INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Scheduling Real-Time Transactions in Parallel Database Systems

by

Sonia Takkar. B.Tech

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

August 1, 1997

© Copyright

1997. Sonia Takkar

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26975-2

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis.

Scheduling Real-Time Transactions in Parallel Database Systems

submitted by

Sonia Takkar. B.Tech

in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director. School of Computer Science

Thesis Supervisor

Carleton University
September 18. 1997

Abstract

Real-time database systems support transactions with timing constraints such as deadlines. Real-time transactions, in addition to preserving consistency of the database similar to a traditional transaction, have to meet their deadlines. Extensive research has been conducted in scheduling real-time transactions in centralized databases. However, real-time transaction scheduling in parallel database systems has not received much attention.

This thesis focuses on real-time transaction scheduling in shared-nothing parallel database systems. We propose a new priority-based scheduling policy. In this policy, all scheduling decisions are made locally by each node. In contrast, several other algorithms, proposed for distributed database systems, require communication among the nodes. Such communication can deteriorate the performance as real-time transactions have to complete before their deadlines. Also, in the other algorithms, all decisions are made globally. Thus, the entire transaction is blocked or aborted even if the conflict is at only one processor. This global blocking can lead to the transaction losing the already completed processing, thus deteriorating the performance of the system. This thesis provides a performance comparison of the proposed scheduling policy with two existing policies and shows that the proposed policy provides a superior performance.

Acknowledgements

I would like to extend my heart-felt gratitude to my supervisor, **Dr. Sivarama P. Dandamudi**, for his continuous help and support during the course of this thesis and also for always being there to help no matter how insignificant the problem.

I would like to dedicate this thesis to my family, who have always been a constant source of inspiration, especially my brother, **Alok** for bearing with me through all my erratic moods.

Notations

<i>db_size</i>	Size of the database
λ	Mean transaction arrival rate
<i>deadline_T</i>	Deadline to transaction <i>T</i>
<i>arrival_T</i>	Time of arrival of the transaction <i>T</i>
<i>slack_T</i>	Slack time of transaction <i>T</i>
<i>service_T</i>	Estimated execution time of transaction <i>T</i>
<i>criticality_T</i>	Criticality assigned to transaction <i>T</i>
<i>priority_T</i>	Priority assigned to transaction <i>T</i>
<i>min_slack_time</i>	Minimum value of slack time that can be allocated to transactions
<i>max_slack_time</i>	Maximum value of slack time that can be allocated to transactions
<i>slack_rate</i>	Factor used to determine the correlated slack time of transactions
<i>pages_accessed_T</i>	Actual number of pages accessed by transaction <i>T</i>
<i>lock_time</i>	Delay encountered by transaction to request lock on a page and locking it
<i>lock_grant_delay_T</i>	Delay in granting locks to transaction <i>T</i>
<i>cpu_time</i>	Delay at the CPU for transaction to process one page
<i>processing_delay</i>	Delay for processing the data of transaction <i>T</i>
<i>io_time</i>	Delay at the disk for transaction to access one page
<i>io_delay</i>	Delay for transaction <i>T</i> to access the disk to read required data

<i>write_pages_T</i>	Number of pages to be written by transaction <i>T</i>
<i>frac_write</i>	Fraction of total pages accessed by transaction <i>T</i> to be written
<i>compute_delay</i>	Delay involved to check for Rule 1 and 2 on one page
<i>communicate_delay</i>	Delay involved to broadcast final decision to all the processors
<i>conflict_delay</i>	Time necessary to check Rule 1 and 2
<i>cost_delay</i>	Time to globally block a transaction
<i>number_of_partitions</i>	Number of partitions into which each relation in the database will be divided into
<i>frac_partition</i>	Degree of declustering of each relation
<i>no_of_processors</i>	Total number of processors in the system
σ	Degree of data skew in the system

Contents

Abstract	iii
Acknowledgements	iv
Notations	v
List Of Tables	x
List Of Figures	xi
1 Introduction	1
1.1 Distributed and Parallel Database Systems	1
1.2 Real-Time Database Systems	4
1.3 Architectures of Parallel Database Systems	6
1.4 Contributions of the Thesis	11
1.5 Overview of the Thesis	12
2 Related Work	13
2.1 Scheduling in Centralized Databases	13
2.2 Scheduling in Distributed Databases	24

3	An Overview of Implemented Scheduling Policies	30
3.1	Synchronized Scheduling Policy	30
3.2	Modified Synchronized Scheduling Policy	32
3.3	Adaptive Highest Priority First Policy	34
4	Simulation Model	39
4.1	Workload Model	39
4.2	System Model	43
4.3	Database Model	48
5	Performance Evaluation and Analysis	52
5.1	Sensitivity to Inter-arrival Time	54
5.2	Sensitivity to Data Skew	56
5.3	Sensitivity to Write Percent	57
5.4	Sensitivity to Slack Time	58
5.5	Sensitivity to Declustering	59
5.6	Sensitivity to Transaction Size	61
5.7	Sensitivity to Number of Processors	62
5.8	Sensitivity to Block Delay	63
5.9	Earliest Deadline First vs Least Slack First	64
5.10	Correlated vs Uncorrelated Slack Time	65
6	Conclusions	67
6.1	Summary	67
6.2	Contributions of the Thesis	68
6.3	Future Work	69

A	Results of Earliest Deadline First with Correlated Slack Time	71
A.1	Sensitivity to Inter-Arrival Time	71
A.2	Sensitivity to Data Skew	72
A.3	Sensitivity to Write Percent	73
A.4	Sensitivity to Declustering	73
A.5	Sensitivity to Transaction Size	75
B	Results with Earliest Slack Time	76
B.1	Effect of Uncorrelated Slack Time	77
B.1.1	Sensitivity to Inter-Arrival Time	77
B.1.2	Sensitivity to Data Skew	77
B.1.3	Sensitivity to Write Percent	78
B.1.4	Sensitivity to Declustering	79
B.1.5	Sensitivity to Transaction Size	80
B.1.6	Sensitivity to Slack Time	81
B.1.7	Sensitivity to Number of Processors	82
B.2	Effect of Correlated Slack Time	83
B.2.1	Sensitivity to Inter-Arrival Time	83
B.2.2	Sensitivity to Data Skew	83
B.2.3	Sensitivity to Write Percent	85
B.2.4	Sensitivity to Declustering	86
B.2.5	Sensitivity to Transaction Size	86
	References	88

List of Tables

5.1	Some simulation parameters and their default values	53
-----	---	----

List of Figures

1.1	Some parallel database system architectures	7
4.1	High-level block diagram of the system model	45
5.1	Sensitivity to inter-arrival time	55
5.2	Sensitivity to data skew	56
5.3	Sensitivity to write percent	57
5.4	Sensitivity to slack time	59
5.5	Sensitivity to declustering	60
5.6	Sensitivity to size of the transaction	61
5.7	Sensitivity to inter-arrival time (with 5 processors)	62
5.8	Sensitivity of the synchronized policies to block delay	63
5.9	Earliest Deadline First vs Least Slack First	65
5.10	Correlated (C) Slack Time vs Uncorrelated (UC) Slack Time	66
A.1	Sensitivity to inter-arrival time	72
A.2	Sensitivity to data skew	73
A.3	Sensitivity to write percent	74
A.4	Sensitivity to declustering	74
A.5	Sensitivity to transaction size	75
B.1	Sensitivity to inter-arrival time	77
B.2	Sensitivity to data skew	78

B.3	Sensitivity to write percent	79
B.4	Sensitivity to declustering	80
B.5	Sensitivity to transaction size	81
B.6	Sensitivity to slack time (uncorrelated)	82
B.7	Sensitivity to number of processors	83
B.8	Sensitivity to inter-arrival time	84
B.9	Sensitivity to data skew	84
B.10	Sensitivity to write percent	85
B.11	Sensitivity to declustering	86
B.12	Sensitivity to transaction size	87

Chapter 1

Introduction

Databases are playing a pivotal role in the success of organizations and the world in general. A database can be defined, in simple terms, as a collection of logically interrelated data. Most of the information needs of industries and individual people are being supported by databases. As the amount of information and data to be managed increases, so does the importance of databases. Today, databases occupy a central position in commercial information systems.

1.1 Distributed and Parallel Database Systems

An increase in the amount of data to be stored and the need to share data among different users, separated by large physical distances led to the emergence of *distributed databases*. A distributed database can be defined as multiple databases located at different sites, interconnected by a computer network. These types of databases are able to store large amounts of data at the different sites and do not exert too much load on a single processor, as was the case in centralized databases. The distribution of data across different processors is not visible to the users. This concept is called *transparency*. Transparency extends the concept of data independence, which plays an important role in distributed databases. The users are provided with a

single, logical image of the entire database even though the database is physically distributed, enabling the user to visualize it as a centralized database. Transparency can be provided in a number of different forms like network transparency, replication transparency and fragmentation transparency.

As in centralized databases, data accesses by multiple users have to be synchronized to maintain consistency of the distributed database. Consistency can be maintained by enforcing a correctness criterion called serializability, which takes the *isolation* property of transactions into consideration [23]. The isolation property states that until a transaction completes execution, the effects of its execution are not seen by other transactions. The task of maintaining serializability in a distributed database is a challenge as transactions are executing at multiple sites and a global serializability has to be maintained. Thus, the execution of a group of distributed transactions is serializable if and only if [24]:

- The execution of the group of transactions is serializable at each site.
- The serialization order of these transactions is identical at each site.

The introduction of parallel, multiprocessor systems opened up a whole new field in the area of databases. A parallel system consists of a number of processors executing jobs in parallel. Most of the multiprocessor systems available today can be broadly categorized into shared memory and distributed memory systems. In shared memory systems, the different processors share the memory available in the system while in distributed memory systems, each processor has its own local, private memory. The shared memory system supports a simple programming model where data sharing is carried out by reading and writing shared data structures in the common memory. In distributed memory systems, the different processors cooperate with each other and communicate through message passing protocols.

In a multiprocessor system, jobs are executed on different processors in parallel. A parallel job is essentially a collection of many computational tasks [22], possibly with some precedence constraints, where a task is identified as a computational fragment that must be sequentially executed with no part of it executable in parallel.

The basic aim of executing jobs in parallel is either to minimize the execution time of each job or to achieve a balanced workload on each processor. Scheduling involves allocation and assignment of processors to different jobs using a particular scheduling algorithm. Different scheduling algorithms can be used to allocate the processors to the different jobs depending on the application for which this system is being used.

This combination of parallel processing and database management gives rise to the concept of *parallel databases*. Parallel databases exploit the parallelism available in multiprocessor systems to bring forth high-performance databases with high data availability. Parallel database systems use a message-based client/server operating system to connect the parallel processors interconnected by a high-speed network [9]. Like distributed databases, parallel databases have been primarily designed to handle large amounts of data. In the original single processor database systems, an increase in the amount of data to be checked increases the response time of the system significantly. With parallel databases, the response time of the system can be kept under control. Another advantage of parallel databases is the use of conventional processors, disks and memories to construct these systems. The components are readily available and inexpensive. The technology available today is more than satisfactory for the construction of fast and inexpensive parallel database systems. The modular architectures used nowadays also enable the systems to grow linearly by adding memory, disks and so on. In addition to the hardware advantage, these systems can also take advantage of the parallelism inherent in the software.

Three types of parallelism is inherent in the flow of user accesses to the system, called *transactions* [23]. *Inter-query parallelism* enables the parallel execution of queries generated by concurrent transactions. *Intra-query parallelism* enables the parallel execution of concurrent, independent operations within the same query. The third type of parallelism, called the *intra-operator parallelism* enables the parallel execution of many sub-operations within a single operation, that can be executed independently. Inter-query and intra-query form of parallelism can be obtained using data partitioning and intra-operation parallelism can be obtained using function partitioning in addition to data partitioning.

To summarize, distributed databases assume a loose interconnection among the different processors, which have their own operating systems and operate independently. On the other hand, parallel databases are implemented on tightly coupled multiprocessor systems and take full advantage of the inherent parallelism present in such systems. As parallel databases are so operationally similar to distributed databases, most of the concepts of distributed databases can be applied to them.

1.2 Real-Time Database Systems

In quite a few of the commercial applications of databases today like nuclear power plants, traffic control systems, stock market, banks and so on, the user accesses submitted to the system, also called transactions, have to be completed within a certain amount of time or they are rendered useless. To satisfy these timing constraints, the concept of real-time transaction processing systems was introduced where the transactions are submitted to the system with a predetermined deadline. The transactions have to be completed within the deadline specified. The databases which consider the timing constraints of transactions are called *real-time databases*. In addition to taking the deadline of the transaction into consideration, real-time databases, like conventional databases, have to maintain the consistency of the database.

A real-time transaction can be characterized by four factors [16]: a deadline, a criticalness, a priority and a size that is dependent on the number of data objects to be accessed and the amount of computation to be performed. Criticalness of a transaction can be determined by the degree of importance of the transaction. Criticalness also determines the severity of the consequences if the transaction does not complete within the given deadline. Priority can be a function of various properties of individual transactions such as deadline, time of arrival and slack time allocated to a transaction. It is essentially used to maintain serializability of the various transactions accessing the database.

Transactions in real-time databases can be categorized into hard transactions and soft transactions. Hard transactions are unyielding about satisfying their timing

constraints. The failure to satisfy the timing constraints can lead to catastrophic results. Soft transactions are flexible about their timing constraints. A small delay in the completion of these transactions can be tolerated.

Scheduling transactions in databases with real-time constraints using their priorities and criticality is a tricky task. The scheduling algorithms used must be able to provide the concurrency control of conventional databases to maintain the consistency of the database while at the same time satisfy the timing constraints of the transactions. One of the many challenges of databases using real-time transactions is the introduction of a theory for real-time scheduling and concurrency control protocols that maximizes both concurrency and resource utilization, subject to three constraints: data consistency, transaction correctness and transaction deadlines [30].

