processor is sending the safety interval from the same node, the ELAN might queue and delay the safety interval.

In the above experiments, the time taken by the central processor to send the start and execution time to communicating processor was also measured. In order to avoid the communication delay that occurs when data moves up and down the protocol layers of the ELAN interface, we modified the architecture to exclude the communicating processor and the central processor transmits data directly to the satellite processors in the same order. Similar experiments were conducted with this architecture and the results obtained are listed in Table 4.2 and Figures (4.8) and (4.9). Figures (4.8) and (4.9) show the effect of packing processors in a node and choosing one processor per node respectively for Partially Clairvoyant schedules with a spacing time of $[0.1\,ms, 0.5\,ms]$. A dot in the Figures (4.8) and (4.9) for a given value job set indicates that the job set was not dispatched by the dispatcher with the concerned number of satellite processors.

	Packing processors in a node		One processor per node	
Processors	$1\,ms - 5\,ms$	$0.5\,ms - 1\,ms$	$1\,ms - 5\,ms$	$0.5\,ms - 1\,ms$
1	3000	1500	3000	1500
2	7500	6000	7500	6500
3	8750	8750	9750	9750
4	7750	7500	9750	9750
5	9750	9750	9750	9750
6	8500	8750	9750	9750
7	8000	7500	9750	9750
8	7500	7500	9750	9750
9	8750	8750	9750	9750
10	7500	7750	9750	9750
11	7750	7500	9750	9750
12	7500	7500	9750	9750

Table 4.2: Largest job set dispatched for varying number of processors by the single controller dispatcher **without** a communicating processor.

The results obtained are similar to the results obtained with the presence of a communicating processor. The experiments were conducted in a shared environment which might have lead to noise in the observations. However, this could not be verified as it was not possible to acquire the system in a dedicated mode. Due to

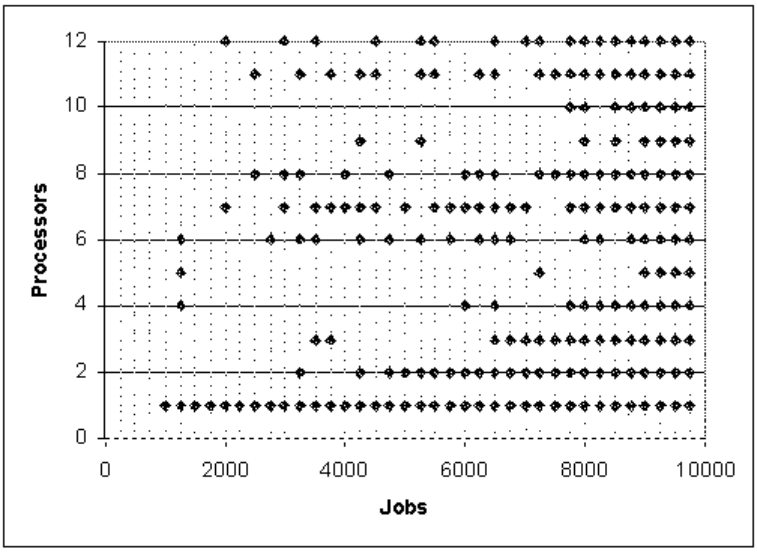

Figure 4.8: In the above tests, the single controller chooses multiple processors per node. For a job set of spacing time $[0.1\,ms, 0.5\,ms]$ and a given number of processors, a dot represents that the job set was **not** dispatched by the single controller **without** a communicating processor.

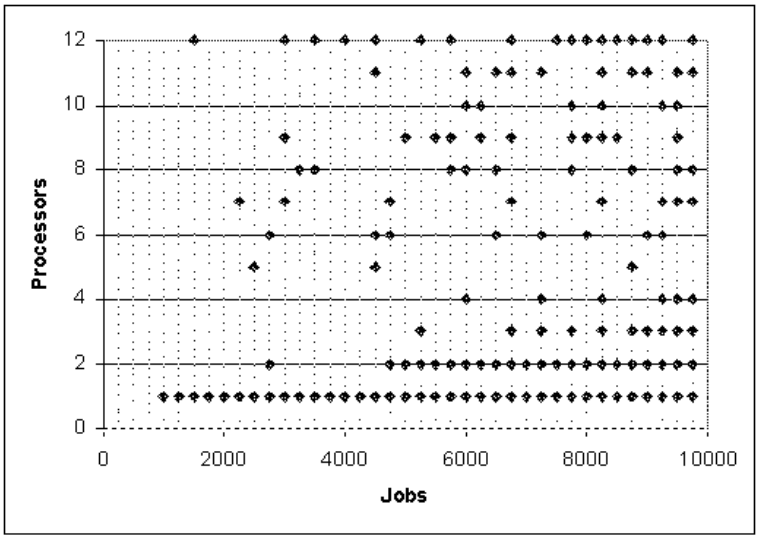

Figure 4.9: In the above tests, the single controller chooses one processor per node. For a job set of spacing time $[0.1\,ms, 0.5\,ms]$ and a given number of processors, a dot represents that the job set was **not** dispatched by the single controller **without** a communicating processor.

lack of resources, the point where the presence of a communicating processor would help was not exclusively determined. The analytical model described in Section §4.2.1 motivates the existence of a communicating processor.

4.5 Empirical Analysis of Multicast controller

4.5.1 Using all the processors in a node

The multicast dispatcher was tested using the same set of experiments as in Section §4.4 and the results are in Figure (4.10) [1]. The results observed are similar to the single controller case in Section §4.4 in that each processor set has different job sets that it can dispatch and all the processors shown in the figure fail to dispatch job sets larger than 8750.

It was concluded that multi-processor dispatchers dispatch more job sets than the serial dispatcher and that different number of processors dispatch different job sets. The failure in dispatching the large job sets is due to the page faults and the network load that decrease the throughput of the processors in the node.