This purpose of this study is to incorporate the concept of real-time transactions in parallel database systems. This can lead to the use of parallel databases in applications where the timing constraints of transactions have to be respected. In single processor systems, the increase in the number of incoming transactions increases the possibility of a number of transactions missing their deadlines. If parallel systems can be used to cater to real-time transactions, the increase in the number of incoming transactions will not lead to an increase in the probability of transactions missing their deadlines, as resources can be incrementally augmented to satisfy the increased demand.

1.3 Architectures of Parallel Database Systems

The design of parallel databases can be categorized into three major categories [9].

- **Shared memory** : In this type of system, all the processors share a common memory and have a direct access to all disks. All processors are connected to the shared memory through an interconnection network. The commercially available systems of this type are the IBM/370, Digital VAX and the Sequent Symmetry.
- **Shared disk** : In this type, each processor has an individual memory but have direct access to all the disks. All processors are interconnected to all the disks through an interconnection network. The commercially available systems of these type are the IBM Sysplex and the original Digital VAXcluster.
- **Shared nothing** : In this type, all processors have individual memories and disks. All the processors are connected via a high speed interconnection network. Communication between the processors is using a message-passing model with send/receive primitives. This design of parallel databases has been the most popular and has led to the emergence of many commercial parallel machines like the system from Teradata and Tandem and research prototypes like Bubba and Gamma.

Figure 1.1 shows the basic models of these three architectures. In addition to the three architectures described, a fourth architecture has recently been proposed. This type is called the Shared Intermediate Memory architecture [34]. The design of this architecture is the same as the shared-disk type. In addition to the basic hardware, a shared intermediate memory has been introduced which serves as a global shared buffer for all nodes. This model shows good response time for all transactions.

Each architecture has its own advantages and disadvantages. The shared-memory design has a common shared memory which is connected to all the processors via an interconnection network. This system has a simple model which basically involves the reading and writing of shared data stored in the common, shared memory. But

Figure 1.1: Some parallel database system architectures

this sharing of memory increases the risk of interference between the different processors and increases latency when accessing the shared memory. The delay in accessing the memory deteriorates the performance of the system and limits their scalability. The shared-disk design shares the disks, which store the data, but the risk of interference between the processors comes up here too. This can result in the delay when accessing and create heavy load on the interconnection network. The shared-nothing architecture does not allow sharing of resources resulting in no interference among the different processors. The shared-nothing architecture also removes the need for a complex interconnection network. As none of the resources are shared, only the refined queries of the user have to be transmitted across the network. In the shared-memory and shared-disk architectures, due to the sharing of memory and disks respectively, the interconnection network used has to be very powerful as it has to transmit large quantities of raw data across to the different processors. This raw data is processed at the respective processors and then transmitted to the output client programs.

The shared-nothing architecture can also be scaled extensively as none of the components are shared. In the shared-memory and shared-disk architectures, scaling is limited as an increase in the number of processors also puts more pressure on the interconnection network, which has to be made more powerful to handle more data. Also, as the number of processors increase, interference among the processors in the shared-memory and shared-disk architectures also increases. With all its advantages, shared-nothing architecture also has its own disadvantages. The presence of individual memory and disk increases the complexity of the system and can lead to potential load-balancing problems. Furthermore, a sudden surge of load on the system causes a bottleneck at the backup server node and lowers the maximum throughput of the system.

Due to its various advantages and minimal disadvantages, the shared-nothing architecture has emerged as the most commonly used architectures in all the commercially available products. The availability of cheap and off-the-shelf components have led to a sharp increase in the development and acceptance of shared-nothing

parallel databases.

In shared-nothing parallel databases, data to be stored has to be partitioned across many disks. Almost all the commercial databases available use the relational data model. The relational data model, which was first proposed by Codd [8], represents the database as a collection of relations. Each relation can be thought of as a table, where each row in the table represents a collection of related data values. The name of the columns of the table represents the manner in which the data in each row will be interpreted, depending on the column under which it is stored. In relational model terminology, a row is called a tuple, a column header is called an attribute and the table is called a relation. A query can be defined as the command given by the user to access the database and retrieve the results.

To store large relations, the relations can be fragmented into a number of partitions and can be stored on the different disks. Partitioning or fragmentation should ensure that the database does not undergo any changes during fragmentation. If a relation has been fragmented, it should be fragmented without any loss of data, all the fragments should be disjoint and it should be possible to regain the original relation by joining all the fragments. Fragmentation can be of two types: horizontal and vertical. *Horizontal* fragmentation involves the fragmentation of a relation using predicates defined on the relation. *Vertical* fragmentation involves partitioning the attributes of a relation. Horizontal fragmentation works very well when most of the queries involve associative access, i.e., tuples with a common value of an attribute or a set of attributes have to be accessed. Vertical fragmentation is used to partition relation in such a way that many of the user applications can run on only one fragment. In distributed systems, horizontal fragmentation is more widely used while vertical fragmentation is used mostly in centralized systems.

Partitioning allows the processors to scan and write data from all the disks in parallel, thus minimizing the execution time of the transaction. The main advantage of partitioning is that each partition of data can be read and written using normal I/O hardware. Additional specialized hardware is not required. The use of commonly available I/O hardware reduces the cost and also the complexity of the

system. The system can also be scaled indefinitely without any requirement of complex and customized hardware. According to a study conducted [21], partitioning or declustering outperforms clustering i.e., placing the whole database on a single disk, because of the use of parallelism and the spreading out of work across all disks in the system in a uniform manner. Furthermore, at low utilization, performance improvement when using declustering is due to parallelism while in systems with medium and high utilization, improvement is due to reduction on the coefficient variation of the service demand time and the spreading out of work among all the disks respectively. Partitioning of data can be done by using many different strategies. The most important partitioning strategies are round-robin, hash partitioning and range partitioning methods [9].

Round robin is the simplest partitioning strategy to implement. In this method, the tuples are distributed among the various fragments of the relation in a round-robin fashion i.e., tuples are distributed among the different fragments sequentially. The round-robin strategy works well when majority of the queries involve the whole relation to be accessed sequentially (sequential access). But this strategy creates a problem when in majority of the queries, tuples have a common value for a particular attribute or a set of attributes have to be accessed. (associative access).

The *hash partitioning* strategy, the tuples are distributed among the different fragments using some hash function mapped on an attribute or a set of attributes of the relation. This strategy is suited for sequential as well as associative access as the tuples will be placed according to a specific value of an attribute or attributes of the relation. The advantage of this strategy is, for associative access, scan will be directed to only one disk instead of all them, thus limiting the overhead of starting queries at multiple disks.

Finally, in the *range partitioning* strategy, tuples with similar attribute values, which are frequently accessed together, are placed together. This technique of storing frequently accessed tuples together is also called clustering. The range partitioning strategy is also suited for sequential as well as associative access. But clustering can lead to a few problems during data access. It can be that as all the data to be

accessed in a particular application is similar and related. all the data will be stored in the same fragment. This problem is called *data skew*. Another problem can be that if a group of tuples are accessed very frequently. this can lead to all execution being limited to that particular fragment. This problem is called *execution skew*. As hash partitioning and round-robin strategies do not cluster data. they are not sensitive to the problems of data skew. Partitioning also leads to an improvement in the response time of the system. In case of sequential access. increase in the number of partitions. brought about by increase in the number of processors used to access the disks. leads to a decrease in the response time of the system. In the case of associative access. as the number of partitions increase. the number of tuples to be accessed at each partition decrease.

1.4 Contributions of the Thesis

Most of the existing work in the area of real-time transaction scheduling has been centered around centralized databases. We will study the behaviour of real-time transactions in parallel databases. We propose a new priority-based policy for scheduling real-time transactions in parallel database systems. This policy depends on the current priority of the transaction and involves minimal communication among the various processors. Continuous need for communication among the various processors is a very important factor in the deterioration of performance in parallel systems and more importantly in real-time systems. where deadlines of transactions have to be met. Also. the proposed policy is purely local in nature and all decisions are made locally. This is in contrast to the existing architectures which carry out all the decisions globally. We will consider the shared-nothing parallel database architecture with hard real-time transactions.

The priority assignment policy considered uses either the deadline or the slack time of the transaction and the relative performance of the new policy to the two existing policies is superior in both cases. The slack time is considered to be either correlated or uncorrelated to the execution time of the transaction. The results

presented in this thesis show that the proposed new policy provides superior performance than the existing policies under various load conditions and remains highly robust and stable under varying degrees of system loads. Also, the results considering the deadline of the transaction as the priority were better than with slack time of the transaction as priority. This can be attributed to the fact that deadline includes the time of arrival, the slack time and the service time of the transaction.

1.5 Overview of the Thesis

Chapter 1 gives an introduction to parallel databases and the different architectures into which they can be categorised. This chapter also gives an introduction to real-time database systems. Chapter 2 gives a brief overview of the real-time scheduling algorithms proposed in the literature for centralized and distributed database systems. This chapter also presents the major differences between the design of algorithms for centralized and distributed database systems. The two policies used to compare the performance of the new policy are discussed in Chapter 3. Our new algorithm, Adaptive Highest Priority First policy is also presented in this chapter and the advantages of this policy over the existing ones are discussed. Chapter 4 gives details on the simulation model used in the experiments. Chapter 5 compares the performance of the proposed policy with the two existing policies. The conclusions of this study and topics for future research are given in the last chapter.

Chapter 2

Related Work

2.1 Scheduling in Centralized Databases

Most of the existing real-time transaction scheduling algorithms have been designed for centralized databases. These algorithms can be broadly categorized into locking and timestamp ordering protocols.

In general, all locking protocols are based on the two-phase locking protocol. The first phase, called the *growing phase*, involves obtaining the locks on the data that are required by the transaction. The locks can be granted before the transaction begins execution (static locking) or on demand during execution (dynamic locking). At the end of the growing phase, the transaction enters the second phase, called the *shrinking phase*, in which all the locks held by the transaction are released [7]. The existing policies for real-time databases schedule the transactions according to the priorities assigned to them. These priorities are a function of either the time of arrival of the transactions, their deadlines or their slack time. In order to improve performance of the basic two-phase locking protocol, other policies, in addition to using the simple two-phase locking protocol, check if certain special conditions are also satisfied. These conditions are used to reduce the number of transactions that are restarted in the system and also to provide serializability of transactions, leading to less transactions missing their deadlines.

It is assumed that the priority of the data item is defined as the maximum priority

of all the transactions that hold a lock on the data item [32]. If the data item is not locked by any transaction, its priority is undefined. The locks on the data item are assumed to be similar to the locks in strict two-phase locking protocol. The locks are categorized into read and write locks. Read locks are also called shared locks as more than one transaction can hold a read lock on the same data item. Write locks are also called exclusive locks as only one transaction can hold a write lock on a data item at a particular time. A shared lock request conflicts with an exclusive lock and an exclusive lock request conflicts with both shared and exclusive locks. If a transaction requests a shared lock on a data item, which has already been shared locked by other transactions, it will be granted the request if it has higher priority than all the transactions that are waiting to lock that data item in the exclusive mode. Some of the policies most commonly used are described next.

1. Always Block Protocol [32]

This protocol is similar to the *Wait* protocol discussed in Abbott & Garcia-Molina [1]. This protocol is based on the strict two-phase locking protocol. In this protocol, transactions are scheduled at the CPU using their priorities but to schedule lock requests, the transaction priorities are not taken into account. This means that if a transaction is requesting a lock on a data item, which has already been locked by another transaction, the requesting transaction will always be blocked.

if T_R conflicts with T_H
then T_R blocks

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

2. Priority Inheritance Protocol [32]

This protocol is similar to the *Wait Promote* protocol discussed in Abbott & Garcia-Molina [1]. In this protocol, transactions are assigned priorities and in both CPU scheduling and scheduling lock requests, the priorities are taken into account. If a transaction requests a lock on a data item, which has already been locked by another transaction, the requesting transaction blocks. If the transaction holding the lock has lower priority than the requesting transactions, the holding transaction T inherits the priority of the transaction that has the highest priority among all the transactions that have been blocked by the transaction T . But, this leads to two transactions having the same priority and may eventually result in a deadlock.

if $P(T_R) > P(T_H)$
 then T_R blocks
 T_H inherits the priority of T_R
 else T_R blocks

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

$P(T_R)$ is the priority of the transaction requesting the lock

$P(T_H)$ is the priority of the transaction holding the lock

3. Priority-based Locking Protocol [32]

In this protocol, all transactions are assigned priority and in case of a conflict, the lower priority transaction will always be blocked. In the case of conflict during requesting locks, the higher priority transaction will always block the lower priority transaction. This protocol has an advantage in systems where real-time priorities of transactions are static and no two transactions have the same priority. In such systems, this protocol does not allow deadlocks to occur, thus saving time and overhead to detect and resolve deadlocks.

4. Priority Ceiling Protocol [32]

The priority ceiling protocol is an improvement over the priority inheritance protocol. It solves the deadlock problem of the priority inheritance protocol. The priority ceiling of a data item is the priority of the highest priority transaction that may have a lock on that data item, i.e., the transaction may already have a lock on the data item or will do so in future. To obtain a lock on a data item, priority of the requesting transaction T should be higher than the highest priority ceilings of data items locked by transactions other than transaction T . This protocol checks all the data items locked by other transactions even if there is no conflict among them.

5. Data Priority-based Locking Protocol [32]

This protocol makes an assumption that each arriving transaction has to submit a list of all the data items it will access called the access list. In this protocol, each data item has a priority equal to the highest priority of all transactions in the system that have the data item in their respective access lists. To find out its priority, each data item maintains a transaction list. The list contains the id and priority of the transactions in the system that have the data item on their access lists. This list is sorted by priority and determines the priority of the data item. The transaction list is modified when a transaction enters or leaves the system. To access a data item, the requesting transaction should have a priority equal to the priority of the data item. If the priority of the requesting transaction is less than the priority of the data item, the transaction is blocked. It is also assumed here that each transaction has a unique priority thus eliminating deadlock from this protocol.

```
if priority(T) = priority(D)
then if D was locked by a transaction  $T_X$ 
    then  $T_X$  is aborted
        Lock on D is granted to T
else
    T is blocked by the transaction that determines
    the current priority of D
```

where T is the transaction requesting the lock

D is the data item that has to be accessed by T

T_X is the transaction that has currently locked

6. Basic Timestamp Ordering Protocol [32]

In this protocol, each arriving transaction is assigned a timestamp. The timestamp of a transaction can be a function of different properties associated with a transaction but is usually determined by the time of arrival of the transaction into the system. The value of the timestamp determines the ordering of the transaction in case of a conflict. A read request by a transaction is completed if the data item to be read has not been updated by a transaction with a larger timestamp. Also, a write request by a transaction is completed if the data item to be written has not been read or written by a transaction with a larger timestamp. If the read/write request of the transaction cannot be completed, the transaction is aborted and restarted with a new timestamp, which will definitely be larger than the previous one.

7. Priority-Based Timestamp Ordering Protocol [32]

The basic timestamp ordering protocol does not take the real-time constraints into consideration. A lower priority transaction with a larger timestamp can always abort a higher priority transaction but with a lower timestamp. This leads to the problem of *priority inversion* in timestamp ordering protocols. To

avoid this problem. transactions arriving at different times are placed into different timestamp groups. Each timestamp group spans over a fixed interval of time. The time interval is decided based on the rate of arrival of the transactions. All transactions that arrive within a particular time interval are placed in one timestamp group. Each of the transactions in a timestamp group are scheduled based on their individual real-time priorities. Thus, each transaction is assigned a two-level timestamp made up of the group timestamp and the real-time timestamp. The group timestamp is the arrival time of the first transaction in that group. Real-time timestamp is based on the real-time priorities of the individual transactions. Also, each data item is associated with the group and the real-time timestamp of the most recent transaction that has accessed the data item. When a transaction attempts to access data item, its group timestamp is compared with that of the data item. If both the group timestamps are equal, the real-time timestamps are compared. If this attempt by the transaction to access the data item is unsuccessful, the transaction is aborted and restarted with new group and real-time timestamps.

The use of this protocol is also restricted as it gives more importance to the group timestamp, which is based on the arrival times of the transactions, than to the real-time priorities of the transactions. Consequently, a transaction arriving late but with a high priority will be aborted by a transaction which arrived earlier but has a low priority.

```
if (group_timestamp(T) > group_timestamp(D))
then T accesses D
    group_timestamp(D) = group_timestamp(T)
    real-time_timestamp(D) = real-time_timestamp(T)
else
    if (group_timestamp(T) = group_timestamp(D) and
        real-time_timestamp(T) > real-time_timestamp(D))
    then T accesses D
        real-time_timestamp(D) = real-time_timestamp(T)
    else
        T is aborted
```

where T is the transaction requesting the lock

D is the data item that has to be accessed by T

3. Optimistic Protocol [32]

This protocol allows all transactions to proceed without being blocked or aborted until they reach the commit stage. The transaction is committed only if it does not have any conflict with any of the transactions that have committed since it started execution. If the transaction fails the validation, it is aborted and restarted. This version is also called the *classical protocol*.

Another version of this protocol, called the *broadcast commit protocol*, during the validation phase of a committing transaction, performs the validation check against the other active transactions in the system and the transactions that are conflicting with the committing transaction are aborted. This protocol uses the rule that if half or more of the transactions conflicting with the committing transaction are of higher priority, the transaction is blocked and the higher priority transactions are completed. If not, the transaction is allowed to commit and the conflicting transactions are aborted.

9. Priority Dependent Locking Protocol [30]

This protocol has been devised for a centralized real-time database system with a single processor and where the transactions arrive randomly. In this system, when each transaction arrives, it is assigned an initial priority and a start time-stamp. The initial priority is based on the deadline and the criticality of the transaction. The start time-stamp can be the time of arrival of the transaction and is considered to be unique. The start time-stamp is appended to the initial priority to form the actual priority. If a transaction has been aborted, it is assigned a new priority when it is restarted. The execution of each transaction has been divided into three phases: *read phase*, *wait phase* and *write phase*.

In the read phase, the transaction reads the database and prewrites into its local buffer. In the wait phase, the transaction waits to commit. If it commits, the transaction switches into the write phase and makes all the updates permanent in the database. During the read phase, if a transaction has to access a data item, it has to obtain a lock on that data item. A transaction obtains a read lock on a data item if it wants to perform a read function on the data item. Similarly, if a transaction wants to perform a prewrite function, it should obtain a write lock on that data item. Each lock contains the priority of the transaction holding the lock and the type of lock. In the priority dependent locking protocol, the original concept of shared and exclusive locks is not very strict. In this protocol, a high priority transaction does not block for a lock, if the lock is being held by a lower priority transaction.

To obtain a lock on a data item, a few conditions have to be considered. Each transaction is associated with two sets called *before-trans-set* and the *after-trans-set*. The *before-trans-set* contains all the active lower priority transactions that must precede the transaction in the serialization order. The *after-trans-set* contains all the active lower priority transactions that will come after the transaction in the serialization order. Also associated with each transaction is a count called *before-count*. *Before-count* is the number of

higher priority transactions that precede the transaction in the serialization order. When the transactions arrive initially, both the *before-trans-set* and the *after-trans-set* are empty and the *before-count* is set to 0. At every conflict between two transactions, the dependency of the two transactions is determined and the values of the two sets and the count are changed accordingly.

In this protocol, the transaction can commit only after all the higher priority transactions that must precede the transaction in the serialization order, denoted by the value of *before-count*, are either committed or aborted, i.e., a transaction can commit only when the value of *before-count* becomes zero. If the transaction is ready to commit, it enters the write phase and releases all its locks. During this stage, the transaction is allotted a final timestamp. The final updates to the database are scheduled by the data manager, which uses the final timestamps to schedule the different transactions. Thomas Write Rule (TWR) [2] is used in this protocol to avoid conflicts between the write requests by different transactions.

10. High Priority [1]

In both the *Always Block* and *Priority Inheritance* protocol, the requesting transaction always had to wait for the transaction holding the lock on the data item. In this protocol, in case of a lock conflict, the priorities of the requesting and the holding transactions are compared. If the priority of the requesting transaction is higher than the priority of the data item, the priorities of all the transactions holding a lock on that data item will have a lower priority than the requesting transaction. In such a case, all the transactions holding a lock on the data item will release their locks, roll-back and restart. The requesting transaction is granted a lock on the data item and continues its execution. If the priority of the requesting transaction is less than or equal to the priority of the data item, the requesting transaction blocks and waits for the data item to become free.

if for all T_H holding a lock on a data item O
 $P(T_R) > P(T_H)$
 then Abort each lock holder
 else T_R blocks

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

$P(T_R)$ is the priority of the transaction requesting the lock

$P(T_H)$ is the priority of the transaction holding the lock

This protocol is not useful in cases where any of the transactions holding a lock on the data item is in the wait phase, i.e., waiting to commit. If the transaction is rolled back and restarted, it may miss its deadline. Also, after the transaction is restarted, it will get a new priority which may be higher than the priority of the transaction, which just aborted it. The previously requesting transaction can then be aborted leading to cyclic aborts. To avoid this, an extension of this protocol was suggested.

When a transaction requests a lock on a data item, the priority of the requesting transaction is compared to the priority of the holding transactions and the new priority assigned to the holding transactions if they were aborted and restarted at that instance of time.

if for all T_H holding a lock on a data item O
 $P(T_R) > P(T_H)$ and $P(T_R) > P(T_H)^A$
 then Abort each lock holder
 else T_R blocks

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

$P(T_R)$ is the priority of the transaction requesting the lock

$P(T_H)$ is the priority of the transaction holding the lock

$P(T_H)^A$ is the new priority of the transaction holding the lock
if it were aborted and restarted.

11. Conditional Restart [1]

This protocol introduces a further modification of the *high priority* protocol. In the revised high priority protocol, if the priority of the requesting transaction was greater than the priority of the holding transaction and the new priority of the holding transaction after it is aborted, the holding transaction is rolled back and restarted. But, if the holding transaction has already completed a major part of its execution, it may lose the already completed execution time if it is aborted. To prevent this, the conditional restart protocol takes the slack time of the requesting transaction into account. The *slack time* of a transaction is defined as the amount of time the transaction can afford to wait without missing its deadline. If the slack time of the requesting transaction, denoted by S_R , is greater than or equal to the remaining execution time of the holding transaction, the requesting transaction is blocked and the holding transaction continues its execution till it completes. If the slack time is less than the remaining execution time, the holding transaction is aborted and restarted after a predefined amount of delay.

if $P(T_R) > P(T_H)$ and $P(T_R) > P(T_H)^A$

then if $S_R \geq E_H - P_H$

then T_R blocks

T_H inherits the priority of T_R

else abort T_H

else T_R blocks

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

$P(T_R)$ is the priority of the transaction requesting the lock

$P(T_H)$ is the priority of the transaction holding the lock

$P(T_H)^A$ is the new priority of the transaction holding the lock
if it were aborted and restarted.

S_R is the slack time of the transaction requesting the lock

$E_H - P_H$ is the remaining execution time of the transaction
holding the lock

The conditional restart protocol has been observed to be useful only in cases where the only one transaction is holding a lock on a data item and also, when the transaction holding the lock is not blocked waiting for some other lock, as this may lead to a deadlock.

2.2 Scheduling in Distributed Databases

Most of the scheduling algorithms for centralized databases have been modified and adapted for distributed databases. Due to the presence of multiple sites where data is stored, the delay involved in the communication between different sites plays a very important role in the performance of these algorithms. Maintaining consistency of the distributed database systems requires considerably more effort than for centralized databases. Additionally, distributed databases are prone to distributed deadlock, which is difficult to detect and not encountered in centralized databases. If the transactions arriving in the distributed system are real-time in nature, in addition to maintaining consistency and concurrency in the database, the transactions will have to be completed before their deadlines.

As in centralized databases, the different policies proposed for distributed databases can be broadly categorized into locking and timestamp ordering protocols. Some of the most commonly used policies are described in this section.

1. Basic 2PL [2]

Most of the distributed locking protocols also use the two-phase locking protocol. To apply two-phase locking in a distributed form, synchronization among the different sites should be maintained. To eliminate data redundancy, logical copies of all the data items are stored at all sites.

In a simple form of distributed two-phase locking, schedulers are also distributed along with the database among the different sites. If a transaction requests a lock on a data item and the request cannot be completed, the transaction is placed in the waiting queue for that data item. When the lock on the data item is released, all the transactions in the waiting queue, waiting for that particular data item, are processed in first-in-first-out (FIFO) order. The basic 2PL protocol uses the *Read One - Write All* policy. To read a data item at one site, the transaction will have to obtain a read lock on the logical copy of the data item at that site. But to update the value of a data item, the transaction will have to obtain write locks on all the logical copies of the data item at all sites and broadcast the updated value. This policy can lead to distributed deadlock between transactions as a transaction T_X may be waiting for a data item held by another transaction T_Y at site A and T_Y may be waiting for a data item held by transaction T_X at site B . In addition to the deadlock problem, this policy does not take into account the real-time constraints of the transactions.

2. Centralized 2PL [2]

This policy is a modified version of the *basic 2PL* policy. In this policy, a centralized scheduler is present at a single site, which receives lock requests from all the transactions. Before a transaction can access data from any site, it has to obtain the appropriate locks from the centralized scheduler. Each transaction sends a lock request to the centralized scheduler irrespective of the location of the data item it wants to access. The scheduler acknowledges the lock request and sends its decision, i.e., whether the lock has been granted

or not. Only after receiving the decision of *lock granted* can the transaction go ahead and send a message to the appropriate site to lock the data item in the required mode.

This policy results in the centralized scheduler becoming a bottleneck, thus lowering the performance of the system. In addition, the communication required among the different sites, to obtain a lock, also increases.

3. Primary Copy 2PL [2]

This policy tackles the problem of data redundancy. Like the *basic 2PL* policy, multiple logical copies of all the data items are present, with one logical copy at each site. One logical copy of all data items is designated the *primary copy*. To access any copy of logical data item, the transaction must first request and obtain a lock on the primary copy. To obtain a read lock on a data item, a transaction must communicate with the data managers at two sites, the site where the primary copy of the data item is located and the site where the copy to be read is located. If the transaction can obtain a lock on the primary copy, it can obtain a lock on the logical copy and read its value. This leads to much more communication among sites than was required by the basic 2PL policy.

But, if the transaction wants to update a copy of a data item, it has to update all copies of the data item but request write lock only on the copy it explicitly wanted to update. This does not require more communication than originally required by the basic 2PL policy.

4. Voting 2PL [2]

This policy is also called the *majority consensus 2PL* policy and attempts to eliminate data redundancy. To achieve this, it uses the concepts of voting and two-phase commit. Two-phase commit involves the broadcast of a commit message to all the sites to find out whether the transaction is willing to commit at those remote sites. The transaction manager, which sends out the broadcast messages, waits for all the sites to respond as to whether the lock has been granted or the transaction has been blocked at that site. Upon receipt of an

acknowledgment from all sites, the transaction manager counts the number of responses indicating that the lock was granted. If the number constitutes a majority, the transaction manager assumes that all the locks were granted. If not, it waits for lock granted messages from the sites which had blocked the transaction. This technique requires strong synchronization among the different sites and the amount of communication can be very high, thereby causing performance deterioration.

In policies where the real-time constraints of the transactions are taken into account [20], almost all policies assume the presence of distributed schedulers at all sites, with the scheduler at a site responsible for locks on data items present at that site. These policies also assume that when each transaction is started a process called the *coordinator* is created at the site where the transaction originates. Other participants of that transaction are created at sites where processing has to be carried out. Each transaction has a unique priority and the priority of the coordinator and participants of that transaction will be the same as that of the transaction. The major goal of most of these policies is to avoid the restart of transactions, in case of a conflict, avoiding wastage of processing already completed by the transaction.