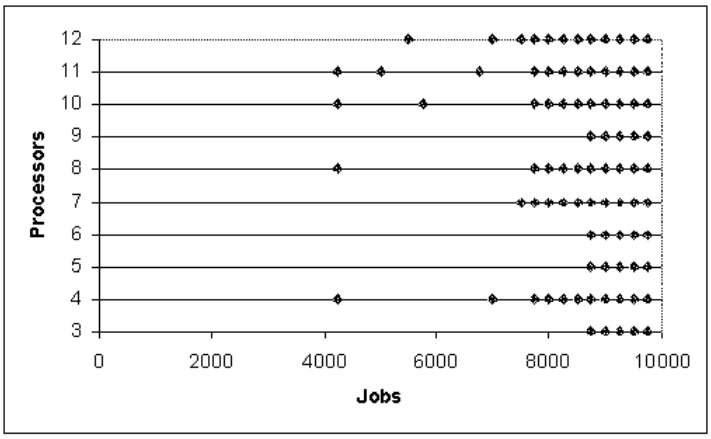

Figure 4.10: In the above tests, the multicast controller chooses multiple processors per node. For a job set of spacing time $[0.1\,ms, 0.5\,ms]$ and a given number of processors, a dot represents that the job set was **not** dispatched by the multicast dispatcher.

[1] These experiments were conducted by Ashraf

4.5.2 Using one processor per node

In these experiments, the processor allocation and test sets are similar to Section §4.4. Figure (4.11) [2] shows that this implementation is better than the implementation in Section §4.5.1 as most of the jobs are dispatched. It was concluded that the processor sets dispatch more jobs when the number of page faults and the load on the ELAN layer is less.

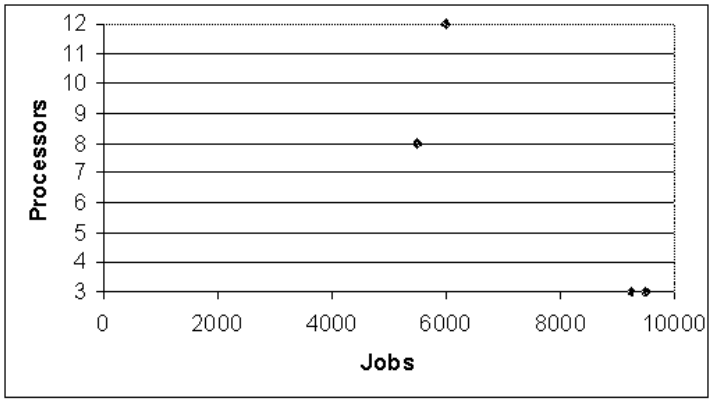

Figure 4.11: In the above tests, the multicast controller chooses one processor per node. For a job set of spacing time $[0.1\,ms, 0.5\,ms]$ and a given number of processors, a dot represents that the job set was **not** dispatched by the multicast dispatcher.

4.6 Related work

While assuming worst case execution time, [MLWP02] uses an interval to model start times. They decide the schedulability of such a system and are exploring situations in distributed applications where the global clock and the local clocks are not synchronized with each other.

A best-case execution time analysis is described in [HKR01], to reduce the jitter and extend the distributed scheduling analysis, so as to yield more accurate upper and lower bounds on system response times. An attempt to make real-time communication on the ethernet more predictable by limiting the packet-arrival rate allowed into the Medium Access Control(MAC) layer is proposed in [KS03]. A middleware architecture was developed in [BBP+01], that proposes to use small microcontrollers as computation nodes in distributed real-time systems.

Operators only communicate the high-level goal and deadlines to a spacecraft, which in turn does the planning

[2] These experiments were conducted by Ashraf

and scheduling. [DKT99] proposes distributed approaches for onboard planning and scheduling that help the spacecraft perform as an autonomous agent. For real-time implementation of control applications, [Tr98] proposes a set of requirements and investigates important sources and characteristics of time-variations in distributed computer systems .

4.7 Conclusion

We implement two models of distributed memory dispatchers and both were observe to have a lower update time than the sequential dispatcher. We study the effect of choosing multiple processors across nodes and find that number of memory accesses effects dispatchability, i.e., the dispatcher fails to dispatch schedules of smaller size.

We show the effect of spacing time on the single controller dispatcher. We were not able to determine the point where having a communicating processor would make dispatching superior than without a communicating processor.

We also observe that the peer to peer model is superior in dispatching than the single controller dispatcher for the schedules created. The peer to peer model depicts that "Loss of Dispatchability" noticed for larger schedules is clearly due to the number of page faults.

A different communicating model based on pipeline communications can be tested for the master slave and the peer to peer model.

Chapter 5

PRAM Strategy

5.1 Introduction

The parallel dispatching algorithm in Section §3.3.2 is implemented as a single controller algorithm. The central process executes a job and flushes the start and execution time of the completed job to the shared memory. The central processor updates a flag on which the other processors wait and waits on another flag to be updated by one of the satellite processors on writing the safety interval of the next job in the shared memory. The other processors (satellite processors) update the safety intervals of the jobs assigned to them.

In this chapter, the original distributed dispatching algorithm proposed in [Sub00] is extended for shared memory machines with fixed number of processors. The original algorithm required the number of processors equal to be the number of jobs and assumed that the time to communicate data to them was constant. We implement the algorithm with fewer processors making an assumption and test the dispatcher with Partially Clairvoyant schedules of different sizes. The results obtained are listed and explained.

The rest of this chapter is organized as follows: Section §5.2 describes the architecture of the machine model and analyzes the algorithm. Section §5.3 describes the machine used and identifies the run-time variables and overshoots in the experiments, Section §5.4 lists and explains the results obtained on testing the dispatcher with Partially Clairvoyant schedules of different sizes and constraints. Section §5.6 lists the conclusions made from our experiments.

5.2 Architecture, Algorithm and Analysis

The CREW-shared memory architecture is discussed in detail in [Ja'92]. Each processor has a separate local memory in addition to the common shared memory and maintains a copy of the data in its local memory. Any changes to the shared data are made in its local memory and the data is flushed for memory coherence. Memory Coherence depends on the protocol followed [LH89], i.e., the shared data variable is marked invalid in the other memories as soon as a local copy is changed or a flush command needs to be executed by the processor to achieve memory coherence. A processor requires far less time to access data from its memory than data in the memory of another processor. The particular machine we are working on is a Non-Uniform Memory Access (NUMA) machine. While reading a shared variable, the value resulting from the most recent write is loaded into the local memory.