5. Distributed High-Priority Two-Phase Locking [20]

In this policy, if the priority of the requesting transaction is higher than the priority of the transaction holding the lock and if the holding transaction is in the committing phase, the transaction holding the lock will not be aborted. Instead, the requesting transaction is blocked. Otherwise, if the holding transaction is not in the committing phase, it is restarted. If the transaction has only one participant and it is located at the site, where the request originated, a local restart of the transaction is required. If the transaction has more than one participants in sites other than the site where the lock request originated, a global restart of the entire transaction is required.

```

if  $P(T_R) > P(T_H)$ 
  then if  $T_H$  is not committing
    then if  $local(T_H)$ 
      then  $localrestartof T_H$ 
      else  $globalrestartof T_H$ 
    else  $block T_R$ 
  else  $block T_R$ 

```

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

$local(T_H)$ is true if T_H has only one participant

6. Distributed Priority Inheritance Protocol [20]

Some policies use a distributed version of the priority inheritance policy, wherein, if the priority of the requesting transaction is higher than that of the holding transaction, the requesting transaction is blocked. Additionally, the priority of the requester is broadcast to all the participants of the holding transaction and their priority is updated at all sites. This policy is prone to the problem of distributed deadlock. Deadlock detection is a communication intensive job that may cause serious performance problems.

```

if  $P(T_R) > P(T_H)$ 
  then if  $T_H$  is not committing
    then if  $local(T_H)$ 
      then  $localrestartof T_H$ 
      else  $globalrestartof T_H$ 
    else  $block T_R$ 
    Broadcast Update Priority of  $T_R$  to
    all sites

```

where T_R is the transaction requesting a lock on a data item

T_H is the transaction currently holding a lock on the data item

$local(T_H)$ is true if T_H has only one participant

7. Distributed Hybrid Two Phase Locking [20]

In this policy, like the *distributed high priority two phase locking* and the *distributed priority inheritance* policies, if a transaction is committing it will not be restarted even if a lock conflict has occurred. In addition to checking all the previous conditions, it also checks if the requesting transaction is blocking a higher-priority transaction and the slack time of the requesting transaction is smaller than the remaining execution time of the holding transaction. If all these conditions hold true and the holding transaction is not in the committing phase, the holding transaction is restarted.

But several problems are encountered in the implementation of this policy such as the blocking information of the transactions at the remote sites is not known, no a priori knowledge of the location of data to be accessed by the transaction, which could provide an estimate of the remaining execution time and slack time of the transaction. The blocking information is kept at the originating site and in case of a conflict, the originating site is contacted to obtain the information. All these operations require a lot of communication among the sites. An increase in the delay involved during this communication can lead to many transactions missing their deadlines, thereby eliminating the advantages obtained by avoiding the restart of transactions using various conditions to resolve conflicts.

Chapter 3

An Overview of Implemented Scheduling Policies

Taking into consideration the different scheduling policies discussed in the previous chapter for distributed databases, two generalized policies have been implemented in this study. The primary objective is the comparison of the performance of these two policies, called the *synchronized scheduling* and the *modified scheduling* policies, with the proposed policy, called the *adaptive highest priority first* policy.

3.1 Synchronized Scheduling Policy

This policy, whose performance is compared with the proposed policy described in Section 3.3, is a basic, generalized version of the previously proposed distributed scheduling policies. This policy, called the *Synchronized Scheduling policy* works as follows. The transactions entering the system have associated with them a deadline, an approximate execution time and the list of data items they will access. The priority of the transaction is determined by these parameters. Depending on the location of the data, the query manager splits the transaction into a set of sub-transactions. Only one sub-transaction of a transaction will be present at a processor. The sub-transactions, at each processor, are queued in decreasing order of priority. If the new sub-transaction preempts a currently executing sub-transaction and these transactions access data in conflicting modes, the lower-priority transaction is restarted.

This is also carried out globally with all the sub-transactions of that transaction, present at other processors, being aborted and restarted. When the sub-transaction gets scheduled, it requests locks from the local lock manager. If all locks can be granted, the sub-transaction goes ahead and reads the data items from the disk.

If any of the data items to be accessed by the transaction could not be locked as they had already been locked by other transactions in conflicting modes, the transaction cannot go ahead and is blocked. This policy is similar to the *Always Block* [32] and the *Wait* [1] centralized policies but have been adapted to a parallel form. All the sub-transactions of that transaction, at all the processors block and are queued in the respective local block queues for a predetermined amount of delay. Thus, if a transaction cannot get all the locks at a processor, the transaction blocks at all the processors. After the expiration of the block delay time, the sub-transactions, at all the processors, attempt to get the locks again. The amount of time for which the transaction blocks gets doubled each time the transaction get blocked. This is similar to the exponential back-off policy used in communication networks to resolve collisions. As a transaction cannot proceed till it gets all the locks on the required data, the problems of local and distributed deadlock are avoided.

After the sub-transaction obtains locks on all the required data items, it reads the data from the disk and processes them. If the data is modified, the modified value will have to be written back to the disk after the transaction commits. The two-phase commit protocol, which is used to commit a distributed transaction, involves broadcasting the message to all the processors and receiving an acknowledgment from each either confirming or denying the willingness to commit. If all the sub-transactions at different processors can commit, the complete transaction commits and all modified data is written back to the disk. The transaction then releases all the locks and exits the system.

In this policy, all the decisions involved, for example, to block, restart or commit a transaction, are made globally. If a transaction blocks at one processor, it will be blocked at all the other processors, irrespective of whether it could proceed at the other processors or not. Similarly, in case of restart of a transaction at one processor,

the decision is communicated to all the other processors and the same is executed all the other processors. Continuous synchronization between the processors is required in this policy which deteriorates its performance under heavy load conditions (see Chapter 5 for details).

3.2 Modified Synchronized Scheduling Policy

This policy is a slightly advanced version of the *synchronized scheduling policy* and whose performance is also compared with the proposed policy. The *modified Synchronized scheduling policy* is similar to the *synchronized scheduling policy* except the manner in which it handles lock conflicts resulting in reduced number of transaction blockings in the system.

Similar to the *synchronized scheduling policy*, the incoming transactions are allotted individual deadlines, estimated execution times and the data items they will access, which will eventually be used to determine their priority. Depending on the location of the data to be accessed, the transaction is split into sub-transactions and queued at each processor, ordered by priority. If the sub-transaction is scheduled, it requests the local manager for appropriate locks on the data items to be accessed. If a data item has already been locked by some other transaction in conflicting mode, the requesting sub-transaction checks the priority of the sub-transaction holding the lock. If more than one sub-transaction has locked the data item in shared mode, the highest priority, among all the holding sub-transactions, is considered as priority of the data item.

If the priority of the requesting sub-transaction T is lower than the priority of the data item, this decision is broadcast to all the processors and the requesting sub-transaction is blocked globally at all the processors. All the sub-transactions of that transaction are enqueued in the block queues at their respective processors. If the priority of the requesting sub-transaction is higher than the priority of the data item, the remaining execution time of the sub-transaction holding the lock is compared with the slack time of the requesting sub-transaction. In the case where more than

one sub-transactions have locked the data item. the maximum remaining execution time among all the sub-transactions is considered as the remaining execution time of the data item. If the slack time of the requesting sub-transaction is greater than the remaining execution time of the data item. the local decision is made to block the requester. Conversely. if the slack time of the requesting sub-transaction is less than the remaining execution time of the data item. the local decision is made to abort all the sub-transactions holding locks on the data item.

After the local decision has been made. the local processor communicates with all the sub-transactions of the requesting transaction at the other processors. to check if they can proceed at their respective processors. to obtain the global decision. The decisions of all the remaining sub-transactions at the other processors is communicated back to all the other processors. To make the final decision based on the local and various remote decisions. the following rules are considered.

- If at any one processor. the requesting sub-transaction T has lower priority than the priority of the sub-transaction holding a lock on the data item required by T . a global decision is made to block the transaction T globally.
- If at any one processor. the requesting sub-transaction T has a lower slack time than the remaining execution time of the data item T wants to access and the decision was made to abort the sub-transactions currently holding a lock on the data item. a global decision is made to abort the transactions that are holding locks in conflicting modes on the data items T wants to access. Aborting a transaction is done globally with all the sub-transactions of the transaction aborted. at all of the processors and restarted after a predetermined amount of delay.
- If at all the processors. the requesting sub-transaction T has a higher slack time than the remaining execution time of the data items T wants to access. the global decision to block the entire transaction T is made. All the sub-transactions of T at all the processors are blocked and queued in their respective block queues.

This policy involves very heavy communication among the processors. Firstly, local decisions made at every processor need to be broadcast to all the other processors. The final decision is made and again communicated to all the processors. In addition to this, when the transactions are blocked and aborted, this status also has to be communicated to all the corresponding processors. Because of such a large amount of communication the performance of the policy deteriorates especially in systems with large number of conflicts and under heavy loads.

If the sub-transaction can obtain all the locks required, it reads and processes the values of the required data items. If the values of the data items have been modified, the modified values are not made permanent till the transaction commits. To commit, the transaction, as in the *synchronized scheduling policy*, uses the two-phase commit protocol and has to communicate to the other processors, to obtain the information if other sub-transactions are ready to commit. After the transaction commits, all the updates made are written back into the disk and made permanent. The transaction, then exits the system.

3.3 Adaptive Highest Priority First Policy

Most of the existing policies have been designed for centralized databases and do not take into account the problems involved in distributed databases. The scheduling policies proposed for distributed databases are simplified versions of the existing centralized policies. These policies involve frequent communication between the different sites leading to substantial communication delays when querying the database. When adapting the distributed algorithms to parallel systems, the communication across the different processors increases, thereby reducing considerably the advantages of using a shared-nothing architecture.

The policy being proposed here takes into consideration the problems associated with existing policies for distributed systems. The major advantages of this policy over existing ones are:

- The processing decisions at each processor are purely local. They do not

involve communication among the processors, thereby, eliminating the communication delay present in existing policies.

- The real-time transactions are not burdened with initial, outdated priorities as the priorities are updated every time a new transaction arrives into the system. If the priority assignment policy used is *Least Slack First*, the continuous updating of priorities makes a considerable difference in the performance of the policy as the slack time of the transaction will decrease with time resulting in increase in the priority of the transaction. Therefore, transactions always have the most up-to-date information, thereby leading to better scheduling among the existing transactions.

The Adaptive Highest Priority First policy works as follows. As in the existing policies for distributed real-time architectures, the transactions coming into the system have associated with them a deadline, an approximation of the execution time and a set of the data items the transaction will access. On the basis of the deadline and execution time, a priority is assigned to the transaction. This priority is not fixed and will change through time depending on the priority assignment policy and the amount of work the transaction has already completed in the system. Also associated with each transaction is its criticality. Criticality of a transaction can be defined as the degree of importance of the transaction. Depending on the location of data, the initial transaction is split into a number of sub-transactions at different processors. Note that, at a processor, only one sub-transaction of a particular transaction is present.

At each processor, the sub-transactions are queued according to decreasing order of their priority. In this policy, the transactions can go ahead only if they get locks on all the data items to be accessed, thereby eliminating the problem of deadlock. Before a sub-transaction can request locks, some checks have to be made. These checks are done locally at each processor, with no communication among the processors. At each processor, a status queue is maintained, which keeps track of the status of all the sub-transactions at that processor, irrespective of their actual posi-

tion. These checks are performed on the status queue, thereby, considering all the sub-transactions at that processor. The two basic rules to be followed in this policy are as follows.

Backward Rule: This rule checks all the sub-transactions that are enqueued behind the newly arrived sub-transaction. If any of these sub-transactions has been scheduled and access data in a conflicting mode with the new, higher-priority sub-transaction, they release all their locks on the data and are restarted. Note that only the local sub-transaction is restarted.

Forward Rule: This rule checks all the sub-transactions enqueued in front of the newly arrived sub-transaction. If any of these higher priority sub-transactions are blocked and will be accessing data in a conflicting mode with the new, lower-priority sub-transaction, the new sub-transaction will be blocked. Again only the local sub-transaction is blocked.

If the newly arrived sub-transaction satisfies the two rules, it can be scheduled. The sub-transaction proceeds to request locks from the local lock manager. If it cannot get locks on all the data items, the sub-transaction blocks. Unlike in the synchronized scheduling policy, the blocking is done locally with no effect on the other processors. On getting all the required locks, the sub-transaction goes about the process of reading the data from the disk and carrying out the necessary processing. If the value of the data item read has been modified, the sub-transaction maintains a local copy of the modified value, thus eliminating the need to rollback the sub-transaction in case of an abort. The only situation where communication among the different processors is required in this policy, is at the time the transaction commits. The two-phase commit protocol is considered in this case, which requires synchronous communication among the processors. The processor at which the final sub-transaction completes is designated as the coordinator of the commit protocol. After a transaction has committed, it cannot be aborted.

After the transaction completes writing the modifications back into the disk and making them permanent, it exits the system. Before exiting the system, each

transaction releases all locks held by it. At this instant, all the sub-transactions blocked at the respective processors are checked to determine if any of them can obtain locks on all the data items it wants to access. As all sub-transactions have been ordered in decreasing order of priority in the status queue and the adaptive HPF policy does not allow any sub-transaction to go ahead if it accesses any page in conflicting mode with a blocked, higher priority transaction enqueued in front of it (Forward Rule), a check is made beginning with the highest priority transaction. Associated with each sub-transaction is the knowledge as to how many data items the sub-transaction is waiting for, i.e., on how many data items the sub-transaction could not obtain a lock. During the check to determine if the sub-transaction can be scheduled, if the count becomes zero, the sub-transaction can obtain all the required locks and it is scheduled.

To maintain the current priority of the sub-transactions at each processor, each time a new sub-transaction arrives at a processor, the priorities of the existing sub-transactions at that processor are updated. If the priority assignment policy used is *Least Slack First*, the priorities of the sub-transactions increase as the slack time has decreased. During this period, if a transaction has missed its deadline, it is immediately aborted as hard transactions are being considered with no flexibility given in satisfying the deadline. In addition to maintaining a priority-based queue to determine if the sub-transaction can be scheduled, processing at the disk and the CPU are also priority-based, thus providing the most emphasis on the real-time property of the sub-transactions.

Correctness of the database is maintained by this policy as the Forward and the Backward rules force all the processors to maintain the same global order of execution of transactions. Note that this policy avoids deadlocks at the local level as well as at the global level. Local deadlocks are avoided as a transaction cannot proceed until it obtains locks on all the required data items. To avoid global deadlock, the global serializability of transactions has to be maintained on all the processors. Thus, the order in which transactions will be scheduled is the same at all the processors. It can be observed that some delay is involved in the continu-

ous calculation of priority and in the elaborate checks conducted, but all these are localized with no communication whatsoever among the different processors. All these delays are incorporated into our simulation model. Simulation experiments conducted on this policy have shown it to perform better than existing scheduling algorithms for distributed systems.

But inspite of its advantages, the *adaptive HPF* policy has some disadvantages. To maintain serializability among transactions in a distributed environment without communication among the various processors, the *Forward Rule* enforces a very restrictive condition for the transactions. As transactions cannot be scheduled if a blocked, high-priority transaction enqueued before it accesses pages in conflicting modes, the probability of chained blockings, in which transactions are blocked due to conflict with the preceding transactions, increases in environments with very high conflicts among transactions.

Chapter 4

Simulation Model

4.1 Workload Model

The workload model characterizes the workload of the database, i.e., the transactions accessing the database. These transactions can be characterized by the amount of data they access, the nature of the accesses (read or write), the priority assigned to them, their criticality, the resources they will access and so on.

Transactions will arrive at an exponential rate with mean λ transactions/sec. Each arriving transaction is associated with its time of arrival, its deadline and criticality. Criticality value is uniformly distributed between 1 and 5, where a criticality value of 1 makes a transaction most critical and a criticality of 5 makes the transaction least critical. The deadline of the transactions is calculated using the following function [32].

$$deadline_T = arrival_T + slack_T + service_T \quad (4.1)$$

where $deadline_T$ is the deadline of the transaction T

$arrival_T$ is the time of arrival of the transaction T

$slack_T$ is the slack time of the transaction T

$service_T$ is the estimated execution time of the transaction T

Slack time of the transactions has been calculated by two different methods. The first method involves the calculation of slack time independent of the estimated

execution time of the transaction. The slack time, in this case, is not correlated to the transaction itself and is calculated using the following function.

$$slack_T = uniform_distribution(min_slack_time, max_slack_time) \quad (4.2)$$

where *min_slack_time* is the minimum value of slack time that can be allocated to all the transactions (default value)

max_slack_time is the maximum value of the slack time that can be allocated to all the transactions (default value)

The slack time of each transaction will be uniformly distributed between a pre-defined minimum slack time and maximum slack time.

The second method used in calculating the slack time ensures that the slack time is dependent on the execution time of the transaction. The correlated slack time has been calculated to be a function of the estimated execution time of the transaction shown as follows.

$$slack_T = slack_rate * service_T \quad (4.3)$$

where *slack_rate* is the factor used to determine the slack time of transaction *T* (default value)

The results presented in this thesis use uncorrelated slack time and the results for correlated slack time are available in Appendix A and B.

The estimated processing time of the transaction *T*, *service_T* can be calculated as follows.

$$service_T = lock_grant_delay_T + processing_delay_T + io_delay_T \quad (4.4)$$

where *lock_grant_delay_T* is the delay for granting locks to the transaction *T*

processing_delay_T is the delay for processing the data of the transaction *T*

io_delay_T is the delay for the transaction *T* accessing the disk

to read the data required by the transaction

The delay for granting locks is the amount of time spent by the transaction to request locks on the data items and to acquire those locks. It can be calculated as follows.

$$lock_grant_delay_T = pages_accessed_T * lock_time \quad (4.5)$$

where $pages_accessed_T$ is the actual number of pages accessed by the transaction T
 $lock_time$ is the delay encountered by transaction T to request the lock manager for lock on a page and actually locking that page (default value)

Processing delay and the delay involved in accessing the disk are the amount of time spent by the transaction at the CPU and disk respectively and are dependent on the number of data items to be processed and read by the transaction respectively. These delays can be calculated as follows.

$$processing_delay_T = pages_accessed_T * cpu_time \quad (4.6)$$

$$io_delay_T = pages_accessed_T * io_time \quad (4.7)$$

where cpu_time is the delay at the CPU for transaction T to process one page (default value)
 io_time is the delay at the disk for transaction T to access one page (default value)

The locking delay depends on the number of data items to be locked by the transaction. Similarly, the processing delay and the delay involved in accessing the disk also depend on the number of data items to be processed and read from the disk respectively.

To determine the priority of the transaction, the following function is used.

$$priority_T = \alpha_1 \left(\frac{1}{deadline_T} \right) + \alpha_2 (criticality_T (deadline_T - arrival_T)) + \alpha_3 \left(\frac{1}{slack_T} \right) \quad (4.8)$$

where $priority_T$ is the priority of the transaction T

$criticality_T$ is the criticality of the transaction T

$deadline_T$ is the deadline of the transaction T

$arrival_T$ is the time of arrival of the transaction T

$slack_T$ is the slack time of the transaction T

α_1 , α_2 and α_3 are the coefficients used to determine which of the properties of the transaction will be used to determine its priority. This function gives more flexibility in determining the factors that can be involved in determining the priority of the transaction.

At the time a transaction enters the system, the transaction is characterized as a *read-only* transaction or a *read-write* transaction. The number of read-write transactions in the system can be controlled and is a deciding factor in determining the load on the system. If a transaction is a read-write, the number of pages the transaction has to modify and update the values on the disk can also be controlled. The number of write pages of a transaction are a fraction of the total pages which will be accessed by the transaction. The number of pages to be written can be determined as follows.

$$write_pages_T = frac_write * pages_accessed_T \quad (4.9)$$

where $write_pages_T$ is the number of pages to be written by transaction T

$frac_write$ is the fraction of total pages accessed by transaction T to be written

The fraction of pages to be written by each transaction can be varied to facilitate experimentation of the system at varying amount of load.

In summary, the following assumptions were considered:

- Each arriving transaction is associated with its time of arrival, deadline and criticality.

- Deadline is calculated as the sum of the time of arrival, the slack time and the estimated execution time of the transaction.
- Slack time can be calculated either to be uniformly distributed between a predefined minimum and maximum slack time or a function of the estimated execution time of the transaction.
- Locking delay, processing delay and delay involved in accessing the disk depend on the number of data items to be accessed by the transaction.
- The number of write pages of a transaction are a fraction of the total pages which are to be accessed by the transaction.

4.2 System Model

The system model characterizes the design and behaviour of the actual parallel database system. The model used in this study is shown in Figure 4.1. Upon arrival, all the transactions are allocated the data to be accessed, their deadline and correspondingly, their priority. Ordered by priority, with the highest priority first, the transactions are enqueued in the ready queue. All the transactions in the ready queue are ready to be scheduled. They possess everything necessary to get scheduled except the processor. At the time a transaction is scheduled, the centralized query manager is enabled which determines at which processors, the data to be accessed by that transaction, is located. Depending on the location of data, the transaction is split into as many sub-transactions as the number of processors where the data to be accessed is located. Note that only one sub-transaction of a particular transaction will be present at a processor.

At each processor, the incoming sub-transaction is enqueued in the wait queue, also ordered by decreasing value of priority. Additionally, the status of the sub-transaction is maintained in the status queue, which keeps track of the status of all the sub-transactions at that processor, irrespective of their actual position. Depending on the two rules of the adaptive HPF policy, the decision whether the

sub-transaction can be scheduled or will be blocked is made. If the sub-transaction is scheduled, it requests locks on the data items it wants to access.

To check whether the sub-transaction can be scheduled or not, the sub-transactions preceding and following it in the status queue are checked. The delay involved in carrying out these checks depends on the number of scheduled sub-transactions following the new sub-transaction and the number of blocked sub-transactions preceding it and can be calculated as follows.

If no conflict is found :

$$total_compute_delay = M \log_2 .N * compute_delay \quad (4.10)$$

If a conflict is found :

$$total_compute_delay = \frac{M}{2} \log_2 .N * compute_delay \quad (4.11)$$

where *compute_delay* is the delay to check for Rule 1 and 2 on one page (default value)

M is the number of pages to be accessed by the new sub-transaction

N is the total number of pages of all sub-transactions to be checked

To calculate the delay involved in checking for Rule 1 and 2, the best search algorithm, with an order of $\log .N$ is assumed to be used. To find that there is no conflict, all the pages of sub-transactions following and preceding the new sub-transaction will be checked. Conversely, if there is a conflict, an average of half of the pages of the new sub-transaction will be checked.

The local lock manager, present at every processor, is responsible for granting the locks, on data items stored at that processor, to sub-transactions. Unless a sub-transaction can obtain all locks, it is not allowed to proceed. If the sub-transaction is unable to get all the required locks, it blocks. A blocked sub-transaction can only be unblocked if it can receive all the required locks.

In the *modified synchronized scheduling* policy, the decision to block the sub-transaction or to abort the sub-transactions holding the locks is made first, decided

Figure 4.1: High-level block diagram of the system model

locally and then, the decision is broadcast to all the related sub-transactions at other processors. The delay encountered in communicating to the processors is calculated as follows.

If at one processor, the requesting sub-transaction has a lower priority than the sub-transactions currently locking the data item, the decision is broadcast to all the other processors.

$$total_communication_delay = communicate_delay \quad (4.12)$$

where *communicate_delay* is the delay to broadcast the decision to all the concerned processors (default value)

If at all the processors the requesting sub-transaction has higher priority than the sub-transactions holding the lock but the remaining execution time of the holders is less than the slack time of the requesting sub-transaction, the final decision is made to block the requesting transaction. In such a situation, each processor would have to send its local decision to every other processor.

$$total_communication_delay = N * communicate_delay \quad (4.13)$$

where *N* is the total number of processors at which the requesting transaction has sub-transactions present

If the requesting sub-transaction has a lower slack time than the remaining execution time of the holders, the final decision is made to abort the holding transactions. In such a situation, firstly, the local decision of each processor will have to be broadcast to all the other processors. Then, if at even at one processor the local decision made was to abort the holders, the final decision will be to abort and will be broadcast to all the other processors.

$$total_communication_delay = (N + 1) * communicate_delay \quad (4.14)$$

Two types of locks can be set on a data item: read lock and write lock. If a transaction wants to perform a read operation, the transaction has to set a read lock on that data item. The read lock is a shared lock, i.e., many transactions can set a read lock on the same data item at a given time. If a transaction wants to perform a write operation, the transaction has to set a write lock on that data item. The write lock is an exclusive lock i.e., only one transaction can write lock a data item at a given time. If a transaction is attempting to set a write lock on a data item which has already been read or write locked by another transaction, the lock request will be denied as it tries to lock the data item in a *conflicting mode*.

If a sub-transaction gets all the locks on the required data items, the sub-transaction proceeds to read the data from the disk. The disk queue is also ordered by priority. Upon completion of reading the data and storing the values, the sub-transaction goes ahead with the required processing of the data. This might involve only using the data or modifying them. In the event of the data being modified, the modified value will have to be written back to the disk. This can only be accomplished after the complete transaction commits. At this time, the modified value is stored in the local log of the transaction. After completing the processing of data, the sub-transaction tries to commit using the two-phase commit protocol.

The property of atomicity of transactions requires that if a distributed transaction completes, either all of its operations are carried out or none of them. As the transactions are distributed in this case, the operations requested by the transaction are at more than one processor. The two-phase commit protocol serves to preserve the atomicity of the transaction and the consistency of the database [7]. The first phase of this protocol, called the *voting phase* involves the originator of the process, designated as the coordinator, to inform the other processors, designated as clients, their status, i.e., if they are ready to commit or not. Each processor makes its own decision and informs the coordinator of its decision. The client processors have an option to either make a decision to commit or abort the transaction. After the decisions from all the processors have been received by the coordinator, the second phase of the two-phase commit protocol comes into play. The second

phase, called the *completion phase* involves the coordinator making the final decision and informing the other processors of the decision. Each transaction at the client processors then carries out the joint decision. Even if one of the client processors made the decision to abort the transaction, the entire transaction is aborted at all the processors. Only if all the processors make the decision to commit, will the transaction be committed. A transaction cannot be aborted after it has committed. After the transaction has committed, the changes to the data values will be written permanently into the disk and the transaction releases all locks and exits the system.

In summary, the following assumptions were considered:

- Only one sub-transaction of a particular transaction will be present at a processor.
- To calculate the delay involved in checking for Rule 1 & 2 in the adaptive HPF policy, the best search algorithm available with $O(\log N)$ complexity is used.
- The final decision made is broadcast to all the processors and the delay involved in this communication is considered in the simulation model.
- The transaction is assumed to use the distributed two-phase commit protocol at the time of commit. At the time of commit, the processor at which the last sub-transaction completes is assumed to be the coordinator of the two-phase commit protocol.

4.3 Database Model

The model of the database considered consists of a number of relations, each containing a large number of tuples. Each relation is horizontally fragmented across different disks. The degree of fragmentation or declustering of the relations can be varied to facilitate experimentation. The number of partitions into which a relation can be divided is decided by the following function.

$$number_of_partitions = frac_partition * no_of_processors \quad (4.15)$$

where *number_of_partitions* is the number of partitions into which each relation is divided

frac_partition is the degree of declustering.

A value of 1 implies full declustering with a partition on every disk.

no_of_processors is the total number of processors in the system.

The complete information of the partitioning of each relation and the position of its each fragment in the disk is maintained in the directory. The directory is used by the query manager to determine to which processors the transaction will be assigned. The locking granularity of the database will be at the page-level. The size of the page considered should be large enough to reduce disk accesses and wastage of CPU time and small enough to avoid unnecessary data transfer across the network. In addition to the declustering of data, static and dynamic skew effects also have to be taken into consideration.