5.2.1 Architecture

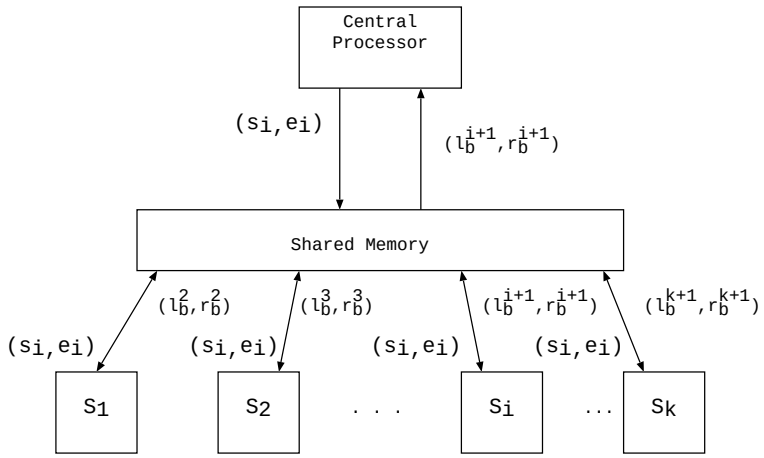

Figure 5.1: Shared Memory Dispatcher Architecture

The processors share data with each other through the shared memory as indicated in Figure (5.1). The variables (s_i, e_i) and ($l_{b_{i+1}}$, $r_{b_{i+1}}$) are stored in the shared memory. The central processor C executes Job J_i and stores (s_i, e_i) into the memory. Then C updates a flag f_1 and waits on another flag f_2. Each satellite processor S_j updates and reports the safety intervals for a class of jobs C_j. The satellite processor S_m which has $J_{i+1} \in C_m$ writes the safety interval ($l_{b_{i+1}}$, $r_{b_{i+1}}$) in the memory and updates the flag f_2. On updating the safety intervals of all the remaining jobs in their class, the satellite processors wait for f_1 to be updated by C.

In this implementation, there are no communication costs as compared to a distributed model but there is a

cost for achieving memory coherence. However, the processors are required to flush the start and execution times and the safety intervals for every job, which is an extra overhead for this algorithm. *In this algorithm, assumption in Section 3.3.2 is assumed to hold.*

5.2.2 Algorithm and Analysis

The 4 constraints between job J_i and J_{i+1}, constant time to compute $[l_{b_{i+1}}, r_{b_{i+1}}]$.

Let w_1 be the cost of writing a floating point number to the shared memory such that the data is coherent through out the memory. C requires to flush the present values of (s_i, e_i, f_1) to the memory. S_k will have to write $(l_{b_{i+1}}, r_{b_{i+1}}, f_2)$ in the memory.

The time required to compute the safety interval is $4 \cdot (T_{add} + T_{comp}) + 6 \cdot w_1$. Hence algorithm (5.2.1) has $O(1)$ dispatch complexity.

The algorithm 5.2.1 updates the dispatch functions in parallel to the execution of the next job. Let k be the number of processors. In such a case, each processor has to update constraints between the completed job and a fraction $(=\frac{1}{k})$ of the remaining jobs. In case n is very large, the time required to update the constraints is larger than the execution time of the current job and might cause the next job to lose dispatchability. Increasing the number of processors would help dispatchability if the memory coherence cost is not great.

5.3 Empirical Analysis

5.3.1 Machine Description

We used SGI Origin2000 of NCSA to test an implementation of the algorithm. The hardware specifications of the machine and environment are listed in Tables 5.1 and 5.2 respectively. The jobs were submitted in the batch queue.

Schedule Execution

A job is executed only if the current time at the beginning of the job is within the safety interval of the job. The central processor then updates the f_1 and writes the triplet (s_i, e_i, f_1) to the memory. The central processor waits for an updated value of f_2 before reading the safety interval of the next job. The C function gettimeofday() is used to find the time T_{bef} before writing (s_i, e_i, f_1) into the shared memory and the time

Function SHARED-ONLINE-DISPATCHER-FOR-J_a $(G = \langle V, E \rangle)$

1: Let $[l_{b_i}, r_{b_i}], (l_{b_i} < r_{b_i})$ denote the current safety interval of J_i.
2: Let P denote the number of satellite processors.
3: **for** $(i = 1$ **to** $n)$ **in parallel do**
4: **if** (central processor) **then**
5: **if** (current-time $< l_{b_i}$) **then**
6: Sleep (l_{b_i}-current-time)
7: **end if**
8: **if** (current-time $\in [l_{b_i}, r_{b_i}]$) **then**
9: Execute job J_i
10: Save (s_i, e_i) to memory
11: Update $flag_1$ and save to memory
12: Wait till $flag_2$ is updated
13: Read $(l_{b_{i+1}}, r_{b_{i+1}})$ from memory
14: **else**
15: Return (Schedule is not dispatchable)
16: **end if**
17: **end if**
18: **if** (satellite processor S_m) **then**
19: Compute S_k, the satellite processor required to report the safety interval
20: Wait till $flag_1$ is updated
21: Read (s_i, e_i)
22: **if** $S_k = S_m$ **then**
23: Update-constraints$(i, i + 1)$
24: Write safety interval to memory
25: Update $flag_2$ and write to memory
26: Update-constraints(i, q) $\forall JobsJ_q \in C_k$
27: **else**
28: Update-constraints(i, q) $\forall JobsJ_q \in C_m$
29: **end if**
30: **end if**
31: **if** i = n **then**
32: Return (schedule is dispatchable)
33: **end if**
34: **end for**

Algorithm 5.2.1: Shared Dispatcher for `<aph|stan|param>`

Function UPDATE-CONSTRAINTS(i, q) (s_i, e_i)

1: Relax constraints between J_i and J_q into absolute constraints of J_q.
2: Compare each absolute constraint with the existing safety interval for J_q
3: **if** (new constraint is not redundant) **then**
4: Update Safety Interval ($[l_{b_q}, r_{b_q}]$)
5: **else**
6: Leave the Safety Interval unchanged
7: **end if**

Algorithm 5.2.2: Update function of Shared Dispatcher for `<aph|stan|param>`

Component	Description
Architecture	Distributed Shared Memory
Processors	MIPS R10000
Available number of processors	64 (or 128)
Clock Speed	250 MHz or 195 MHz
Instruction Cache Size	32Kbytes
Data Cache Size	32 Kbytes
User Virtual Address Space	4 GB
Interconnect between machines	Gigabit Ethernet

Table 5.1: Machine specifications of SGI Origin2000 of NCSA.

Component	Description
Operating System	Irix 6.5
Compiler	C
Programming Models	OpenMP
Floating Point Format	IEEE
Batch System	Load Sharing batch system

Table 5.2: Software Specifications of SGI Origin

T_{aft} after reading the safety interval ($l_{b_{i+1}}$, $r_{b_{i+1}}$) from the memory. The time difference $T_{aft} - T_{bef}$ gives the time required to update and report the safety interval to C. The simulation starts at time $t = 0$ and proceeds by adding $s_i + e_i + T_{aft} - T_{bef}$ to obtain the time at the instant C is ready to execute the job J_{i+1}.

5.3.2 Runtime Approximations

The time required to write the start time, execution time and the flag in the memory depends on various factors such as the load on the processors, the system bus and the number of memory requests. The wait depends on the time, the concerned satellite processor takes to read the updated flag f_1 and compute the dispatch functions.

With the execution of jobs, the number of constraints to be updated decrease and the update time should decrease, but a few overshoots were observed in the update time. These overshoots are caused by the load on the workstation and the scheduling policy of threads by the operating system *Iris*. Figure (5.2) shows the frequency of the observed update times in various intervals while dispatching a job set of 5000 jobs on 16 processers. Figure (5.2) also shows that the update times are heavily concentrated in a certain time interval and that a few discrepancies are located far away from this interval. Figure (5.2) shows the time difference between recurring update times and suggests a multiplicative factor after which the update time can be categorized as overshoots. The frequent update times were observed to be within 4 times the mean update time. These overshoots were detected by checking if the current update time is greater than 4 times the previous update time. The overshoots in the update time were neglected by taking the previous update time, atmost for five consecutive overshoots, after which the observed update time is considered as the actual update time. These overshoots were neglected as real-time systems require and use dedicated machines. The test environment on Origin machine is batch, where the response times depend on the load on the machine.

5.4 Results

In the experiments conducted, the number of processors or the distribution of threads across processors could not be controlled. The number of threads executing in parallel was the only parameter that could be set before submitting a job. The operating system then schedules the threads balancing the load across the processors. The optimal condition for the dispatcher would be for one thread to execute on one processor alone, equivalent to setting the desired number of processors. In all other cases, multiple threads run on a processor increasing the load on the satellite processors, which decreases the efficiency of the dispatcher.

The update time of the sequential dispatcher increases linearly with the number of jobs. After a certain

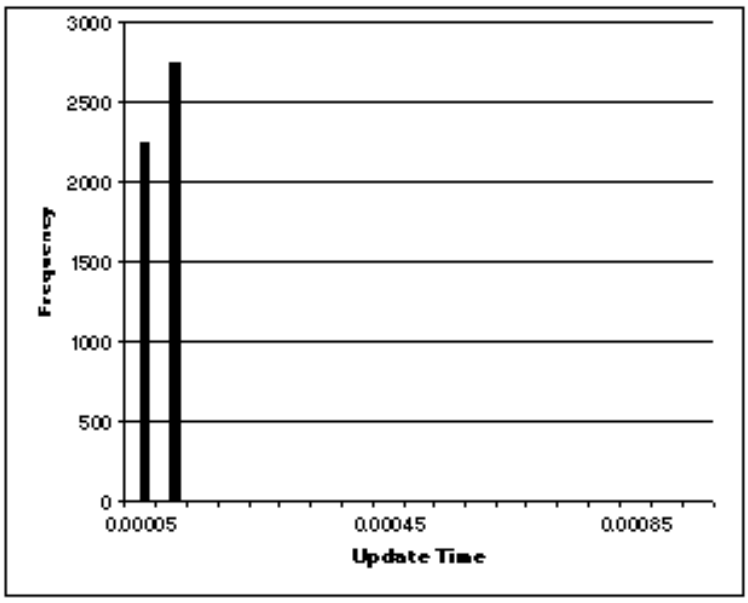

Figure 5.2: Observed update time frequency for 5000 jobs on 16 processors

number of jobs N_1, the update time is greater than the spacing time and the job set would no longer be dispatchable.

We conducted experiments to measure the update time of the sequential and the shared memory dispatchers. For the sequential dispatcher, the update time of all the jobs was measured by finding the time T_{bef} at the completion of a job and the time T_{aft} after completing the update of all the dispatch functions. While in the case of the shared dispatcher, T_{bef} is the time before writing the start and execution time of the job completed and T_{aft} is the time after reading the safety interval of the next job. In both the cases, the difference between T_{aft} and T_{bef} gives the update time required to compute the safety interval of the next job. The multi-processor dispatcher takes around the same update time for any number of jobs. The update time includes the time required to read (s_i, e_i, f_1) and update 4 constraints between J_i and J_{i+1} and write back ($l_{b_{i+1}}$, $r_{b_{i+1}}$, f_2). The satellite processors update the remaining constraints in parallel with the execution of job J_{i+1}. Figure (5.3) and (5.4) plot the update time in seconds as the number of jobs in the schedule increase.

Figure (5.3) compares the update time of the sequential dispatcher and the shared dispatcher with two processors. Figure (5.3) shows that the update time of the sequential dispatcher increases with the number of jobs. The shared dispatcher with two processors is almost constant and takes at most $2.5 \times 10^{-5} s$ to find the safety interval of the next job.

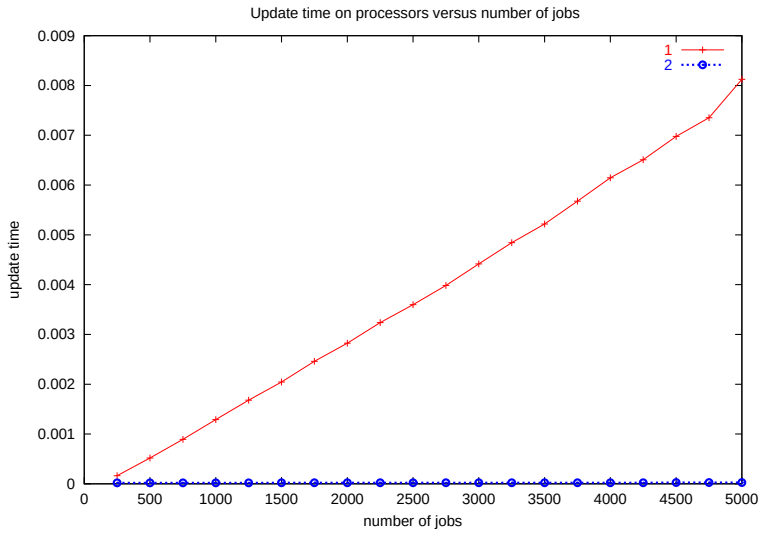

Figure 5.3: Plot of update time of single processor dispatcher and multi-processor dispatcher with 2 processors versus number of jobs.

Using best fit, we calculated the equation of line passing through the single processor update times to be $y = a \cdot x + b$, where $a = 1.52 \times 10^{-6}$ and $b = -2.214 \times 10^{-4}$ were obtained. Using the above line and the maximum update time of the two processor shared dispatcher, it was arithmetically calculated that after 163 jobs the shared memory dispatcher with two processors is superior to the sequential dispatcher.

Figure (5.4) plots the update time taken by the shared memory dispatcher for schedules of different size with varying number of processors. It is observed that the update time of the shared dispatcher is almost constant and in the range of 10^{-5} seconds, while the update time of the sequential dispatcher increases into milliseconds, as the number of jobs increases. Figure (5.3) and (5.4) show that the shared memory dispatcher has very less update time compared to the sequential dispatcher.

Figure (5.4) shows that the update time of the shared dispatcher increases with the number of processors, which can be accounted to the increase in the time taken to obtain a lock by a processor on the shared data, as processors are augmented. In case the number of jobs increases, the number of safety intervals to be updated increases and the satellite processors would have an update time linearly increasing with the jobs as the sequential dispatcher in Figure (5.3). The increase in the update time would make the assumption in Section §3.3.2 void as the satellite processors would be updating safety intervals when the central processor updates flag f_1 after executing a job. Following which, the dispatcher would not be able to dispatch the next job if the concerned satellite processor was delayed. On increasing the number of satellite processors, the number of safety intervals to update per processor decreases and hence the update time. The number of

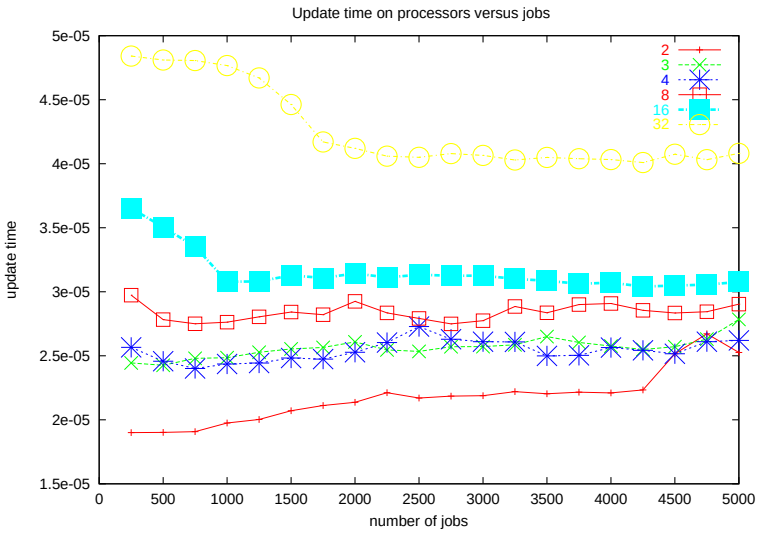

Figure 5.4: Update time taken by dispatcher versus number of jobs as the number of processors are increased.

processors should be increased till the time to achieve memory coherence is less that the spacing time, after which the assumption will not be valid.

The shared and sequential dispatcher were tested with job sets of different sizes. These experiments were conducted thrice with different random seeds. Table 5.3 summarizes the results of the experiments performed. Similar results were observed in the three cases, i.e., the schedules were dispatchable or not dispatchable. In all the three cases, the sequential dispatcher broke at different jobs as it took longer to update the safety intervals of all the jobs. While in the case of the shared dispatcher, the update times were small and the jobs were dispatched in their safety intervals.

Processors	Number of Jobs							
	250	500	750	1000	2000	3000	4000	5000
1	√	√	√	√	×	×	×	×
2	√	√	√	√	√	√	√	√
3	√	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√	√

Table 5.3: Results of dispatching job sets of different size using single and multiple processors. √ is when the schedule was successfully dispatched and × was not. $[l, u] = [1ms, 5ms]; [p, q] = [1ms, 5ms]$

5.4.1 Scalability

In this section, the effect of increasing the number of processors on dispatchability of job sets is shown. The size of the Partially Clairvoyant schedules generated was increased from 1000 up to 9750 in steps of 250. The jobs in the schedules have execution time periods varying between one to five milliseconds and the spacing time between two adjacent jobs to be between one-tenth to one-half a millisecond.

We varied the number of processors used by the dispatcher and found the largest job set successfully dispatched by the dispatcher. Table 5.4 summarizes the results of the experiments conducted and shows the largest job set up to which all the schedules were dispatched with a certain number of processors. It is observed that the dispatcher with fewer processors failed as the size of the job set increased. An increase in the size of the job set causes an increase in the update time on the satellite processors which would delay the satellite processors from reading the start and execution time of the next job as soon as it is stored in the memory by the central processor and results in the breaking down of the schedule. Figure (5.5) plots the largest job set dispatched by a given set of processors and shows the scalability of the shared memory algorithm. With 10 processors, all the schedules created were dispatched and the exact number of jobs up to which 10 processors would succeed was not found.

Processors	Maximum size of job set dispatched
1	100
2	2500
3	3500
4	4250
5	5500
6	6500
7	8750
8	8500
9	9250
10	9750

Table 5.4: Scalability of the shared dispatcher. 9750 indicates that all the job sets were dispatched.

In Figure (5.5), it is observed that the slope of the curve decreases for large job sets even as the number of processors are increased. The satellite processors need to access different locations of the array to update the safety intervals resulting in frequent page faults. As the size of the job set increases, the satellite processors need to access memory locations in the central memory for every constraint they relax. The number of read

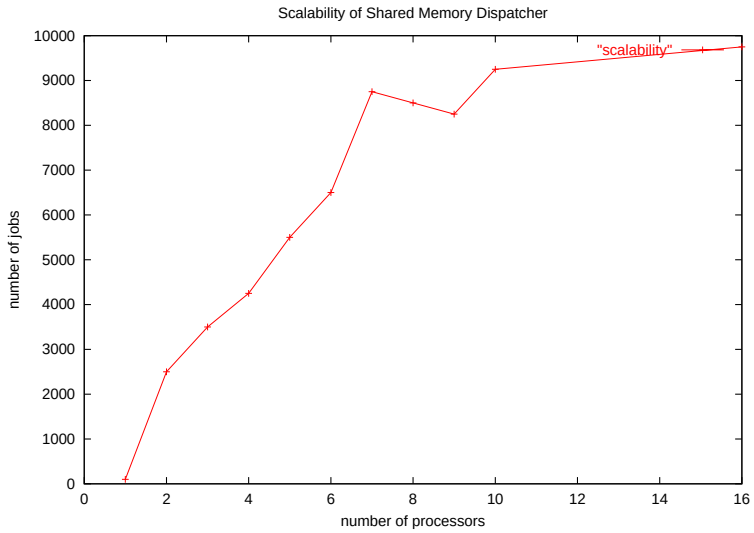

Figure 5.5: The number of jobs that can be successfully dispatched by a given number of processors, where the job execution time was between 1 to 5 milliseconds and the spacing time was between 0.1 to 0.5 milliseconds. The area under the curve shows the schedules which can be successfully dispatched.