The status of each page of every relation in the database is kept track of by the local lock manager at the concerned processor. The lock manager maintains the status of each page, stored at that processor, whether it is locked or not. If the page is locked, the mode in which it is locked, whether read or read-write, is also maintained. Also maintained at the lock manager is the id of the sub-transaction or sub-transactions that have locked the page.

Static skew is observed when data are stored non-uniformly across a subset of the disks. This type of uneven distribution can lead to some processors becoming potential bottlenecks. *Dynamic skew* is observed due to non-uniform data accesses across different processors by the transactions. The processor which will be accessed more often can also become a potential bottleneck at high loads. To investigate the effects of both dynamic and static data skew on the performance of the system, the database is designed in the following manner [15].

If

T = size of the database (in pages)

P_S = number of pages in the skewed partition of a relation

P_U = number of pages in each of the unskewed partitions of a relation

N = number of processors in the system

σ = the degree of data skew

σ can be given as follows.

$$\sigma = \frac{|P_S - P_U|}{P_S} \quad (4.16)$$

In the model, only one skewed processor is considered, which can be a potential bottleneck in the system. Then, the following can be deduced.

$$\begin{aligned} P_S + (N - 1) * P_U &= T \\ \Rightarrow P_S &= \frac{T}{1 + (N - 1) * (1 - \sigma)} \end{aligned} \quad (4.17)$$

$$\Rightarrow P_U = \frac{(1 - \sigma) * T}{1 + (N - 1) * (1 - \sigma)} \quad (4.18)$$

Depending on the formulae for P_S and P_U , the size of the skewed partition of each relation is determined and stored at the predetermined skewed processor. Only one partition of each relation can be present at a processor.

In summary, the following assumptions were considered:

- The database is assumed to consist of relations, each containing a large number of tuples.
- The number of partitions into which a relation will be divided is a factor of the total number of processors in the system and can be varied.
- The directory maintains complete information on the partitioning of each relation and the locations of its fragments in the disk.
- Locking granularity is assumed to be at the page level.
- The status of each page of every relation in the database is kept track of by the local lock manager at each processor.

- Effects of both data and execution skew are taken into consideration in comparing the performance of the three policies.
- Only one skewed processor is considered which can be a potential bottleneck in the system.

Chapter 5

Performance Evaluation and Analysis

In this section, the performance of the proposed Adaptive HPF policy is compared with two existing policies, the synchronized scheduling and the modified synchronized policies. The default values of the transaction parameters used in the simulation are summarized in Table 5.1. The number of nodes is fixed at 10. *page_mean* is the mean number of pages accessed by all the transactions with a default value of 13 pages. *conflict_delay* is the delay experienced to check for both the Forward and Backward Rule of the adaptive HPF policy. *compute_delay* is the time required to recompute the priority of all the existing transactions in the system. This delay is encountered only if the priority assignment policy used is *Least Slack First*. *lock_delay* is computed to be the time required to check and set the lock on a page to be accessed by a transaction. *write_percent* is the percentage of write operations each transaction will carry out. The slack time is distributed uniformly between 200 and 400.

Due to the small size of the database and the large number of pages accessed by each transaction, the possibility of conflict is very high in this system. Therefore, the experiments can be said to have been conducted on a heavily loaded system. This is very important as real-time database systems have applications in critical environments, where the reliability of the system is top priority and especially in highly loaded environments. All these experiments were carried out for 30 batches of 2000 transactions each and with a *write_percent* of 25%.

Table 5.1: Some simulation parameters and their default values

Parameter	Meaning	Default value
<i>BATCHES</i>	Number of Batches	30
<i>N</i>	Number of nodes	10
<i>db_size</i>	size of the database	400 pages
λ	Mean transaction arrival rate	36 ms
<i>page_mean</i>	Mean pages accessed by each transaction	13 pages
<i>cpu_time</i>	CPU time per page	8 ms
<i>io_time</i>	I/O time per page	20 ms
<i>lock_time</i>	Time to check and lock a page	2 ms
<i>commit_delay</i>	Time to commit a transaction	0.1 ms
<i>block_delay</i>	Time a transaction remains blocked before retrying	4 ms
<i>conflict_delay</i>	Time necessary to check Rule 1 and 2	0.1 ms
<i>compute_delay</i>	Time to recompute priority of existing transaction	1 ms
<i>cost_delay</i>	Time to globally block a transaction	2 ms
<i>communicate_delay</i>	Communication Delay	1 ms
<i>slack_time</i>	Slack time	unif[200...400]
<i>frac_partition</i>	Degree of declustering of a relation	1
<i>write_percent</i>	Percentage of pages written by a transaction	25%
<i>num_trans</i>	Number of Transactions	2000

The primary performance metric used in analyzing the results of the experiments is the *miss percent* of the system, which can be defined as the percentage of transactions that do not complete before their deadlines [30].

Our main objective in this section is to demonstrate that the adaptive HPF policy is relatively superior to the synchronized scheduling and modified synchronized policies and to show that the adaptive HPF policy is more robust to the various workload and system parameters. The results shown here use *Earliest Deadline First* as the priority assignment policy with the slack time of transactions uniformly distributed between a predefined minimum slack time and a maximum slack time. The performance results with *Least Slack First* as the priority assignment policy and slack time as a function of the service time of the transaction are given in Appendix B.

Each experimentation value was averaged over 30 runs, with each run consisting of 2000 transactions. 95% confidence intervals were obtained for the results to ensure independent values. The value of the confidence interval for each statistical data point is less than 9% of the point estimate when the system is heavily loaded and less than 5% of the point estimate in a lightly loaded system.

5.1 Sensitivity to Inter-arrival Time

Inter-arrival time determines the load of the system. As the inter-arrival time increases, the average time between the arrival of two consecutive transactions increases, thus reducing the load of the system. Similarly, if the inter-arrival time of transactions in a system is low, transactions arrive faster in the system thereby making the system heavily loaded.

The performance of the synchronized scheduling, the adaptive HPF and the modified synchronized policies is shown in Figure 5.1. It can be observed that even at higher loads, the percentage of transactions missing their deadlines, in the adaptive HPF policy, is much lower than for the other two policies. Therefore, it can be deduced that the adaptive HPF policy is more capable of handling transactions

Figure 5.1: Sensitivity to inter-arrival time

at higher loads and the system does not become saturated easily. But, in the other two policies, the system, when swamped with transactions, can become saturated very easily. Thus, if the inter-arrival time is decreased further, it would lead to most of the transactions missing their deadlines in the synchronized scheduling and the modified synchronized policies.

The performance superiority of the HPF policy is due to the following reasons. The main drawback with the synchronized policies is that, whenever a sub-transaction gets blocked or aborted, all sibling sub-transactions of the transaction are blocked or aborted. This strategy particularly wastes resources, which are particularly valuable in a highly loaded system. This point is illustrated by means of an example. Assume that there is a local conflict and a sub-transaction of T_x is aborted at node 1. Suppose further that there is no local conflict at any of the remaining nine nodes. The synchronized policies abort all ten sub-transactions of T_x , even though it is not necessary to abort the other nine sub-transactions. This is precisely what the adaptive HPF policy avoids. Therefore, its performance is superior to that of the synchronized policies. Additionally, since the adaptive HPF policy localizes all decisions, a blocked sub-transaction is unblocked only when the required locks are available. There is no need to repeatedly try to get the locks as is necessary in

Figure 5.2: Sensitivity to data skew

globally synchronized policies.

Note that if the priority assignment policy used is *Least Slack First*, communicating recomputed priorities causes only a negligible overhead as they can be piggy backed on to the sub-transactions of the newly arrived transaction into the system. Since it is necessary to send these values only to the nodes that receive sub-transactions of the new transactions, no additional messages are needed to communicate these updated priority values to nodes.

5.2 Sensitivity to Data Skew

Data skew modifies the distribution of data and transactions across the processors. The skewed processor has more stored data than the other processors and thus, more transactions are scheduled at the skewed processor. This can lead to the skewed processor becoming a bottleneck and the degree of data skew in the system can drastically affect the performance of the system.

The performance of the synchronized, modified synchronized and adaptive HPF policies with various degrees of data skew is shown in Figure 5.2. The value of inter-arrival time used is 45ms. As the degree of data skew increases, the load on

Figure 5.3: Sensitivity to write percent

the predetermined skewed processor also increases and can lead to an increase in the number of transactions missing the deadlines. It can be observed that the adaptive HPF policy is relatively less sensitive to the degree of data skew. It maintains a low miss percentage with the skewed processor on an average of 95% disk utilization. Therefore, it performs very favourably on systems with a high degree of data skew.

On the other hand, both the synchronized scheduling and the modified synchronized policies exhibit similar behaviour and are very sensitive to a slight increase in data skew. A further increase in skew can easily saturate the systems for both the synchronized policies.

5.3 Sensitivity to Write Percent

Write percent is defined as the percentage of total pages accessed by a transaction, that will be modified and updated. As the percentage of pages to be written increases, the amount of time spent by the transaction in the system increases.

The performance of all the three policies with varying values of write percent is shown in Figure 5.3. With an inter-arrival time of 47ms, as the write percent increases, the percentage of transactions missing their deadlines also increases. At 0%

write, i.e., when all the transactions are read-only, there will be no conflicts among the transactions. Thus, all three policies perform identically with a *miss_percent* of almost 0%. On the other hand, if the priority assignment policy used was *Least Slack First*, the continuous evaluation of priority in the adaptive HPF policy, enables it to perform better even with no conflict, due to adaptive HPF's superior scheduling of the transactions, as shown in Appendix B.

The adaptive HPF policy performs better than both the synchronized policies at low *write_percent* but as the percentage of pages to be written approaches 50%, it can be observed that the miss percent increases rapidly in both the synchronized policy and the adaptive HPF policy. The modified synchronized policy, on the other hand, performs better at high *write_percent* values. This is due to the fact that as the transactions remain longer and longer in the system, the probability of chained blockings of transactions increases in the adaptive HPF policy. As the chained blocking condition is not encountered in the modified synchronized policy, due to global blockings and aborts, the percentage of transactions missing their deadlines at higher load settings remains stable. The synchronized policy performs the worst among the three policies as it does not take the priority of transactions into consideration to resolve lock conflicts. Additionally, in contrast to the modified synchronized policy which considers the time the transaction can afford to wait (i.e., the slack time) to resolve lock conflicts, the synchronized policy always blocks the requesting transaction irrespective of its priority and slack time. Therefore, the synchronized policy performs worse than both the other policies.

5.4 Sensitivity to Slack Time

Slack time is the maximum amount of time a transaction can spend waiting without missing its deadline. Slack time is assigned to a transaction to provide it with some extra time, to allow it the opportunity to complete before its deadline. If the slack time of a transaction is reduced, the waiting time of the transaction reduces and thus, probability of a transaction missing its deadline increases.

Figure 5.4: Sensitivity to slack time

The performance of all three policies is shown in Figure 5.4. It can be observed that the adaptive HPF policy performs better than both the synchronized policies, even at low slack times. When the slack time is low, the deadline of the transactions is very tight, thus, reducing the time a transaction can wait without missing its deadline. Both the synchronized and the modified synchronized policies perform identically with a high miss percent, when the slack time is tight, but the modified synchronized policy pulls ahead of the synchronized policy, when the slack time is fairly loose, due to its superior scheduling of transactions. At very high slack times, all the three policies converge to 0 as the slack time is more than sufficient for almost all the transactions to complete within their deadlines.

5.5 Sensitivity to Declustering

Declustering is defined as the partitioning of data across the different disks in the system. In this system, degree of declustering is considered to be the degree of partitioning of relations across the different disks. An increase in the declustering of a relation enables partitions of the relation to be placed on more disks, thus improving load balance in the system. A degree of declustering of 1 partitions each

Figure 5.5: Sensitivity to declustering

relation in the database across all the processors in the system, resulting in uniform access of data at all the processors.

Note that declustering is similar to static data skew by increasing the amount of data to be stored at a processor, but in the case of declustering, the increase in data stored is not limited to one processor, termed as skewed processor, but is distributed throughout the system. Also, the data skew considered in this system takes into account both data and execution skew.

The performance of all the three policies at varying degrees of declustering can be observed from Figure 5.5. As can be seen from the figure, the adaptive HPF policy performs better than both the synchronized and the modified synchronized policy even at low degrees of declustering. A low degree of declustering results in a relation partitioned across selected processors, thus increasing the amount of data per partition. This leads to more transactions accessing each processor. The adaptive HPF policy remains fairly robust even at low declustering, confirming the fact that the policy remains stable even at high loads. The other two policies result in very high *miss_percent* at low declustering, with the modified synchronized policy performing better as the degree of declustering increases. Further, the adaptive HPF policy still maintains an edge even at full declustering, where every relation is

Figure 5.6: Sensitivity to size of the transaction

partitioned across all the processors, thereby, uniformly distributing the database across the processors.

5.6 Sensitivity to Transaction Size

The size of the transaction is the number of data items the transaction will access. If the average number of data items accessed by a transaction increases, the transactions require more locks on data items to proceed, and also spend more time in the system. This increase in transaction size tends to increase the number of conflicts and consequently, increases the percentage of transactions missing their deadlines in the system.

The performance of all the three policies for different transaction sizes is shown in Figure 5.6. When the size of the transactions is small, all the three policies perform identically, but as the size of the transactions increases, the percentage of transactions missing their deadlines in both the synchronized and the modified synchronized policies increase sharply. The percentage of transactions missing their deadlines in the adaptive HPF policy, on the other hand, increases gradually. Thus, the adaptive HPF policy maintains a stable system environment even when size of

Figure 5.7: Sensitivity to inter-arrival time (with 5 processors)

the transactions is large.

5.7 Sensitivity to Number of Processors

The default value of processors considered in the system is 10 processors. A variation in the number of processors varies the degree of multiprogramming in the system, thus improving or degrading the performance. The number of processors considered in this case is 5 processors. As the number of processors have been reduced by half of the default value, 50% more data will be stored at each processor, increasing the load on each processor. The relative performance of all the policies is considered in such a case.