and write operations to the main memory increase and slow down the updating process. After a certain number of jobs, the time to relax the constraints takes longer and causes the concerned satellite processor to be inapt in reading the start and execution time of the completed job. This shows that the memory to processor latency would become a bottleneck for updating constraints and further increasing the processors would not help as the latency and the spacing time are of the same order. Another reason for the delay in the latency is the increasing number of read/write requests from all the processors. There is limited bandwidth between the shared memory and the processors which would prevent all the processors from accessing and updating the memory at the same time. This shows the requirement of high speed connections and high memory bandwidth for the dispatchability of the schedules.

In the experiments, it was observed that a certain number of processors were able to dispatch job sets larger than the job sets for which they failed. These observations are due to the unpredictable memory flush time (time to achieve memory coherence) caused by the load on the machine. We also conducted experiments using 32 processors and it was observed that the dispatcher fails for some schedules dispatched by 7 processors. The update time for 32 processors as observed in Figure (5.4) is comparable to the spacing time of the test cases and here, the memory coherence time is a significant overhead.

5.4.2 Effect of execution time

An increase in the execution time of the jobs would give the satellite processors more time to complete updating the safety intervals. This would cause the satellite processors to wait for the central processor to update the flag f_1 which upholds our assumption in Section §3.3.2. A decrease in the execution time would make our assumption in Section §3.3.2 void as the satellite processors would be still updating safety intervals when the central processor completes a job. Hence larger the job execution time, larger is the job set that can be dispatched.

In the test cases created, the spacing time between two adjacent jobs was set between one-tenth to one-half of a millisecond and execution time was varied to be $[0.1ms, 0.5ms]$, $[0.5ms, 1ms]$, $[1ms, 5ms]$ and $[5ms, 10ms]$. The size of the job set was increased from 250 to 5000. Experiments were conducted by setting the number of processors to be used by the dispatcher and finding the largest job set up to which schedules were successfully dispatched. Each entry in Table 5.5 indicates the maximum number of jobs that were dispatched successfully by the number of processors, when the execution time of jobs is in a certain range. The sequential dispatcher was not able to dispatch any job set used in these experiments.

The observed results are listed in Table 5.5 and are in excellent co-ordination with the expected behavior. A table entry of 5000 implies that the dispatcher dispatched all the schedules created and can dispatch larger job sets.

Execution time	Processors											
(ms)	2	3	4	5	6	7	8	9	12	16	20	24
5-10	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000
1-5	2500	3500	4250	5000	5000	5000	5000	5000	5000	5000	5000	5000
0.5-1	750	1750	2250	2750	3250	3750	4000	4750	5000	5000	5000	5000
0.1-0.5	750	750	1000	1000	1500	1500	1750	1750	2750	3500	3500	4750

Table 5.5: Effect of varying the execution time on the dispatchability of a schedule, with a fixed spacing time $[0.1ms, 0.5ms]$

Figure (5.6) plots the largest job sets dispatched with a given number of processors for four different intervals of job execution time. Clearly, schedules with higher job execution time intervals got dispatched with lesser number of processors. From the Figure (5.6), it is concluded that greater the execution time, greater is the number of jobs that can be dispatched by the shared dispatcher.

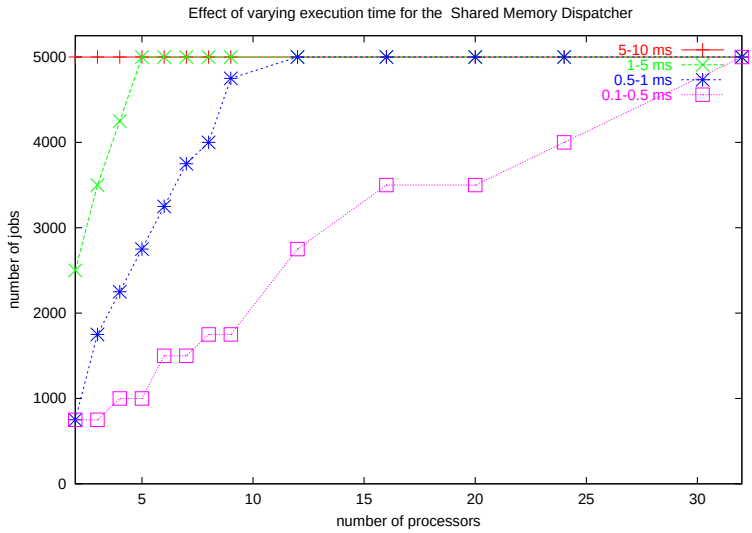

Figure 5.6: The plot shows the effect of varying the execution time of jobs on the dispatchability of a job set by a certain number of processors. The spacing time was assumed to be between 0.1 to 0.5 milliseconds.

5.4.3 Effect of Spacing time

On decreasing the spacing time, the allowed time between the finish time of a job and the start time of the next job is decreased. The satellite processors need to compute the safety interval of the next job within this time and hence the assumption in Section §3.3.2 becomes a necessity. The satellite processors need to complete updating all the safety intervals by the time the central processor completes executing the next job.

We generated Partially Clairvoyant schedules with spacing times of $[0.1ms,0.5ms]$, $[0.5ms,1ms]$ and $[1ms,5ms]$. For each spacing time, the schedules were generated with the number of jobs increasing from 250 to 5000 and execution times of $[0.1ms,0.5ms]$, $[0.5ms,1ms]$, $[1ms,5ms]$ and $[5ms,10ms]$. We fix the number of processors to be used by the dispatcher and empirically determine the largest job set up to which the dispatcher successfully dispatched.

Table 5.6 summarizes the results of the experiments conducted. Each entry in the table is the largest job set dispatched for the given number of processors with the concerned spacing and execution time. All job sets of smaller size with the same parameters of execution time, spacing time and number of processors were dispatched.

Figure (5.7) plots the largest job set dispatched for different values of spacing time versus the number of processors . Figure (5.7) shows that the size of the schedules dispatched increases with increasing the spacing

time. The effect of execution time of jobs with spacing time is also shown in Figure (5.7). The non-crossed lines show job sets with higher execution time of [0.5ms,1ms] while the crossed lines have execution time of [0.1ms,0.5ms]. The plot shows that job sets with larger execution times have greater dispatchability. Table 5.6 and Figure (5.7) shows that increasing the spacing time will allow job sets of larger size to be dispatched using the same number of processors and also that increasing the execution times of the jobs increases the number of jobs dispatched for a given value of the spacing time.

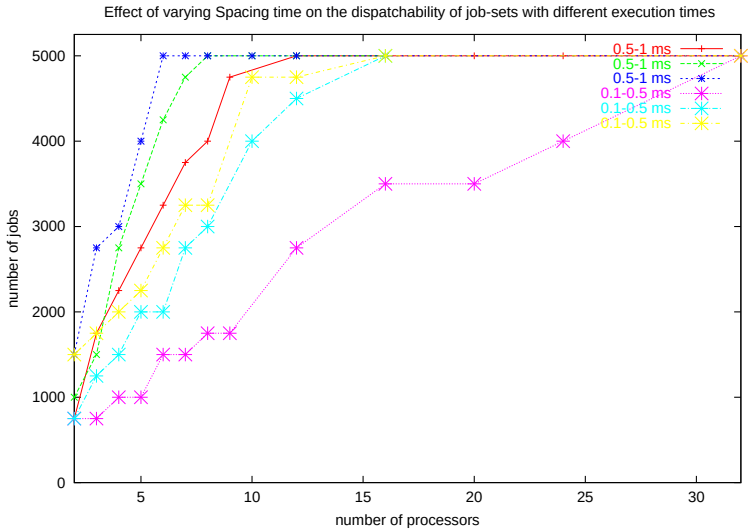

Figure 5.7: Effect of varying the spacing time on the dispatchability of job sets with different execution times.

5.5 Related Work

Some of the references stated in Section §4.6 are applicable to this architecture also, when a dispatcher is implemented based on the peer to peer model. An investigation was performed in [TZ02, TZ01] to determine how the performance and speedup of applications would be affected by using non-blocking rather than blocking synchronization in parallel systems. These papers also provides a set of efficient and simple translations that show how typical blocking operations found in parallel applications, such as simple locks, queues and lock trees can be translated into non-blocking equivalents that use hardware primitives common in modern multiprocessor systems. A non-blocking protocol was proposed in [TZ99], that allows real-time tasks to share data in a multiprocessor system. The protocol gives the means to the concurrent real-time tasks to read and write shared data and allows multiple write and multiple read operations to be executed concurrently.

Processors	Spacing time	Execution time			
		5-10 ms	1-5 ms	0.5-1 ms	0.1-0.5 ms
2	0.1-0.5	5000	2500	750	750
	0.5-1	5000	3000	1000	750
	1-5	5000	2000	1500	1500
3	0.1-0.5	5000	3500	1750	750
	0.5-1	5000	4000	1500	1250
	1-5	5000	4000	2750	1750
4	0.1-0.5	5000	4250	2250	1000
	0.5-1	5000	4000	2750	1500
	1-5	5000	5000	3000	2000
5	0.1-0.5	5000	5000	2750	1000
	0.5-1	5000	5000	3500	2000
	1-5	5000	4000	4000	2250
6	0.1-0.5	5000	5000	3250	1500
	0.5-1	5000	5000	4250	2000
	1-5	5000	5000	5000	2750
7	0.1-0.5	5000	5000	3750	1500
	0.5-1	5000	5000	4750	2750
	1-5	5000	5000	5000	3250
8	0.1-0.5	5000	5000	4000	1750
	0.5-1	5000	5000	5000	3000
	1-5	5000	5000	5000	3250
10	0.1-0.5	5000	5000	4750	1750
	0.5-1	5000	5000	5000	4000
	1-5	5000	5000	5000	4750
12	0.1-0.5	5000	5000	5000	2750
	0.5-1	5000	5000	5000	4500
	1-5	5000	5000	5000	4750
16	0.1-0.5	5000	5000	5000	3500
	0.5-1	5000	5000	5000	5000
	1-5	5000	5000	5000	5000

Table 5.6: Effect of varying the spacing time on the dispatchability of jobs for different execution times and different number of processors.

5.6 Conclusion

The shared memory dispatcher is shown to be superior to sequential dispatching, even though the communication time increases with number of processors. The shared dispatcher is shown to be scalable, i.e., larger schedules are dispatched with more processors. The effect of execution times and spacing times of jobs on the shared dispatcher is shown.

We observed "Loss of Dispatchability" in some experiments when the spacing time of jobs is less or when our assumption is violated. In case of larger schedules, the dispatcher failed due to the number of page faults that occur while updating constraints. For the schedules created, we showed that a schedule not dispatched by the distributed dispatcher cannot be dispatched by the sequential dispatcher.

The shared dispatcher will dispatch larger schedules when provided with a large memory that has low latency and high bandwidth.

Chapter 6

Conclusion

In this thesis, we proposed and empirically evaluated Partially Clairvoyant dispatchers on shared and distributed memory machines. The dispatchers were tested with schedules of different size and jobs of different execution times. On the whole, the approach is targeted to power up the controller by providing it with more processing power.

For the shared memory dispatcher, we demonstrate the scalability of the dispatcher showing that larger job sets can be dispatched by increasing the number of processors provided the machine has a high memory bandwidth. The shared dispatcher performs better when the computation time is greater than or comparable to the memory flush time(time to achieve memory coherence) of the processors. If the execution time of the jobs is small, the sequential dispatcher is not effected, while, the shared memory dispatcher would suffer as the available computing time in parallel decreases with respect to the communication time. The sequential dispatcher is a better choice for small job sets in such situations. In situations where the allowed update time between jobs is less, the shared memory dispatcher is better than the sequential dispatcher. In case the shared memory dispatcher cannot dispatch the schedule while updating constraints, the sequential algorithm would definitely fail.

In the distributed case, we proposed and tested dispatching a Partially Clairvoyant schedule on multiple processors. The results show the superiority of distributed dispatching over the sequential dispatching. It is shown that for every schedule, there would be a processor set which would dispatch the schedule successfully.

It was observed that a schedule dispatched by a smaller number of processors was not dispatchable with more processors. This is because each processor set has a different worst case scenario, depending on the assignment of jobs and the constraints. The experiments were conducted in a shared network where the

system load might have lead to noise in the observations made. However, this could not be verified due to lack of a dedicated machine.

It was observed in the distributed experiments that choosing a processor per node is better than choosing multiple processors per node. Choosing one processor per node increases the length of the connection paths between processors through switches but decreases the load on the ELAN layer. The ELAN layer that uses a shared memory to communicate data between processors in a node, becomes a bottleneck when all the processors in a node send data to each other. Thus, investigating the underlying system architecture and hardware while implementing the dispatcher helps in obtaining better performance.

In the future, the performance of the dispatcher on a dedicated machines is to be tested. The experiments revealed that page swapping while dispatching large job sets would lead to Loss of Dispatchability in clusters. Increased swapping decreases the throughput of the nodes which in turn would lead to the failure of the dispatcher. The above motivates three problems, namely, the problem of pruning the dispatch functions to remove redundant constraints between jobs, the problem of distributing jobs among processors and the problem of determining the number of processors such that the job set would be successfully dispatched. Other existing distributing paradigms of load balancing and work balancing can be applied and empirically tested.

Power aware processors change their clock speeds causing the execution time intervals to change. The effect of such changes on the schedulability and dispatchability of Partially Clairvoyant systems is to be studied.

A hybrid model constructed using start time intervals in [MLWP02] and variable execution times would be better at modeling the scenarios observed in existing real-time systems.

Bibliography

[AMMMA01] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *The 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, pages 95–105, Washington - Brussels - Tokyo, December 2001. IEEE.

[BBP+01] U. Brinkschulte, A. Bechina, F. Picioroaga, E. Schneider, T. Ungerer, J. Kreuzinger, and M. Pfeffer. A microkernel middleware architecture for distributed embedded real-time systems. In *The 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, pages 218–226, Washington - Brussels - Tokyo, October 2001. IEEE.

[CJD91] S. E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1991*, pages 74–83, San Antonio, Texas, USA, December 1991. IEEE Computer Society Press.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[DE73] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[DKT99] Subrata Das, Raffi Krikorian, and Walt Truszkowski. Distributed planning and scheduling for enhancing spacecraft autonomy. In *Proceedings of the third annual conference on Autonomous Agents*, pages 422–423. ACM Press, 1999.

[GPS95] R. Gerber, W. Pugh, and M. Saksena. Parametric Dispatching of Hard Real-Time Tasks. *IEEE Transactions on Computers*, 1995.

[HCF03] Karin Hgstedt, Larry Carter, and Jeanne Ferrante. On the parallel execution time of tiled loops. *IEEE Trans. on Parallel and Distributed Computing*, 14(3):307–321, March 2003.

[HFL95] D.L. Hull, W. Feng, and J.W.-S. Liu. Enhancing the performance and dependability of real-time systems. In *Proceedings of International Computer Performance and Dependability Symposium (IPDS'95)*, pages 174–182. IEEE Computer Society, april 1995.

[HJLL90] Huynh, Joskowicz, Lassez, and Lassez. Reasoning about linear constraints using parametric queries. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 10, 1990.

[HKR01] William Henderson, David Kendall, and Adrian Robson. Improving the accuracy of scheduling analysis applied to distributed systems computing minimal response times and reducing jitter. *Real-Time Systems*, 20(1):5–25, 2001.

[Ja'92] Joseph Ja'Ja'. An introduction to parallel algorithms (contents). *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 23, 1992.

[KS03] Seok-Kyu Kweon and Kang G. Shin. Statistical real-time communication over ethernet for manufacturing automation systems. *IEEE Trans. on Parallel and Distributed Computing*, 14(3):322–335, March 2003.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LW73] C. L. Liu and J. W. Wayland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20, January 1973.

[MFFR02] P. Marti, J.M. Fuertes, G. Fohler, and K. Ramamritham. Improving quality-of-control using flexible timing constraints: metric and scheduling issues. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 91–100. IEEE Computer Society Press, 2002.

[MLWP02] Aloysius K. Mok, Chan-Gun Lee, Honguk Woo, and Konana. P. The monitoring of timing constraints on time intervals. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 191–200. IEEE Computer Society Press, 2002.

[RGH+02] Paul E. Rybski, Maria Gini, Dean F. Hougen, Sascha A. Stoeter, and Nikolaos Papanikolopoulos. A distributed surveillance task using miniature robots. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 1393–1394. ACM Press, July 2002.

[RSE+00] Paul E. Rybski, Sascha A. Stoeter, Michael D. Erickson, Maria Gini, Dean F. Hougen, and Nikolaos P. Papanikolopoulos. A Team of Robotic Agents for Surveillance. In Carles Sierra, Maria Gini, and Jeffrey S. Rosenschein, editors, *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 9–16, Barcelona, Catalonia, Spain, June 2000. ACM Press.

[Sak94] Manas Saksena. *Parametric Scheduling in Hard Real-Time Systems.* PhD thesis, University of Maryland, College Park, June 1994.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[Sub00] K. Subramani. *Duality in the Parametric Polytope and its Applications to a Scheduling Problem.* PhD thesis, University of Maryland, College Park, July 2000.

[Sub02] K. Subramani. A specification framework for real-time scheduling. In W.I. Grosky and F. Plasil, editors, *Proceedings of the* 29^{th} *Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, volume 2540 of *Lecture Notes in Computer Science*, pages 195–207. Springer-Verlag, November 2002.

[Sub03] K. Subramani. An analysis of partially clairvoyant scheduling. *Journal of Mathematical Modelling and Algorithms*, 2003. (Accepted). Conference version available in Proceedings of the 8^{th} International Conference on High-Performance Computing (Hi-PC), Lecture Notes in Computer Science, volume 2228, pages 36-46, Springer-Verlag.

[Tr98] Martin Trngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14(3):219–250, 1998.

[TZ99] Philippas Tsigas and Yi Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 247. IEEE Computer Society, 1999.

[TZ01] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, Crete Island, Greece, July 3–6, 2001. SIGACT/SIGARCH and EATCS.

[TZ02] Philippas Tsigas and Yi Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proceedings of the 3rd International*

Workshop on Software and Performance (WOSP-02), pages 55–67, New York, July 24–26 2002. ACM Press.

[YYM01] Yang.S.X, Guangfeng Yuan, and Meng M. Real-time collision-free path planning and tracking control of a nonholonomic mobile robot using a biologically inspired approach. In *Proceedings of Computational Intelligence in Robotics and Automation*, pages 113–118. IEEE Computer Society, 2001.