The performance of all the three policies in a system with 5 processors is shown in Figure 5.7. As in the system with 10 processors, shown in Figure 5.1, the adaptive HPF policy performs better than the other two synchronized policies. The adaptive HPF policy shows a gradual decrease in the miss percent with increasing inter-arrival times, due to a decrease in the load on the system. Similarly, as in Figure 5.1, the modified synchronized policy performs marginally better than the synchronized policy, owing to its consideration of priorities in solving lock conflicts

Figure 5.8: Sensitivity of the synchronized policies to block delay

and its reluctance to abort transactions holding locks.

5.8 Sensitivity to Block Delay

The amount of time for which a transaction should be blocked, before retrying to get locks, is called *block delay*. This delay is a very important parameter in determining the state of the system. The value of block delay should not be too large to cause unnecessary waiting and should not be too small to cause repeated blockings. The optimum value of block delay depends on several system and workload parameters and often difficult to predict. However, the adaptive HPF policy does not require this parameter as a blocked sub-transaction is unblocked only when the necessary locks are available. This is primarily facilitated because global consultation is not needed in the adaptive HPF policy.

The adaptive HPF policy does not allow repeated blockings as a sub-transaction can only be unblocked, if it can get all locks requested. As repeated blockings are possible only in the synchronized and the modified synchronized policies, the adaptive HPF policy is not considered in this case. The performance of both the synchronized policies for different values of block delay is shown in Figure 5.8. The

default value of inter-arrival time used is 40ms and the average disk utilization observed in this case is 85%. It can be observed that as the block delay is increased from 4ms to 12ms, the *miss_percent* decreases. On the other hand, an increase in the delay from 12ms to 32ms causes a slight increase in the *miss_percent* but the average number of blockings per transaction drop. Therefore, it can be deduced that the optimum value of block delay for this type of system is between 8 to 12ms. A delay of value greater than 12ms is too large as this causes the transactions to wait unnecessarily and miss their deadlines.

5.9 Earliest Deadline First vs Least Slack First

The priority assignment policies used to determine the priority of transactions also play an important role in transaction scheduling. Better scheduling of transactions can result in better overall performance of the system. In all the experiments conducted so far the *Earliest Deadline First* (ED) policy was considered. Another well-known priority assignment policy is the *Least Slack First* (LS) policy. If the *Least Slack First* policy is used to calculate the priority of the transactions, in the adaptive HPF policy, the priority is updated every time a new transaction enters the system. As a transaction progresses through the system, the slack time of the transaction decreases, thereby increasing the priority of the transaction. Thus, the transactions always have up-to-date priorities and are not burdened with initial, outdated priorities. This may result in an improvement in the scheduling of the transactions in the system.

The performance of the modified synchronized policy and the adaptive HPF policy with both *Earliest Deadline First* and *Least Slack First* as priority assignment policies is shown in Figure 5.9. It can be observed that in both the modified synchronized and adaptive HPF policies, the ED heuristic outperforms the LS. The better performance of the *Earliest Deadline First* policy is due to the fact that it takes deadline of a transaction into account, which composes of the arrival time, slack time and the service time of the transaction. The *Least Slack First* policy,

Figure 5.9: Earliest Deadline First vs Least Slack First

on the other hand, takes only the slack time of a transaction into consideration. In a case where a larger transaction, i.e., with a greater deadline, and with less slack time will get higher priority if LS policy is used and might eventually cause several smaller transactions with larger slack times but lower deadlines than the large transaction to miss their deadlines. If the ED policy were to be used in this case, the smaller transactions would get higher priority than the larger transactions, thus eventually lowering the percentage of transactions missing their deadlines. The relative performance of both the policies, using the ED and LS heuristics, is similar.

5.10 Correlated vs Uncorrelated Slack Time

The slack time of a transaction is also a very important factor in determining the probability of the transaction completing within its deadline. Slack time considered is either a function of the service time of the transaction (Correlated) or is independent of the service time (Uncorrelated). In all the previous results, the slack time was calculated independent of the service time of the transaction with the value uniformly distributed between a minimum slack time and a maximum slack time. This can lead to situations where a transaction with a very large service time has

Figure 5.10: Correlated (C) Slack Time vs Uncorrelated (UC) Slack Time

a small slack time, thus practically ensuring that it will miss its deadline. In correlated slack time, as the slack time is a function of the service time, if the service time of the transaction is large, the transaction will be allocated a slack time larger than that allocated to a transaction with a lower service time.

The performance of the synchronized scheduling policy and the adaptive HPF policy with both correlated and uncorrelated slack time is shown in Figure 5.10. The priority assignment policy considered in this case is the *Earliest Deadline First*. The results using the *Least Slack First* as the priority assignment policy were observed to be relatively similar and are presented in Appendix B. It can be observed from Figure 5.10 that correlated slack time performs better than uncorrelated in both the policies with the difference between the performance of a policy with correlated and uncorrelated slack time decreasing with increasing inter-arrival time. An increase in the inter-arrival time decreases the frequency of arrival of transactions, thereby reducing the load on the system. Therefore, at low system loads, the mode of slack time calculation of the transactions does not affect the system performance, but at high system loads, the difference is considerable.

Chapter 6

Conclusions

6.1 Summary

Scheduling real-time transactions involves maintaining the consistency of the database and taking into consideration the timing constraints of the transactions. Existing scheduling policies for distributed environments involve massive communication among the different processors, thereby deteriorating the performance of the policies. A new policy has been proposed, which avoids the increasing communication problem and its performance is compared with two existing policies.

Chapter 1 gave a brief introduction of distributed and parallel databases. The similarities and differences between them have been discussed. A discussion of the concepts of real-time systems and their relevance in this study has also been presented.

Chapter 2 gave a brief overview of the existing real-time scheduling algorithms for both centralized and distributed database systems. This chapter also highlighted the differences between scheduling in centralized and distributed systems and also discussed various problems encountered in porting centralized concepts on distributed systems.

Chapter 3 discussed the proposed policy, the Adaptive HPF policy and the advantages of this policy over existing policies for distributed systems. The two existing

synchronized scheduling and modified synchronized policies, whose performance has been compared with the proposed policy are also briefly discussed in this chapter.

Chapter 4 provided a detailed description of the simulation model used in this study. This chapter discussed the workload model, system model and the database model. The various parameters considered in this study were also presented.

Chapter 5 presented the results obtained from the simulation experiments. These results show that the adaptive HPF policy performs better than the existing policies under various workload and system parameters.

6.2 Contributions of the Thesis

In this thesis, a new scheduling policy for real-time transactions in parallel database systems, called the Adaptive HPF Policy, has been proposed. The performance of this policy has been compared with two existing policies, the synchronized scheduling policy and the modified synchronized policy. The major disadvantage of the existing scheduling policies is the continuous communication required among the various processors. Every scheduling decision made in the system is global and has to be executed at all the processors. In addition to deteriorating the performance of the system, these policies also exhibit a large amount of communication overhead among the processors.

The Adaptive HPF policy, on the other hand, is purely local in nature and requires only minimal communication among the processors during the decision-making process. Communication among the processors is required only when a transaction distributed across various processors has to be killed as it has missed its deadline. Furthermore, if the priority assignment algorithm, used to determine the priority of the transactions, is *Least Slack First*, the priorities of the transactions are updated repeatedly. Thus, the transactions are always associated with current priorities, which improves overall scheduling of the transactions in the system enabling them to perform relatively better and complete within their respective deadlines.

The simulation experiments conducted have shown that the adaptive HPF policy performs better in both a lightly-loaded and heavily-loaded system. The adaptive HPF policy is seen to be robust and stable under most conditions, thereby making it a reliable policy for real-time applications. The performance of the three policies was compared under two well-known priority assignment algorithms, the *Earliest Deadline First* and the *Least Slack First* algorithms and the adaptive HPF policy performs better irrespective of the priority assignment algorithm used.

6.3 Future Work

Due to time constraints, the effect of only selected parameters on the three policies was considered in the simulation experiments. The impact of other parameters on individual as well as relative performance of the three policies, especially the adaptive HPF policy, is interesting to study.

One interesting scenario to observe would be the effect of the three policies upon variation in the size of the database. An increase in the size of the database would reduce the conflicts between transactions. In the adaptive HPF policy, the stringent conditions imposed on transactions to determine if they can be scheduled or not lead to additional transactions being blocked. What will be the effect of a large number of blocked transactions, brought about by a reduced size of the database?

In this thesis, hard real-time transactions have been considered, i.e., if the transaction cannot complete before its deadline, it is killed. If soft real-time transactions are considered, the average lateness of the transactions can be observed, i.e., by how much do the transactions miss their deadlines. The performance of the three policies on a soft real-time system is also interesting to study. As soft real-time system carries all the transactions to completion irrespective of whether they have missed their deadlines or not and this effectively increases the overall load on the system.

One of the shortcomings encountered in the proposed adaptive HPF policy is the possibility of chained blockings at individual processors. Transactions may remain blocked because they want to access certain pages in conflicting modes with trans-

actions blocked before them in the status queue. This can lead to many transactions being blocked because of blocked transactions before them. This creates a chain of blocked transactions that can only be scheduled serially. If this shortcoming can be overcome, the adaptive HPF policy would perform even better.

Appendix A

Results of Earliest Deadline First with Correlated Slack Time

The main results presented in Chapter 5 considered uncorrelated slack time assigned to transactions in the system, i.e., slack time of transactions is not dependent on the service time of the transactions but is uniformly distributed between a given minimum slack time and maximum slack time. If the slack time is correlated to the service time of the transactions, the slack time assigned to each transaction will depend on the execution time of the transaction. This tends to increase the probability of transactions meeting their deadlines particularly at heavy system loads.

A.1 Sensitivity to Inter-Arrival Time

Similar to the observation made about data presented in Figure 5.1 for uncorrelated slack time, the adaptive HPF policy performs better than the two synchronized policies in the case of correlated slack time. Even though the relative performance of the three policies remains similar to the case with uncorrelated slack time, the actual value of the *miss_percent* for varying values of inter-arrival time has decreased considerably.

The performance of all the three policies as a function of inter-arrival time is shown in Figure A.1. The adaptive HPF policy appears more stable and robust.

Figure A.1: Sensitivity to inter-arrival time

even at high system loads, than the other two policies. The other two policies tend to move the system into saturation at the highest load considered here.

A.2 Sensitivity to Data Skew

Data skew causes one of the processors in the system to act as a bottleneck. If the slack time of the transaction is correlated to the service time, the slack time will play an even more important role in determining whether the transaction completes within its deadline or not. The performance of the three policies is shown in Figure A.2.

As can be observed from the figure, both the synchronized policies are very sensitive to data skew, with the percentage of transactions missing their deadlines increasing rapidly with data skew. On the other hand, the adaptive HPF policy exhibits less sensitivity to data skew.

Figure A.2: Sensitivity to data skew

A.3 Sensitivity to Write Percent

An increase in the write percent of a transaction will result in the transaction conflicting with more transactions and spending more time in the system. The performance of the three policies is shown in Figure A.3. If the write percent is 0%, all the transactions are read-only resulting in no conflict. This will enable almost all the transactions to complete within their deadlines irrespective of the scheduling policy used (see Figure A.3).

Additionally, as in the case of uncorrelated slack time, performance of the adaptive HPF policy deteriorates when the *write_percent* approaches 60% due to chained blockings.

A.4 Sensitivity to Declustering

Similar to the relative behaviour observed so far, the adaptive HPF policy performs better than both the synchronized policies. The performance of the three policies can be compared from Figure A.4.

The adaptive HPF policy appears less sensitive to changes in declustering of

Figure A.3: Sensitivity to write percent

Figure A.4: Sensitivity to declustering

Figure A.5: Sensitivity to transaction size

relations. Even at a low degree of declustering, the adaptive HPF policy performs much better than the synchronized and the modified synchronized policies. At high levels of declustering, all the three policies exhibit similar behaviour.

A.5 Sensitivity to Transaction Size

The relative behaviour of all the three policies is similar to the case with uncorrelated slack time shown in Figure 5.6. In both cases, the adaptive HPF policy performs better than both the synchronized policies for the various transaction sizes considered. The performance of the three policies with correlated slack time is shown in Figure A.5. The default *write_percent* considered is 25%.

It can be observed from the figure that when the mean size of the transactions is less, all the three policies perform identically with a *miss_percent* of almost 0%. But as the transaction size increases, the *miss_percent* in both the synchronized policies increases rapidly with the adaptive HPF policy showing less sensitivity to transaction size.

Appendix B

Results with Earliest Slack Time

The results with *Earliest Deadline First* as the priority assignment policy have already been presented in Chapter 5. In addition to the *Earliest Deadline First* policy, the *Least Slack First* policy has also been proposed. The results with the *Least Slack First* as the priority assignment policy are presented here.

The adaptive HPF policy displays an interesting behaviour with *Least Slack First* as the policy continuously evaluates the priority of transactions in the system, whenever a new transaction arrives. This ensures that each transaction has the updated priority and thus, improves the performance of the policy relative to the other two policies. In the synchronized and modified synchronized policies, as the priority of a transaction is only recalculated when it restarts or unblocks, the effect of continuous priority evaluation on relative performance is not as much as in the adaptive HPF policy.

The slack time of each transaction can be either uncorrelated or correlated to the service time of the transaction. The effect of both on the relative performance of the policies is also presented. The uncorrelated slack time is uniformly distributed between a predetermined minimum value, *min_slack_time* and maximum value, *max_slack_time*. On the other hand, the correlated slack time is a function of the service time as discussed in Chapter 4.

Figure B.1: Sensitivity to inter-arrival time

B.1 Effect of Uncorrelated Slack Time

B.1.1 Sensitivity to Inter-Arrival Time

The observation in this case is similar to the *Earliest Deadline First* case, shown in Figure 5.1, with the adaptive HPF policy performing significantly better than both the synchronized and the modified synchronized policies. The performance of the three policies is shown in Figure B.1.

A significant difference between *Earliest Deadline First* and *Least Slack First* policies is that with the *Least Slack First* policy, even at high inter-arrival times, when the system is lightly loaded, the relative performance of the adaptive HPF policy is much better than the two synchronized policies. This can be attributed to the continuous evaluation of priority, resulting in a better scheduling of transactions and thus, better performance.

B.1.2 Sensitivity to Data Skew

The effect of increasing data skew leads to an increase in the number of transactions missing their deadlines, as the skewed processor increasingly becomes the bottleneck.

Figure B.2: Sensitivity to data skew

The performance of the three policies is shown in Figure B.2.

The adaptive HPF performs better than the other two policies. Additionally, in the adaptive HPF policy, the miss percent increases gradually with data skew, while in the other two synchronized policies, the miss percent increases rapidly with an increase in data skew. The adaptive HPF policy is more robust compared to the two synchronized policies even at high values of data skew.

B.1.3 Sensitivity to Write Percent

An increase in the write percent of each transaction results in an increase in the number of pages to be written by a transaction. As a result, the transaction remains longer in the system. This increases the probability of lock conflicts and consequently, increases the miss percentage. The performance of the synchronized, modified synchronized and the adaptive HPF policies is shown in Figure B.3.

The adaptive HPF policy performs better than the other two policies. It can be observed that even when the write percent is 0%, i.e., no conflict among the transactions as they are all read-only in nature, the performance of adaptive HPF policy is significantly better. This can be attributed to the continuous evaluation of

Figure B.3: Sensitivity to write percent

priorities of the transactions, which have an impact on performance if the priority assignment policy is *Least Slack First*. The use of current priorities to schedule transactions improves relative performance of the system even with no lock conflicts.

On the other hand, at write percent greater than 50%, as shown in Figure 5.3, the presence of chained blockings deteriorates the performance of the adaptive HPF policy and the modified synchronized policy performs the best for *miss_percent* greater than 50%.

B.1.4 Sensitivity to Declustering

The degree of declustering determines the amount of data of each relation that will be placed at a processor. A decrease in the degree of declustering leads to more data being placed at each of the processors. This increases the probability of data skew at most of the processors, thereby increasing the miss percentage. The performance of the synchronized, modified synchronized and the adaptive HPF policies is shown in Figure B.4.

It can be observed that the adaptive HPF policy performs better than the two synchronized policies at all the degrees of declustering considered. Both the syn-

Figure B.4: Sensitivity to declustering

chronized policies perform almost identically at the different declustering conditions with a very high percentage of transactions missing their deadlines.

B.1.5 Sensitivity to Transaction Size

An increase in the size of the transaction translates into the transaction spending more time in the system, requesting locks on more pages and consequently, increasing the probability of lock conflicts with other transactions. Therefore, an increase in the transaction size will result in an increase in the percentage of transactions missing their deadlines. The performance of the three policies with varying transaction sizes is shown in Figure B.5.

The relative performance of the three policies is identical to the one seen in Figure 5.6. When the size of the transaction is relatively small, all the three policies behave identically with almost all transactions completing before their deadlines. As the size of the transactions increases, the percentage of transactions missing their deadlines increases with the adaptive HPF policy showing a relatively smaller increase and thus, better performance. The modified synchronized policy shows better performance among the synchronized policies due to its superior scheduling

Figure B.5: Sensitivity to transaction size

of transactions.

B.1.6 Sensitivity to Slack Time

Slack time of a transaction determines how much the transaction can afford to wait in the system without missing its deadline. If the slack time of a transaction is reduced considerably, it would be able to wait for far less time thereby, increasing the probability of missing its deadline if the system is heavily loaded. The slack time of transactions used in this case is uncorrelated, with the slack time uniformly distributed between a predefined minimum slack time and maximum slack time. The performance sensitivity of the three policies to uncorrelated slack time is shown in Figure B.6.

As in *Earliest Deadline First* policy, the percentage of transactions missing their deadlines decreases with an increase the mean slack time. The adaptive HPF policy performs better than the synchronized policies at low slack times with all the three converging at *miss_percent* of 0% at high slack time. This condition is encountered as the slack time is enough for almost all the transactions to complete within their deadlines irrespective of the scheduling policy used.

Figure B.6: Sensitivity to slack time (uncorrelated)

It can also be observed that at very low slack times, when the deadline of the transactions is extremely tight, the synchronized policy shows relatively better performance. This is due to the fact that the synchronized policy avoids aborting transactions much more than the modified synchronized and the adaptive HPF policy, which proves a much better approach when the transactions do not have enough slack time to abort and restart.

B.1.7 Sensitivity to Number of Processors

The default value of processors considered in all the other experiments considers 10 processors. The performance of the three policies for a 5 processor system is shown in Figure B.7.

It can be observed that the relative performance of the three policies remains the same as that in a 10 processor system shown in Figure B.1, with the two synchronized policies showing similar performance at this reduced degree of multiprogramming. The adaptive HPF policy shows better performance than the other two policies at higher loads even when multiprogramming in the system is reduced.

Figure B.7: Sensitivity to number of processors

B.2 Effect of Correlated Slack Time

B.2.1 Sensitivity to Inter-Arrival Time

As observed in the previous cases, the adaptive HPF performs better than both the synchronized policies, even at high system loads. The performance of the three policies is shown in Figure B.8.

It can also be observed that due to correlated slack time, the relative performance of the three policies with the corresponding uncorrelated slack time (shown in Figure B.1) is better. Additionally, the adaptive HPF policy performs better even at light loads due its superior scheduling of transactions and minimal communication among the various processors.

B.2.2 Sensitivity to Data Skew

Effect of data skew on the three policies is also similar to that observed with uncorrelated slack time (shown in Figure B.2). The adaptive HPF policy performs better than the synchronized policies. The performance of the three policies with correlated slack time is shown in Figure B.9.

Figure B.8: Sensitivity to inter-arrival time

Figure B.9: Sensitivity to data skew

Figure B.10: Sensitivity to write percent

The adaptive HPF policy appears to be less sensitive in this case as well. We can conclude that the adaptive HPF policy is not sensitive to changes in data skew irrespective of the priority assignment policy and the type of slack time. On the other hand, both the synchronized policies are very sensitive to data skew of the system. It can also be observed that even at no skew, the adaptive HPF policy performs the best due to its superior scheduling of transactions.

B.2.3 Sensitivity to Write Percent

The relative performance of the three policies as a function of write percent remains the same irrespective of the priority assignment policy and the type of slack time. The adaptive HPF policy performs the best among the three at low write percent as can be seen from Figure B.10.

However, as the write percent approaches 50%, the miss percentage increases sharply in the adaptive HPF policy due to the presence of chained blockings. This results in the modified synchronized policy performing better at high write percent values. If the problem of chained blockings in adaptive HPF policy can be eliminated, there is no doubt that it would perform the best under all conditions of

Figure B.11: Sensitivity to declustering

system loads.

B.2.4 Sensitivity to Declustering

Using correlated slack time for transactions improves the overall performance of all the three policies. The use of low declustering of relations is an excellent method to test the performance of the three policies under high loads. The relative behaviour of the three policies is similar to that seen earlier. The adaptive HPF policy performs better than the two synchronized policies even at low degree of declustering.

The performance of the three policies is shown in Figure B.11. The adaptive HPF policy maintains a relatively low *miss_percent* even at low degrees of declustering and maintains its performance superiority even at full declustering, i.e., at a degree of declustering of 1. The two synchronized policies show relatively high *miss_percent* at degree of declustering of 0.2 and then decreases rapidly.

B.2.5 Sensitivity to Transaction Size

As with the uncorrelated slack time and *Earliest Deadline First* priority assignment policy, the adaptive HPF policy performs much better than the two synchronized

Figure B.12: Sensitivity to transaction size

policies, even when the transaction size is large. The performance of the three policies is shown in Figure B.12.

When the transaction size is comparatively small, all the three policies perform identically with a *miss_percent* of almost 0%. But, with an increase in transaction size, the *miss_percent* in the adaptive HPF policy increases gradually. In contrast to the adaptive HPF policy, the *miss_percent* of both the synchronized policies increases rapidly with increasing transaction size.

References

- [1] Abbott, R. K. and Garcia-Molina, H., Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, Vol. 17, No.3, pp. 513-560, September 1992.
- [2] Bernstein, P. A. and Goodman, N., Concurrency Control in Distributed Database Systems. *Computing Surveys*, Vol. 13, No.2, pp. 185-221, June 1981.
- [3] Bestavros, A., Advances in Real-Time Database Systems Research. *ACM SIGMOD Record*, Vol. 2, No.1, pp. 3-7, March 1996.
- [4] Boral, H. et al., Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No.1, pp. 4-23, March 1990.
- [5] Buchmann, A.P., McCarthy, D.R., Hsu, M. and Dayal, U., Time-Critical Database Scheduling : A Framework For Integrating Real-Time Scheduling and Concurrency Control. *IEEE Fifth International Conference on Data Engineering*, Los Angeles, California, pp. 470-480, 1989.
- [6] Carlson, B.M. and Dowdy, L. W., Static Processor Allocation in a Soft Real-Time Multiprocessor Environment. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No.3, pp. 316-320, March 1994.
- [7] Coulouris, G., Dollimore, J. and Kindberg, T., *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 1994.

- [8] Date, C. J.. *An Introduction to Database Systems*. Addison-Wesley Publishing Company. 1996.
- [9] DeWitt, D. and Gray, J.. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*. Vol. 35. No.6. pp. 85-98. June 1992.
- [10] DeWitt, D. J. et al.. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*. Vol. 2. No.1. pp. 44-61. March 1990.
- [11] Elmasri, R. and Navathe, S. B.. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Company. 1994.
- [12] Ghandeharizadeh, S. and DeWitt, D. J.. MAGIC: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 5. No.5. pp. 509-524. May 1994.
- [13] Haritsa, J. R.. Earliest Deadline Scheduling for Real-Time Database Systems. *Proceedings of the Twelfth Real-Time Systems Symposium*. San Antonio, Texas. pp. 232-242. December 4-6 1991.
- [14] Hong, D.. et al.. Real-Time Transaction Scheduling: A Cost Conscious Approach. *ACM SIGMOD Record*. pp. 197-206. May 1993.
- [15] Hua, K. A. and Lee, C.. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. *Proceedings of the 17th International Conference on Very Large Data Bases*. Barcelona. pp. 525-535. September 1991.
- [16] Huang, J. et al.. Experimental Evaluation of Real-Time Transaction Processing. *Proceedings of the Real-Time Systems Symposium*. Santa Monica, California. pp. 144-153. December 5-7 1989.

- [17] Jenq, B. P., Twichell, B. and Keller, T., Locking Performance in a Shared Nothing Parallel Database Machine, *IEEE Fifth International Conference on Data Engineering*, Los Angeles, California, pp. 149-158, 1989.
- [18] Kumar, V. (Ed), *Performance of Concurrency Control Mechanisms in Centralised Database Systems*, Prentice Hall, 1996.
- [19] Kuo, T. and Mok, A. K., Real-Time Database - Similarity Semantics and Resource Scheduling, *ACM SIGMOD Record*, Vol. 25, No.1, pp. 18-22, March 1996.
- [20] Lam, K. and Hung, S., Concurrency Control for Time-constrained Transactions in Distributed Database Systems, *The Computer Journal*, Vol. 38, No.9, pp. 704-716, 1995.
- [21] Livny, M. and Khoshafian, S., Multi-Disk Management Algorithms, *Proceedings of the 1987 Sigmetrics Conference on the Measurement and Modeling of Computer Systems*, Performance Evaluation Review, Vol. 15, No.1, pp. 69-77, May 11-14 1987.
- [22] Menasce, D. A. et al., Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures, *Journal of Parallel and Distributed Computing*, Vol. 28, No.1, pp. 1-18, July 1995.
- [23] Ozsu, M.T. and Valduriez, P., Distributed and Parallel Database Systems, *Handbook of Computer Science and Engineering*, CRC Press, pp. 1093-1111, 1997.
- [24] Ozsu, M. T. and Valduriez, P., Distributed and Parallel Database Systems - Technology and Current State-of-the-Art, *ACM Computing Surveys*, Vol. 28, No.1, pp. 125-128, March 1996.
- [25] Protic, J. et al., Distributed Shared Memory: Concepts and Systems, *IEEE Parallel and Distributed Technology*, Vol. 4, No.2, pp. 63-79, Summer 1996.

- [26] Ramamritham, K. et al.. Integrating Temporal, Real-Time and Active Databases. *ACM SIGMOD Record*. Vol. 25, No.1, pp. 401-420, March 1996.
- [27] Seetha Lakshmi, M. and Yu, P.S.. Effect of Skew on Join Performance in Parallel Architectures. *IEEE Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, Texas, pp. 107-120, December 1988.
- [28] Setia, S. K. et al.. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No.4, pp. 401-420, April 1994.
- [29] Shin, K. G. and Hou, C. J., Design and Evaluation of Effective Load Sharing in Distributed Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No.7, pp. 704-719, July 1994.
- [30] Son, S. H. and Park, S.. Scheduling Transactions for Distributed Time-Critical Applications. *Distributed Computing Systems*, T. L. Casavant and M. Singhal (Eds), pp. 592-619.
- [31] Son, S.H. and Chang, C.H.. Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment. *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, pp. 124-131, May 1990.
- [32] Ulusoy, O. and Belford, G. G.. Real-Time Transaction Scheduling in Database Systems. *Information Systems*, Vol. 18, No.8, pp. 559-580, 1993
- [33] Valduriez, P.. Parallel Database Systems: Open Problems and New Issues. *International Journal on Distributed and Parallel Databases*, Vol. 1, No.2, pp. 137-165, April 1993.
- [34] Yu, P. S. and Dan, A.. Performance Evaluation of Transaction Processing Coupling Architectures for Handling System Dynamics, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No.2, pp. 139-153, February 1994.

- [35] Yu. P. S. et al.. On Real-Time Databases: Concurrency Control and Scheduling.
Proceedings of the IEEE. Vol. 82. No.1. pp. 140-157. January 1994.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved