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.

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.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Scheduling on a Scalable Multiprocessor-Based Telephone Switch

By

Minru Liu, B.Eng.

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of

**Master of Science
in Information and Systems Science**

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada K1S 5B6

September 2000

Copyright 2000, Minru Liu

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.

0-612-57775-9

Canada

The undersigned recommend to the Faculty of Graduate Studies
And Research acceptance of the thesis

Scheduling on a Scalable Multiprocessor-based Telephone Switch

Submitted by Minru Liu, B.Eng. in partial fulfillment of
the requirement for the degree of Master of Science in
Information and System Science

Chair, Department of Systems and Computer Engineering

Professor Shikharesh Majumdar, Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
September 2000

Abstract

One of the key obstacles to achieving full processing capacity of a shared memory multiprocessor is memory access contention, the extent to which processes access the same location at the same time. Accesses to the same variable by two or more concurrently executing processes may result in an inconsistent state of the variable. A common way for preventing memory access contention is disjointing mutual data domains shared by concurrent processes. However, in many situations disjointing all mutual data domains is not feasible. For instance, a single processor based call control system developed over the past decades will be evolved to a multiprocessor based system. Comprehensively redesigning all mutual data domains and thus rewriting vast amount of source code is impossible from the point of view of cost effectiveness. In this case, mechanisms that control the sharing of data through not scheduling contentious processes concurrently, become attractive due to the limited code change involved. Thus the change occurs mostly in the operating system, and changes in the application programs are limited to the minimum.

This thesis investigates three scheduling algorithms that attempt to reduce the number of memory access conflicts over shared variables by controlling the concurrent executions of contentious processes. The investigation focuses on the multiprocessor based call control system in a telephone switch. Our goal is to identify the potential capacity gain through various scheduling algorithms, compare the performances of the algorithms under a number of memory contention scenarios, and identify the key factors that affect the performance of the scheduling algorithms. A simulation model has been built for the investigation, and the results of simulation are analyzed and presented.

Acknowledgements

I would like to thank my supervisor Dr. Shikharesh Majumdar for his valuable guidance, advice and help while performing this research.

I would also like to take an opportunity to thank Brian Carroll and Dale Streibel from Nortel Networks for their valuable guidance and encouragement throughout this research.

Thanks also to my manager Nicole Klappholz and Mark Christopher at Nortel Networks for their continued encouragement and excellent help.

Support from Nortel Networks and the provincial centre of excellence Communication and Information Technology Ontario (CITO) are gratefully acknowledged.

My thanks to my family who has always been there for me. I thank my daughter for being very well behaving during my studies and research. My work could have never been completed without her cooperation.

Table of Contents

Chapter 1 Introduction.....	1
1.1 Multiprocessing.....	1
1.2 Shared Memory Contention.....	6
1.3 Motivation for the Thesis.....	12
1.4 Goals of the Thesis.....	13
1.5 Outline of Thesis.....	14
Chapter 2 Multiprocessor Based Call Control System.....	15
2.1 Motivation for Moving to Multiprocessing System.....	15
2.2 Architecture of Multiprocessor Based Call Control System.....	17
2.3 Basic Concepts.....	20
2.4 Memory Contention in Multiprocessor Based Call Control System.....	22
2.5 Scheduling Algorithms.....	24
Chapter 3 Design of the Simulation.....	29
3.1 The simulation Package.....	29
3.2 The simulator.....	32
3.2.1 Overview.....	32
3.2.2 The Scheduler.....	35
3.3 The Workload.....	42
3.3.1 The Traces.....	42
3.3.2 The Blocking Parameters.....	45
3.3.3 PROPHET.config File.....	46
3.4 Output of The Simulation Model.....	47

3.4.1 Experiment Report File.....	47
3.4.2 Validation Process.....	48
Chapter 4 Results of Experiments.....	50
4.1 Performance Index and Contention Matrix.....	50
4.2 High Contention between Primary and Secondary.....	52
4.2.1 Primary Capacity.....	52
4.2.2 Overall Capacity.....	58
4.2.3 Sensitivity to Qnn.....	60
4.2.4 Capacity vs. Number of PEs.....	60
4.2.5 Summary.....	62
4.3 Low Contention Between Primary and Secondary.....	63
4.3.1 Primary Capacity.....	63
4.3.2 Overall Capacity.....	67
4.4 The Impact of Blocking Parameters.....	68
4.4.1 Variation of QNP.....	69
4.4.2 Variation of QNN.....	71
4.4.3 Summary.....	73
4.5 The Impact of Other Workload Parameters.....	73
4.5.1 Average Execution Time of Transactions.....	74
4.5.2 Frequency of Batch Switch.....	77
4.5.3 Summary.....	79
4.6 Scheduling Algorithms with No Waiting Option.....	79
4.6.1 Algorithms with No Serial Idle	80
4.6.2 Algorithms with No Sync. Idle	82
4.6.3 Summary	84
Chapter 5 Conclusions.....	85
5.1 Summary.....	85

5.2 Contribution.....	87
5.3 Future work.....	88
References.....	89

List of Figures

Figure 1-1 Distributed memory multiprocessor system	2
Figure 1-2 Shared memory multiprocessor systems	3
Figure 1-3 Symmetric multiprocessing system	5
Figure 1-4 Memory contention	7
Figure 1-5 Cache line and memory line	9
Figure 1-6 Memory access contention	11
Figure 2-1 The architecture of multiprocessor based call control system	18
Figure 3-1 PROPHET Modeling Paradigm and Modeling Classes	30
Figure 3-2 The simulator	33
Figure 3-3 The format of a trace file	43
Figure 3-4 The format of a trace	44
Figure 3-5 An example of PROPHET.config file	47
Figure 4-1 Primary Capacity in Scenario $Q_{np}=0.59$ $Q_{nn}=0.14$	53
Figure 4-2 Overall Capacity in Scenario $Q_{np}=0.59$ $Q_{nn}=0.14$	58
Figure 4-3 Primary Capacity in Scenario $Q_{np}=0.59$ $Q_{nn}=0.079$	59
Figure 4-4 Overall Capacity in Scenario $Q_{np}=0.59$ $Q_{nn}=0.079$	59
Figure 4-5 Primary Capacity in Scenario $Q_{np}=0.17$ $Q_{nn}=0.079$	64
Figure 4-6 Primary Capacity in Scenario $Q_{np}=0.17$ $Q_{nn}=0.14$	66
Figure 4-7 Overall Capacity in Scenario $Q_{np}=0.17$ $Q_{nn}=0.14$	67

Figure 4-8 Overall Capacity in Scenario $Q_{np}=0.17$ $Q_{nn}=0.079$	68
Figure 4-9 The Impact of Q_{np} Variation for Serial Algorithm.....	69
Figure 4-10 The Impact of Q_{np} Variation for Hybrid Algorithm at $s:g=1:1$	70

List of Tables

Table 3-1 Example of share allocation.....	39
Table 3-2 Serial idle from experiment versus analytical predication.....	49
Table 4-1 Rollback Overhead	56
Table 4 - 2 Rollback Serialization Idle.....	57
Table 4-3 Primary Capacity Speedup under scenario $Q_{np}=0.59$ $Q_{nn}=0.14$	61
Table 4-4 Primary Capacity Speedup for Scenario $Q_{np}=0.59$ $Q_{nn}=0.079$	62
Table 4-5 Scheduling overhead observed in scenario $Q_{np}=0.17$ $Q_{nn}=0.14$	64
Table 4-6 Number of Collisions Observed From Scenario $Q_{np}=0.17$ $Q_{nn}=0.14$	65
Table 4-7 Primary capacity degradation when Q_{np} increase from 0.17, 0.40 to 0.59.....	70
Table 4-8 primary capacity degradation when Q_{nn} increase from 0.0, 0.079 to 0.14	72
Table 4-9 Performance Degradation after Increasing AET_s for 5 PE.....	75
Table 4-10 Performance Degradation after Increasing AET_s for 10 PE.....	75
Table 4-11 Scheduling Overhead Before and After Increasing AET_s	76
Table 4-12 Number of Rollbacks Before and After Increasing AET_s	77
Table 4-13 The impact of frequency of batch switches	78
Table 4-14 Sync. idle after / before reducing the frequency of batch switches.....	78
Table 4-15 Primary Capacity With No Serial Idle Option vs. With Serial Idle Option...81	
Table 4-16 Primary Capacity: No Sync. Idle option vs. Sync. idle option.....	83

Chapter 1

Introduction

Multiprocessor system architectures with several processing units provide a method to increase available processing power by allowing more than one CPU per system. For decades the methods of multiprocessing have offered a significant source for performance improvement to a wide range of scientific and engineering applications. In this research we are concerned with a multiprocessor based call control system in a telephone switch. The focus of our research is on the issue of shared memory contention, which has been proven to be the main limitation towards reaching the targeted capacity requirements. General techniques associated with the resolution of shared memory contention are discussed in this thesis. The investigation of three scheduling algorithms, which are devised to address the shared memory contention issue encountered in the multiprocessor based call control system, is presented in the thesis.

1.1 Multiprocessing

Multiprocessing systems can be implemented as either distributed or shared memory architecture depending on the way memory is connected to the processors. Figure 1-1 and Figure 1-2 illustrate the two architectures for comparison.

Distributed memory multiprocessor systems, often called multi-computers, are characterized by the

Figure 1-1 Distributed memory multiprocessor system

factor that processors in these types of systems can access only their own local memory modules, see Figure 1-1. Communication between processing nodes is required in order to coordinate activities and move data. Data can be exchanged, but not shared. In distributed memory multiprocessor system processors do not share a common address space, therefore they do not have to contend with each other to get to data. Because each node is a complete computer system by itself, a distributed memory multiprocessor system can be built from off-the-shelf components. From the point of view of machine construction, such architecture is scalable and cost-effective. The only practical limit to the performance improvement achieved through adding processing nodes is dictated by the topology used to connect the nodes. In fact the topology of the interconnection network, e.g. rings, arrays, cubes etc., greatly impacts the performance of any distributed memory multiprocessor system. Besides the complexity of the interconnections, data

management across distributed memory can be challenging because replicated data is difficult to synchronize and remote data access is inefficient. As a result, data is usually partitioned and owned by specific processors, and processors become dedicated to certain functions, making the system vulnerable to node failure. Communication through message-passing presents another problem: the explicit invocation of send/receive primitives required when coding. Such a programming model imposes a significant burden on the application programmer because application software must be designed or redesigned for optimal data distribution across the system. Example systems include the Intel Paragon and Teraflop systems, nCUBE3 from nCUBE, and Cray T3D and T3E [Vandersteen96].

Figure 1-2 Shared memory multiprocessor systems

On the other hand shared memory systems are built from multiple high performance microprocessors that logically share the same memory space (see Figure 1-2). Construction of

such a machine is far more complex than a distributed memory machine. Its efficient and safe use also requires the introduction of additional hardware mechanisms for resolving memory access contention. Once constructed, however, a shared memory system offers great advantages from the programmers' point of view. The system-wide unified address space that is equally accessible to all processors in the system enables a general and simple programming model which eases the programming burden. Furthermore direct access to the shared memory allows a medium-grained approach to multiprocessing. Such architecture is readily adaptable to a single processor system that uses time sliced multitasking for achieving apparent concurrency. However, as the machine grows, shared memory multiprocessors typically suffer from increased contention and longer latencies in the accessing of shared memory, resulting in a somewhat lower peak performance and limited scalability compared to distributed system. A detailed discussion of shared memory multiprocessors is provided in [Lenoski95]. Example systems include IBM PR3, HP-Convex Exemplar, and Sequent NUMA-Q 2000 [Almasi94].

Increasingly the distinction between the shared memory and distributed memory architectures is becoming blurred with the adaptation of hybrid architectures. Distributed shared memory scheme, which uses a physically distributed memory architecture but a logically shared address space, tries to combine the best characteristics of these two architectures. Most of the distributed shared memory multiprocessing systems are built for massive parallel computing.

Shared memory architectures vary from symmetric multiprocessor to asymmetric multiprocessor, depending on the software approaches used. Figure 1-3 illustrates a symmetric multiprocessing system. In symmetric multiprocessing systems, all processors are equal in that all of them can run

operating system code as well as application code. Any work that needs to be done in the system can be performed by any of the processors. All processors share the kernel image in memory and

Figure 1-3 Symmetric multiprocessing system

the same queues of processes. Works can be done in parallel. Especially, I/O requests and device driver interrupts can be processed simultaneously on some processors while application processes are running on other processors. The advantages of symmetric multiprocessing are characterized as follows: fast response time, uniform memory access, load balancing, no single point of failure, high availability, low interconnect network traffic, and the compatible program model with a single CPU multitasking system – an important feature that enables the portability of legacy software.

Shared memory multiprocessor with a symmetric programming model is the type of system we are

concerned with in our research.

1.2 Shared Memory Contention

Multiprocessing promises significant gains in performance, but it also brings problems that make an efficient multiprocessor system difficult to realize. One of the problems is memory contention, a limiting factor that is encountered in any shared memory multiprocessor. Memory contention results from multiple processes executed on different processors attempt to access to a shared resource simultaneously. In [Nanda96] memory contention is categorized as contention for memory path, contention for memory module, contention for memory location. In this thesis we are concerned with contention for memory locations, called shared data contention.

Memory contention results from multiple processes executing on different processors accessing the same piece of data simultaneously. This contentious access may result in an inconsistent state of the variable. Figure 1-4 illustrates a possibly incorrect outcome from concurrent data accesses. Both process A (running on processor1) and process B (running on processor2) tries to increment the value stored in variable x. Depending on the timing they may leave an incorrect final value of 6 in x. As a rule to preserve the integrity of data, no two processes should be allowed to execute in parallel if they access the same memory location.

Memory contention first arose in single processor based multitasking systems, where multiple processes were interleaved in execution to achieve apparent concurrency. A scenario of such contention in single processor based multitasking system can occur when two processes share some

pieces of data. Although the CPU executes one process at a time, time slice timer can force the

Figure 1-4 Memory contention

process to release the CPU while it is in the middle of manipulating the data, leaving the data in a potential inconsistent state if other process happens to manipulate this data as well. Memory contention in single processor based multitasking systems has been well studied and the effective mutual exclusion mechanisms, such as semaphore [Silberschatz99] and setting process unpreemptable until coming out of critical section [Nortel98], have been devised to guarantee exclusive access to the shared data. In multiprocessor systems, however, nonpreemptability alone

no longer guarantees that the data is not touched by another process executing on another processor (as in the case captured in Figure 1-4). Additional mechanisms that prevent multiple processors' accessing the same memory locations concurrently have been devised.

[Catanzaro92] and [Motorola94] have implemented a 'lock' scheme in hardware. In these systems, the data within caches is most often organized into blocks of data, called cache lines. The shared memory space is also divided into lines of the same granularity as the cache line (see Figure 1-5). Each running process is dynamically assigned a unique identifier by the operating system before it is scheduled in. The identifier is invalidated after it is scheduled out. This identifier is used to tag the memory lines if the process ever accesses the shared memory. A process must lock and tag the memory line before accessing it. If the memory line is already owned by another process, the memory management unit (MMU) issues a memory access exception to signal the operating system that one processor has collided with another processor over this memory line. Locking schemes rely on the operating system to take necessary actions to resolve the conflicting access of shared memory and to coordinate processes so that only one process accesses the memory line. The unlock procedure following the completion of process execution includes instructions for making any modification in memory visible to the whole system and for releasing the lock. Once the lock is released, the memory line is available to other processes. The Lock algorithm implemented in hardware and the collision resolution logic implemented in the operating system hide the complexity of handling memory contention from application process. The application processes may not even be aware of the memory access collision occurring under their feet. The locking scheme ensures the data residing in the memory line is guarded from being touched by more than one process concurrently. Thus contentious application processes are allowed to run in

parallel speculatively without worrying about corrupting memory. If two contentious processes happen not to read from or not to write into any of their shared pieces of data concurrently, they can complete their slice of execution. Otherwise, the operating system serializes the memory access enforced by memory line 'lock'. A variety of lock schemes have been studied and are in use in practice [Thomasian93][Tay85]. Examples include test-and-test-and set locks, exponential backoff, and clever algorithms for spin locks and barriers etc. In addition to lock schemes, semaphore mechanism is still an effective way in multiprocessor systems to protect shared variables especially large chunk of shared data structures such as queues, pools, as described in [Benniger98].

Figure 1-5 Cache line and memory line

Locks and semaphores preserve the integrity of data by serializing accesses to the shared data. Too much serialization, however, would compromise the parallel nature of multiprocessing.

Furthermore the performance penalty comes from not only serializing memory location access, but also the extra work required in the operating system for resolving any collision possibly involved. [Drweiga96.1] investigated the impact of memory access contention on a shared memory multiprocessor based system in a telephone switch, and developed an analytical model that can estimate the contention probability between contentious processes when given the sets of addresses accessed by these processes [Drweiga97]. To reduce the adverse effects of the contention, there have been proposals intending to prevent or reduce conflicting access to the shared data. In [Wilson96.1] the object-oriented programming with parallelism paradigm was introduced as one of the multi-processing approaches to the software design for a product developed in Nortel. In this document the memory access contention problem is described in terms of data domain overlaps. A data domain is a collection of related data and functionality. A system then can be divided up into distinct domains. Each domain provides a function or service. Data is often shared between domains. This sharing between domains creates domain overlap and is the cause of memory access contention. Therefore the problem of designing contention free software involves the removal of the overlap between domains. Since the data-oriented approach allows data variables and process functions to be grouped together and treated as a single self-contained object, data structures are contained within objects and hidden from the external view. Encapsulated data is only accessible via the object's interface procedures or methods. Data encapsulation exemplifies the sound data management practices that application users need for controlling how and when data is manipulated. Data contention is more easily controlled once the problem data is properly defined and partitioned. In addition to the benefit of data management, OO offers better program comprehension, resilience to change and code reuse [Wilson96.2].

False sharing contributes to contention by causing unnecessary collision. False sharing refers to data structures belonging to two different processes but residing within the same memory line although they do not have any shared data subset. This sharing of the same memory lines will cause potential contention over memory access. See Figure 1-6.

Figure 1-6 Memory access contention

A processor accesses a whole memory line at a time, loading it into one of the processor data caches as exact copies of the data in external memory. So when a process accesses a data field, it will lock all data items residing in the memory line. This is not always a problem because adjacent data items tend to be related and used by the same process. However, there is a possibility that some of the data items may be accessed by a parallel process on another processor. Research results

[Torrie96][Lilja94] have led to a better understanding of some problematic patterns, such as small inner parallel loops, typically occurring in compiler-parallelized applications, and coming up with standard methods for eliminating false sharing and minimizing true sharing.

Processor scheduling has been studied extensively in shared memory multiprocessor architecture, mostly for improving response time of scientific program [Feitelson94][Feitelson96] and not for controlling memory contention. [Majumdar99.1] proposed different scheduling policies which are investigated in this thesis through a trace driven simulator. The key feature of these algorithms is preventing contentious processes from concurrent execution. In addition, controlling shared data contention through process scheduling hide the complexity of parallel programming from application process level. This may help spread the use of the multiprocessing.

1.3 Motivation for the Thesis

A call control system in a telephone switch is evolving to become a multiprocessor based system to provide higher processing capacity and scalability. The proposed architecture consists of one central shared memory system, accessed by more than one independent processor. Each processor has a local memory containing its own copy of the program store, but all data resides in the shared memory. The main problem the project has is shared data contention. Shared data contention problem in this project is particularly challenging due to the large amount of global variables used in legacy software, and the requirement of porting legacy software written for single processor multitasking system to multiprocessor system without changing a vast amount of code.

As described in section 1.2, concurrent accesses by processes executing on different processors may cause inconsistent shared memory. Locking schemes preserve the shared memory's integrity but serialize the data access between processes. Furthermore the resolutions involved in operating system for resolving the conflicting accesses result in significant performance degradation on system. For highly used data, the best way of designing multiprocessing software is to construct data domains that are disjointed. However, disjointed data domains are not always possible, especially for the transition of the legacy software written for a single processor multitasking system to a multiprocessor system without rewriting vast amount of the source code. In this case, controlling the sharing of data in shared memory by not scheduling contentious processes to run concurrently becomes an important alternative used in conjunction with limited mutual data domain disjointing.

This thesis evaluates three scheduling algorithms devised for controlling the sharing of data between processes and identifies the potential capacity gain of the multiprocessor based call control system through various scheduling algorithms.

1.4 Goals of the Thesis

The goals of this thesis are:

- 1) to evaluate the performance of three scheduling algorithms under various contentious scenarios;
- 2) to identify the potential capacity gain through various scheduling algorithms;
- 3) to study the scalability for each of the scheduling algorithm under different data contention levels.

A trace driven simulator was built for carrying out the investigation.

1.5 Outline of Thesis

This thesis is organized into five chapters. Chapter 2 provides a brief background information of the multiprocessor based call control system with which our research is concerned, describes the characteristics of the shared data contention experienced in this system and presents the scheduling algorithms to be evaluated in this thesis. A key feature of these algorithms is controlling the sharing of data by selecting less contentious processes to execute simultaneously. Chapter 3 describes the simulation model, the system workload and the validation of the simulation model. Chapter 4 evaluates the relative performance of three scheduling algorithms by examining the results from experiments. Chapter 5 provides the conclusions, the contributions of this research, and the recommendations for future research.

Chapter 2

Multiprocessor Based Call Control System

This chapter is organized into five sections. Section 2.1 provides brief background information on the evolution of the DMS SuperNode built by Nortel. Section 2.2 describes the architecture of the multiprocessor based call control system. Section 2.3 presents the basic concepts for describing the multiprocessor based call control system. Section 2.4 describes characteristic of shared data contention experienced in DMS switches. Section 2.5 presents the scheduling algorithms to be evaluated in this study.

2.1 Motivation for Moving to Multiprocessor System

A telephone switch consists of three components: the call control module, the line interfaces and the switching fabric. In this thesis our primary focus is on the call control system. The call control module, called the call control system in this thesis, is the control component that performs call processing and switch maintenance functions [Nortel94]. In the DMS SuperNode the call control system is a single processor based multitasking system. To handle constant demand for increased call volumes and advanced call services, DMS SuperNode, developed in Nortel during 1970s, is challenged to meet current technology requirement. A project to construct a scalable shared memory multiprocessor based call control system is chosen to push the SuperNode core capacity to a new milestone. Several reasons contribute to this technical decision [Benniger98]:

1. Although uniprocessor performance continues to increase roughly 50 percent each year, DMS is unable to realize the full stated performance gain of these new processors, primarily because call processing is exceptionally memory access intensive. Given the current memory-access technology faster processors tend to spend more clock cycles waiting for the slow memory response. Besides, the legacy DMS SuperNode software running on a multitasking platform has inherent parallelism. Thus multiprocessing is a natural and effective way towards improving their capacity. In addition to computing speed, capacity evolution involves many engineering aspects of the switch such as reliability in terms of fault tolerance, scalability in terms of number of processors and also memory sizes. To realize the targeted capacity requirements, multiprocessing is an available and cost-effective alternative to achieve the capacity objectives DMS is facing.
- 2 As stated in [Stenstrom97], progress in shared memory multiprocessing research has led shared memory multiprocessor to be a key technology for industrial projects. The mutual exclusion provided in shared memory hardware and the shared data contention resolved in the operating system make the transition from conventional programming to parallel programming almost transparent to application programmers.
- 3 The system-wide unified address space in shared memory multiprocessor offers similar programming model used in single processor based multitasking platform. This enables a huge amount of DMS legacy software to be ported to the new multiprocessor platform with only reasonable amount of software re-writing. Compared to distributed memory multiprocessor programming model, the shared memory solution is thus a financially viable alternative for DMS evolution.

- 4 A shared memory multiprocessor enables symmetrical processing model. A symmetrical multiprocessor brings a number of benefits: constantly low latency in accessing data and communication due to physically shared memory; efficient data management due to no data movement across processing nodes; automatic load balancing (i.e. the ability of transparently distributing tasks across processors) due to its self-scheduling characteristic; resilient to processor outage due to the equal capability of all processors, the interrupted work on faulty processor can be taken over by other survival processors.

2.2 Architecture of the Multiprocessor Based Call Control System

The proposed architecture for call control system of DMS is a small scale shared memory multiprocessor [Baker95]. It is designed to have three capabilities: capacity scalability, reliability, and compatibility with DMS legacy software. Figure 2-1 presents the architecture of the shared memory multiprocessor based call control system. The system consists of a central shared memory, a number of processing Elements (PE0...PE_n), and an interconnection network between the shared memory and all PEs.

The shared memory is a number of memory cards (M0...M_n) that can be plugged in. It is built in duplex mode (i.e. two copies of memory cards storing identical information) for fault tolerance. DATA Store (DS) is in the shared memory, which is fully addressable by every PE. Program Store (PS) is stored in shared memory as the master copy of PS as well as every PE's local memory (LM0...LM_n). All copies of PS are identical, and a PE executes code using local PS. When a PE recovers from fault, it copies the PS from master PS. The entire shared memory is organized into

memory lines with the same granule as cache line. Read and write accesses to shared memory by the PE normally manipulate an entire memory line. The shared memory supports a concept of

Figure 2-1 The architecture of multiprocessor based call control system

dynamic memory ownership, and the unit of memory ownership is a memory line. Memory ownership provides the mutual exclusive access to a memory line in hardware, which is required to preserve shared memory consistency. To claim the memory ownership, each process that is executing on a PE is assigned a unique identifier by the operating system. Whenever reading from or writing into a memory location, this identifier is passed to the hardware to tag the entire memory line in shared memory as owned by the process. Any other process attempting to access a memory location that is already owned is blocked from processing. One of the contending processes must abort what it was doing and try again later. The outcome is that those processes requiring the same

shared memory locations may be serialized by the system if executing concurrently, whereas independent processes are allowed to execute in parallel.

A Processing Element is a super scalar RISC CPU. It includes instruction caches and data caches, each 32 bytes long, to improve the average response time for memory accesses. Each PE has a certain amount of local synchronous dynamic random access memory, which gives fast access to program store and certain performance sensitive data. A PE is a simplex device, but the system employs one or more 'spare' (extra) PEs for redundancy. PEs are symmetric, meaning all PEs are functionally equivalent, and all PEs share the software load equally. Shared memory multiprocessors with symmetric programming model have proven their scalability in some other industry applications [Sequent96]. To increase the system capacity all you need to do is add one or more PE cards as well as memory cards into the system. Since each PE is equally capable and responsible for managing the flow of work through the system, the newly plugged in cards will go in the system automatically once activated. The reliability is provided by the extra PEs and the PEs' symmetric characteristic. If a PE takes a fault, or if routing service is down, there is no need to de-load or re-distribute traffic, the system will take over any failed component, balance the load, transparently distribute work across the surviving PEs, and will remain alive with graceful degraded service capacity [Benniger98].

The Interconnect is an internal call control system network that provides access to the shared memory for PEs. It comprises a matrix of unidirectional point-to-multipoint links. Every PE or memory card has a single transmit port that other cards listen to, but has multiple receive ports, one from each card that it is receiving from. This matrix of links relieves the potential congestion that a single bus or a multistage interconnection network would suffer in other shared memory

multiprocessors [Dandamudi97]. The interconnection network can grow with the number of PE and memory cards installed. Reliability is provided by the dedicated link for each card. A link failure would affect only the one card thus isolating the single point fault effectively.

2.3 Basic Concepts

This section introduces some basic concepts that are necessary to describe the multiprocessor based call control system.

Primary process and secondary process: All processes in the call control system can be classified into two broad domains: primary and secondary. The functionality of primary processes is call processing related, such as call setup, call disconnection, call forward or billing record etc., whereas the functionality of secondary processes are system maintenance related, such as port monitoring or resource auditing etc. Based on their functionality, it is possible to divide the secondary processes into additional classes but is not necessary in the content of the high level investigation presented in this thesis.

The assumptions made in this thesis about the CPU shares given to different process domains are presented. On legacy call control system the primary domain is typically allocated 75% - 80% of CPU resources and the secondary domain is allocated 20% - 25% of CPU resources. On scalable multiprocessor based call control systems with less than 5 PEs, the primary domain is given 80% of the total CPU resources, and the secondary domain is given 20% of the total CPU resources; for large system with more than 5 PEs the secondary domain is given $1/N$ (N is the number of PEs in

the system) of the total CPU, the primary domain is given $(1 - 1/N)$ of the total CPU capacity [Corroll99].

Transaction: A process performs one or a number of transactions during one time slice. Each transaction completes one piece of relatively independent work. Often a transaction corresponds to an unpreemptable execution that runs to completion on a switch to ensure memory consistency.

Block: Every process that is executing on a PE is assigned a unique ownership identifier (oid), and this oid is used as a tag to claim ownership of all locations in shared memory that are read or written by the process. Any other process attempting to access a memory location that is already owned is *blocked* from further processing. The process that is causing other processes to be blocked is *blocking* the other contending processes from execution. Block is the built-in mutual exclusion mechanism provided in shared memory to preserve the integrity of shared memory in the presence of multiple PEs.

Commit and Rollback: There are two copies of every memory line in shared memory: the commit copy and the active copy. The commit copy is the official copy that records the most recent changes in memory. The active copy is where an executing process temporarily stores the changes until completion. During execution a process writes into active copy. Upon completion the operating system commits the process by making all updates the process made visible to the whole system, i.e. exchange the role of the two copies. The active copy with changes achieved by the committing process becomes the new commit copy, the old commit copy becomes the new active copy to be updated on. If a process is blocked from further execution, the blocked PE is notified by

the shared memory of the collision. Then the operating system is invoked to resolve the collision. The operating system resolves the collision by rolling one of the contending processes back to the start point of this transaction's execution. Rollback is done by not committing the active copy of memory updated by the rolled back process, effectively erasing any trace of the process' execution. If a process is rolled back, the operating system puts it back to ready queue for later rescheduling. Upon commit or rollback the ownership tag is released so that the memory lines are available to other processes.

Commit and rollback are two primitives done by the operating system. When a process commits, the operating system flushes all cached information back to the active copy of memory, exchanges the two memory copies, and performs a context switch to schedule another process in, if necessary. When a process is rolled back, the operating system invalidates the data cache, puts the process back into ready queue, performs a context switch to schedule another process in. Rollbacks expend work to preserve data consistency. The processing time spent on the rolled back process (including time for scheduling in the process, time for resolving the collision, time for executing the process) is wasted, and is recorded as rollback overhead.

2.4 Memory Contention in Multiprocessor Based Call Control System

The DMS SuperNode version of call control system is single CPU multitasking system that employs a great amount of global data such as pools, tables, and queues. In single CPU multitasking system, communication between processes via global data is convenient and efficient.

However, when ported to a multiprocessor with a shared memory model, the global data causes serious contention problems. For example, primary processes may share a work queue. If several primary processes execute in parallel on different PEs, they may contend for accessing the same work queue. A secondary process may occasionally audit the work queue shared by primary processes. If the secondary process happens to execute in parallel with some primary processes on different PEs, they may collide on accessing the work queue or a queue element. Another example is concerned with two secondary processes sharing a table. If they execute in parallel on different PEs, they also may collide on accessing the shared table. The sharing of global data can be described by the concept of data domain. A *Data domain* is a collection of related data and functionality. The whole system can be divided up into distinct domains. Each domain provides a function or service. Data is often shared between domains. This sharing between domains can create domain overlap and is the cause of shared data contention [Wilson96.2].

The shared data contention in the multiprocessor based call control system has been studied by Nortel since the beginning of the project. [Dreweiga96.2] observed some key characteristics of the shared data contention between primary processes, and analyzed the impact of primary process data contention on the system throughput. [Dreweiga96.1] observed the shared data contention between primary process and secondary process. An analytic model for computing the level of contention between primary process and secondary process is described in [Dreweiga97]. To describe the contention between domains, a blocking parameter Q was introduced. This blocking parameter, Q , is the probability that two processes collide on a data access, if they are executed in parallel. The probability of contention between two primary processes is denoted as Q_{pp} ; the probability of contention between a primary process and a secondary process is denoted by Q_{np} ; the probability

of contention between secondary processes is denoted by Q_{nn} . Typical value of data contention parameters expected in DMS switches are $Q_{pp} = 0.036$, $Q_{np} = 0.17$ to 0.59 , and $Q_{nn} = 0.079$ to 0.14 [Kolski97][Carroll99].

The Q_{pp} of 0.036 in the current multiprocessing based call control system is very small. The reason is that the overlapped data between primary processes has been disjointed through code re-engineering. The primary domain gets 80% of the machine, so they are the high runners in the system. To reach the targeted capacity objective they must be blocking free most of the time. Disjointing overlapped data domain is the common way of resolving shared data contention. The problem with it is that disjointing data domain is not always feasible such as in the case of porting DMS source code onto the multiprocessing platform. In this case controlling sharing of data to avoid contention becomes interesting.

2.5 Scheduling Algorithms

The idea of controlling shared data contention through process scheduling is to manage concurrent process execution in such a way that the contentious processes have less chances to run in parallel, thus reducing the number of shared data access collisions. A high level study based on an analytic model is presented in [Majumdar99.2]. Because the contention among primary processes has been very well studied and has been reduced to be small enough to run in parallel without much blocking, our focus is on the contention between primary process and secondary process and among secondary processes. A trace driven simulator is therefore developed to investigate the scheduling strategies.

The scheduling policies investigated in this thesis are briefly described in the following.

Serial algorithm:

All secondary processes are run serially. The scheduler does not allow any secondary process to execute in parallel with another secondary processes, though they are allowed to execute in parallel with primary processes.

Wait Option:

Any PE ready to execute a secondary process must first wait for the completion of a secondary process in another PE. The scheduler may not substitute a primary process because of the concerns about the consequential distortion in processing time allocation across process classes [Benniger98].

No Wait Option:

When a PE is ready to execute a secondary process but can not do so because another PE has not completed a secondary process, the scheduler may substitute a primary process and take special care to not disturb engineered processing time allocation across process classes.

The serial scheduling algorithm is developed in the early project phase when the contention behavior of secondary processes was not clear. To minimize the impact of secondary processes, the execution of secondary process is serialized. There is no collision between secondary processes because there is only one secondary process executing at a time. There is controlled collision between primary process and secondary process because only one secondary process can interfere

with primary processes. A serial algorithm with a wait option results in a scheduling overhead that affects processor utilization. This scheduling overhead is called the serial idle. The impact of serial idle on system capacity is discussed in Chapter 4 along with experiment results. A serial algorithm with no wait option tries to eliminate the scheduling idle by skipping to a primary process. This may disturb the engineered processing time allocation across process domains. However the simulation results support the 'no wait option' from the point of view of obtaining capacity improvement. The serial algorithm is subjected to a constraint that the total execution time of secondary code must be within one PE worth of capacity. As the system grows, managing the total execution time of secondary code within one PE worth of capacity may be very difficult, causing the serial algorithm to fail to meet the system maintenance requirements.

Gang Scheduling:

All secondary processes form a gang while all primary processes form another gang. Scheduling policies similar to the gang scheduling algorithm discussed in this thesis are considered for improving the performance of scientific computations (see [Feitelson94] and [Feitelson96]). Members from the same gang run in parallel as a batch. All processors are devoted to the members of the running gang; no members from the other gang are allowed to run until the processing time allocated to the running gang is exhausted. Then the scheduler will make a gang switch to let the processor run a process from another gang. If no process can be run from the running gang, a PE will wait until all other PEs commit the transaction of the running gang. If a PE has finished its execution and the processing time allocated to the running gang is over, the PE must wait for the completion of all the processes from the running gang.

The gang scheduling algorithm assumes the blocking among secondary processes is small enough to allow secondary processes run in parallel. Under such an assumption, it expects a capacity gain from separating primary and secondary processes so that secondary processes will never contend with primary processes for memory access. However, this separation may delay processes that are responsible for events requiring immediate attention from being scheduled in time and the processes must wait for their batch to come. The gang scheduling algorithm results in a scheduling overhead that affects processor utilization. We name this scheduling overhead as “synchronization idle”. The impact of synchronization idle on system capacity will be discussed in chapter 4 with some experiment results. The synchronization idle can be reduced by adjusting the interval between batches. Increasing the interval would also increase process scheduling latency.

Hybrid Algorithm:

A hybrid algorithm uses an engineered mix of serial and Gang. The proportion of the secondary workload measured in processor seconds that is run under Gang and Serial is captured in a metric termed s:g. For example, s:g = 3:1 indicates that processes that account for 25% of the secondary workload are run under the Gang algorithm whereas processes that account for 75% of the secondary workload are run under the serial scheduling algorithm. Secondary processes that do not interact with one another but collide with primary processes are ideal candidates for forming a gang. Secondary processes that interact heavily with one another but do not collide with primary processes are the best choice for including in the serial group.

Note that the serial algorithm and the gang algorithm can be expressed as special cases of the

hybrid algorithm with $s:g = 1:0$ and $s:g = 0:1$ respectively. All these scheduling approaches must satisfy the rules of fair share scheduling [stallings98]. The primary domain and secondary domain must be guaranteed to receive a predetermined fraction of the available processing time. This prevents process starvation since each process domain is guaranteed a share of the processing resources.

Chapter 3

Design of the Simulation

This chapter describes the design and implementation of the simulation model. It consists of four sections. Section 3.1 describes the package used in our simulation. Section 3.2 describes the development of the simulator. The simulator captures the real time behavior of the system. Section 3.3 describes the development of the workload for our simulation model. The workload intends to provide the simulator an input that models the working environment of the real system. Section 3.4 describes the output of our simulation model, as well as the results of the model validation experiments.

3.1 The Simulation Package

The simulation package used to build our simulator is PROPHET (Proprietary Hierarchical Evaluation Tool) [Nortel96]. It is a library of C++ classes and functions developed at NORTEL as a design capture and performance analysis toolkit for multi-tasking/multi-processor systems. Like many other general purpose simulation packages, PROPHET is based on discrete event simulation and collects statistics that can be used to analyze or predict the performance of the system being modeled. As a proprietary simulation tool, however, PROPHET provides a unique modeling paradigm that can capture the applications, scheduling and hardware characteristics separately, thus facilitating the reconfiguration of the components of a simulation model. To develop a simulator, a C++ program is written to describe the real time behavior of the system to model, following the basic modeling paradigm of PROPHET.

PROPHET Modeling Paradigm:

PROPHET models a multi-processor computing system using three distinct layers – hardware, scheduling and applications. Each layer consists of a set of C++ base classes needed to create a model for that layer (see Figure 3-1). The application layer is a set of classes derived from base

Figure 3-1 PROPHET Modeling Paradigm and Modeling Classes

class *task*. Each derived task class defines its control logic, real-time delays and messaging of a part of the system being modeled. At run time a task instance is an independent executable image of the task. The task instances run concurrently and communicate with messages. The scheduling layer is a class derived from base class *scheduler* or *InstSched*. A derived scheduler class defines how contention for processors is resolved. Its job is to determine, according to the scheduling algorithm provided, which task will get the processor when several of them are ready to run on it. A scheduler specifies actions to perform when tasks request the processor, release it, or reach the end of their time slice. It has ready queues to store tasks waiting to be executed. For a symmetric

system, there is exactly one scheduler instance per processor. The hardware layer is a class derived from base class *processor* or *multiProcessor*. It has a relative speed and a scheduling algorithm.

Under PROPHET paradigm, each layer can be modeled separately. In our case, we are interested in comparing scheduling alternatives. We need to change scheduling algorithms from experiment to experiment but not to change the real time behavior of the call processing system nor the processors. PROPHET offers such modeling flexibility.

PROPHET Statistics Collection:

PROPHET facilitates the statistical measurement needed for simulation. It collects system statistics, such as processor utilization, message queues and ready queues, automatically. In addition, it provides users functions to collect user statistics. To do so, the programmer declares the user statistics first, then inserts statistic function calls wherever or whenever needed to the C++ program. For example, to get statistics on the AVERAGE WORK TIME for a primary process transaction, we define the following:

```
# include "prophet.h"
# include "ManualBM.h"

...
// declaration of user statistic
UserSampleStat AWT ("primay_transaction_AWT");
...
...
// in program where a primary transaction execution begins
transaction_start_time = clock();
...
...
// in program where the primary transaction execution ends
AWT.update(clock()-transaction_start_time);
...
```

PROPHET can compute average, variance, minimum, maximum, number of observation automatically. However, to get meaningful as well as correct statistics, users are required to ensure that the chosen statistics are appropriate for characterizing the simulation model, and that the value provided to update() function is correct.

3.2 The Simulator

3.2.1 Overview

Figure 3-2 is a conceptual view of the simulator. It consists of three components: Message generators, Message Queues and Multiprocessor Based Call Control System. Each component is written as C++ classes with variables and methods to describe its state and run time behavior. Of the three components, Multiprocessor Based Call Control System is the focus of our study. This simulator is an extension of another simulation program which models data contention among primary processes [Streibel97]. Our work extends the original simulator to include secondary processes and to include three scheduling algorithms.

The role of the message generators is generating inputs for the simulator only. They work in the following way: CallGenerator regularly generates a message, piggybacks the workload data for primary domain process on the message, and sends the message to Call Control System through primaryQ. These messages are queued up on primaryQ, then retrieved and processed by primary processes. The inter-arrival time for message is exponentially distributed with an average being adjusted as part of the input. Similarly are the actions of SerialGenerator and GangGeneraor.

Figure 3-2 The simulator

SerialGenerator sends messages containing the workload data for the serial domain process to Call Control System through SerialQ, serial domain processes then retrieve these messages and process them. GangGenerator sends messages containing the workload data for gang domain processes to Call Control System through GangQ. Gang domain processes then retrieve these messages and process them. Note that the three generators work independently. Although they all interact with Call Control System, they do it through different queues - PrimaryQ, SerialQ and GangQ. And each generator generates messages at its own interval of time, one would not affect another.

As in the case of the communication channels, the Message Queues are shared by message generators and Call Control System. The ordering discipline of these Queues is chosen to be FIFO; however class 'MessageQ' in PROPHET provides methods such as InsertQ(whichQ) to facilitate some Queue operations other than the default discipline – coming to the queue tail and going from queue head. This is important for modeling process rollback, in which case the message that contains workload data must be inserted back into the origination queue for later re-execution.

The Multi-processor Based Call Control System represents the call control module of the real switch. Following PROPHET modeling paradigm, the concerned components of Call Control System are modeled by three layers of C++ program - Task layer, Scheduling layer and Hardware layer. Each layer captures the states and real time behavior of the component to model.

1. The task layer simulates the process memory access activities of the Call Control System. In our model, we classify all processes in Call Control System into two categories: primary processes and secondary processes. Secondary processes are further divided into two domains: serial process and gang process. In total there are three process domains - primary process domain, serial process domain and gang process domain. To simplify our simulator, the modeling of resolving memory access collision between processes is placed in task layer, even though it takes place in the operating system on a real switch. The methods in task layer simulate the shared memory access behavior of processes executing in parallel on multiple PEs, i.e. how a process owns memory lines, how long a process holds those memory lines, how the system rolls one process back when two processes collide on accessing a same memory line, how the system keeps track of the processing time wasted on the rolled back process and the extra context switch spent for the operating system

processing a rollback when rollback occurs and how the system commits the successful execution once a process completes its execution etc.

2. The scheduling layer program models the three scheduling algorithms described in section 2.3.

To make the simulator configurable, three alternatives are implemented in one class MPScheduler with a switch (config.alternatives) statement to differentiate them explicitly during experiment. The simulation configuration variable 'alternatives' is passed as a parameter to the configuration file provided by PROPHET experiment engine. Section 3.2.2 presents detailed assumptions and implementations of the scheduling layer.

3. The hardware layer simulates processors running in parallel. For modeling the symmetric characteristic of the Multiprocessor Based Call Control System, the implementation of class 'PROCESSOR' allows one instance of processor being managed by one instance of "MPSCHEDULER" so that every processor is capable of self-scheduling. That is, every processor can run scheduler code as well as task level code, having the full functions of a CPU in single processor based multitasking system.

3.2.2 The Scheduler

The scheduling layer program captures the most important characteristics of the scheduler being evaluated in our study. These characteristics are symmetric, multitasking and memory access contention control.

Symmetric can be interpreted as 'all processors function the same' in the system. There is no master processor, no central scheduler for distributing processes among processors. A processor is not dedicated to any process or any process domain. A ready process can be executing on processor whichever is available first and the outcome of execution is the same no matter which processor the process is on. To model the symmetric system, every instance of 'processor' is managed by an instance of 'MPScheduler', and all instances of 'MPScheduler' share the system wide ready queues. Ready queues are a set of linked lists that hold all processes of the system. When a running process voluntarily releases the processor or is rolled back, the instance of scheduler on that processor is triggered to perform scheduling operation. Scheduling operation includes a bunch of activities, such as handle process suspension request, place ready process on ready queues, and select a process to execute from the ready queues etc. Being shared variables, the ready queues can be accessed by all processors executing scheduler code. However, although multiple instances of the scheduler can be invoked concurrently on multiple processors, the access to the ready queues is serialized under a semaphore to prevent multiple scheduler instances from selecting the same process concurrently.

Multitasking is inherited from the single processor multitasking based call control system. In a real system, processes are time sliced to utilize the processors. A process is allowed to run for a number of time slices without being scheduled out. The system treats every process with equal priority, no process can preempt another, thus every process is given chance to finish its work. To respond to events in time and to utilize the processors in accordance with system requirements, a scheduler template is employed to engineer the distribution of available processing time across all processes. Under this scheduler template, processes are bundled into groups based on their

functionality. Processes within the same group form a ordered list. Every group is guaranteed a certain processing time percentage at capacity, called share. Process groups are not prioritized, but the shares do differ from process group to process group to weight their contribution to the quality of service that the switch provides. When selecting a process from many ready processes, the scheduler traverses the process groups in a “round-robin” fashion. Usually, the process at the head of the list of the group gets selected first. Upon being scheduled out, the processing time consumed on the running process is recorded against the process group, and the process is placed back to the tail of the list of the group. If the share in the current group is used up, the scheduler moves to the next process group. The above scheduling process repeats forever.

In the simulator, to focus on studying memory access contention, a process is not timesliced but “transaction divided”. A process is transaction divided in the sense that a process is scheduled out after every transaction. The only activity of a process within one transaction is accessing a set of memory locations. The input of the simulator provides one set of memory locations per transaction, so different transactions access different sets of memory locations, and the time taken for finishing a transaction varies from transaction to transaction. Upon completing a transaction or being rolled back, the running process releases the processor, the instance of the MPScheduler on that processor then is triggered to handle the release of processor and to select next ready process to run. The scheduler instance selects the process from process domains. As mentioned previously, the simulator groups all processes into three process domains - primary process domain, serial process domain, and gang process domain. Each domain is allocated a certain share of CPU resource. When scheduled out, the processing time spent on the running process is recorded against the process domain. If the share in the current domain is used up, the scheduler instance

moves to the next process domain. In current single processor based call control system, primary processes get roughly 80% of the total share, while secondary processes get 20% of the total share. In the proposed multiprocessor based call control system, the share allocation for primary processes and secondary processes may be re-engineered to give more portion of processing power to primary processes. Therefore our simulator has made the following assumptions on shares:

- for small scale system with no more than 5 processors
the primary process domain is given 80% of the total share; and
the secondary process domain is given 20% of the total share.
- for large scale system with more than 5 processors
the secondary process domain is given $1/(\text{num_PEs})$ of the total share; and
the primary process domain is given $1-1/(\text{num_PEs})$ of the total share.
Note that num_PEs means the total number of processors in the system
- share allocated to secondary process domain is divided by the serial and the gang domains. The ratios used in the simulator are serial : gang = 1:0, 3:1, 1:1, 0:1.
Note that 1:0 case corresponds with the serial scheduling algorithm, 0:1 case corresponds with the gang scheduling algorithm, and 3:1 and 1:1 cases correspond with the hybrid scheduling algorithm.

Table 3-1 is an example of the share allocation in our simulator.

Memory access contention control is the objective of our scheduling algorithm study. As described in Chapter 2, the investigation on the memory access contention behavior of Multiprocessor Based

Table 3-1 example of share allocation

num. of PEs in system	3	5	6	7	8	9	10
share for primary	80%	80%	83%	86%	87%	89%	90%
share for secondary	20%	20%	17%	14%	13%	11%	10%
share for serial with 1:1 ratio	10%	10%	8.5%	7%	6.5%	5.5%	5%
share for gang with 1:1 ratio	10%	10%	8.5%	7%	6.5%	5.5%	5%

Call Control System indicates that, the blocking between a primary process and other primary processes has been reduced to be small enough for them to be executed in parallel [Drweiga96.1], the blocking between a secondary process and another secondary process varies from pair to pair [Benniger98], and the blocking between a primary process and a secondary process is large if executing in parallel [Drweiga96.2]. Motivated by the above blocking assumptions, scheduling algorithms are proposed to manage the possibly contentious processes from executing in parallel. Serial scheduling algorithm serializes the execution of secondary processes to reduce the blocking between primary process and secondary process, and eliminate the blocking between secondary process and secondary process. The gang algorithm totally separates the secondary processes from primary processes to eliminate the blocking between primary processes and secondary processes. The hybrid algorithm serializes a portion of the secondary processes and separates the rest portion of the secondary processes from the primary processes and from the first portion of secondary processes. To model the above scheduling algorithms, a number of implementations have been done to build the scheduler.

Process domain selection. For simplicity, our simulator does not model the scheduler template mechanism used in a real scheduler, it can not traverse the process domains like the way the real scheduler does for scanning a suitable ready process. Instead, a random number generator is used in our simulator to help the serial process domain selection. When a instance of MPScheduler is invoked and if the MPScheduler is not searching from gang process domain, the random number generator would point to a serial process domain with a probability of $\text{serial_share}/(\text{serial_share}+\text{primary_share})$. Here serial_share means the share allocated to the serial process domain, and primary_share means the share allocated to the primary process domain. For example, if serial_share = 0.2 and primary_share = 0.8, the probability of the scheduler pointing to the serial domain is 0.2. Once the scheduler decides to select a process from serial domain, it de-queues the process at the head of serial domain ready queue and execute it. Otherwise, the scheduler would de-queue the process at the head of the primary domain ready queue and execute it. If the scheduler finds out both the share of the serial domain and the share of the primary domain have been used up, it then points to the gang domain and de-queues the process at the head of gang domain ready queue and executes it. Later if the share of gang domain has also been used up, the scheduler would go back to the random number generator to start another round of process domain selections. The order of process domain searching is fixed, it always starts from the random number generator.

Serial process serialization. Serializing serial processes requires processors that are about to run serial domain processes to suspend themselves if there is already a running serial process in the system. This ensures that only one serial domain process is running in the system at a time. The suspended processors cannot do anything but wait, resulting in a type of scheduling overhead

called serialization idle. Serialization idle is an overhead associated with the serial scheduling algorithm that lowers the performance of the system. It is therefore one of the important factors to be considered in our simulation model.

The suspension of the processor is implemented by queuing those processors that are about to run serial processes on a FIFO message queue – `serialization_token_Q`. A processor must have the serialization token to run a serial domain process. If the serialization token is not available, it has to wait on `serialization_token_Q`. The processor that is holding the serialization token is responsible for freeing up the processor at the head of `serialization_token_Q`, and passing the serialization token to it upon scheduling out the serial process, regardless of commit or rollback. The processor waiting at the head of `serialization_token_Q` can then start executing the serial process.

The statistics, such as average waiting time of all processors, i.e. serialization idle, and average length of the `serialization_token_Q`, are automatically collected by PROPHET experiment engine.

Processor synchronization. Separating gang domain processes from both primary processes and serial process requires processors suspend themselves if the scheduler instances are about to leave the current process domain(s). The suspended processors can not do anything until all processes from the current process domain(s) finish their executions, then all scheduler instances move to the next process domain(s). The synchronization among all processors in the system results in a type of scheduling overhead called synchronization idle. Like serialization idle, synchronization idle is an overhead associated with gang scheduling algorithm that lowers the performance of the system. It is therefore one of the important factors to be considered in our simulation model also.

The processor synchronization is implemented by scheduler instances' spin-locking on the synchronization flag when it is set. The synchronization flag is set by the instance of the scheduler that first detects the share exhaustion of current process domain, and is cleared by the last instance of scheduler that leaves the current process domain(s). By synchronizing all scheduler instances under the synchronization flag, no process from primary domain or from serial domain could execute when processes from gang domain start executing. On the other hand, when processes from the serial domain or the primary domain start executing, no process from the gang domain could remain running.

Unlike serialization idle, the synchronization idle can only be collected by defining a set of user statistics.

3.3 The workload

This section describes the inputs of the simulator. There are two types of inputs, the traces and PROPHET.config file. The traces provide memory access contention scenarios for the simulation model, while PROPHET.config file sets up the experimental configuration for the simulation model.

3.3.1 The Traces

A trace is usually a list of time-ordered records of events and their associated variables obtained on a real system [Jain91]. This trace of events is used as the input to a simulation model to model the

run time environment of the real system. In our study since we are interested in studying processes' memory access interference, the traces we made would therefore intend to represent a real switch environment in terms of memory references.

A telephone switch consists of thousands of processes. A process on a real telephone switch performs one or multiple of transactions during one time slice. Each transaction completes one piece of relatively independent work. In the course of the execution, a transaction accesses a set of memory locations to do its assigned work. The memory reference pattern of a transaction can be modeled in a formatted file, called the trace file. A trace file contains a sequential list of memory locations, and every location is associated with a time reference. Figure 3-3 gives an example of a trace file. The first column of entries is a list of ordered memory locations. The second column of entries indicates the percentage of the total time for the transaction been consumed when reaching a location. For example, when the transaction reaches address 20a67300, 80% of the time assigned to this transaction is used up. The trace file is the basic element of our workload. When fed into the simulator, each trace file provides a run time memory reference scenario for a transaction.

memory locations	time reference
...	
2003f9a0	0.2965
208d1a60	0.4334
267049a0	0.4498
22fff940	0.5364
241184c0	0.5773
20ad8900	0.7246
20a67300	0.8000
24330460	0.9164
...	

Figure 3-3 The format of a trace file

A trace is a formatted file that contains a list of trace files. Hundreds of thousands of trace files are linked in an orderly fashion according to a certain type of model, referred to as the call model, which describes the incoming call events of a real switch. Figure 3-4 shows a segment of a trace. The first column of entries is a list of trace files. Each trace file corresponds to a transaction. The second column of entries is a list of timing statistics which indicates how much time the associated transaction would take in average. A trace provides a run time memory reference scenario for a process domain. In our simulation model, the real system is described as three process domains. The workload, therefore, includes three traces, named as the primary trace, the serial trace, and the gang trace. During the experiment, the three traces are opened by callGenerator, serialGenerator and gangGenerator respectively, and the name of each trace file as well as the transaction average work time are extracted from the trace, then piggybacked on a message. When the process receives the message, it then opens the trace file, accesses the list of memory locations sequentially. If a process finishes visiting every memory locations of a trace file without collision, it commits this transaction; otherwise it inserts the trace file back to the origination queue and requests a rollback.

list of trace files	average work time of transaction in time units
...	
xxx_ATCH_CALLP.3825.CALLP.00.addr	28.47
xxx_RABSC_HO_CALLP.3777.CALLP.07.addr	25.533
xxx_ATCH_CALLP.3825.CALLP.02.addr	13.130
xxx_ERBSC_HO_CALLP.3775.CALLP.07.addr	32.528
...	

Figure 3-4 The format of a trace

3.3.2 The blocking parameters

Two processes that are executing on two processors collide if they access the same memory location. If two trace files contain common memory location(s), these location(s) are the potential points of data contention because two transactions may collide if they happen to be executing in parallel. In this thesis the potential blocking characteristic of the traces is described by parameter Q . Blocking parameter Q is defined in [Drweiga96.2] as the probability that two processes collide over a memory access, if they are executed in parallel. The factors that affect the blocking parameter Q are the memory location overlap between traces, the average work time of the transactions, and the frequency of the transaction's execution. Generally, the bigger the average work time of the transaction, the longer the transaction holds the memory location, thus the more chance it blocks other transactions. The more frequent a transaction executes, the more chance it collides with others. Collision may come from intra process domain when two or more trace files within the same trace have common memory locations, or inter process domains when two or more trace files in different traces share some memory locations. We denote the potential intra primary domain blocking as Q_{pp} , the intra secondary domain blocking as Q_{nn} , and the inter blocking between primary and secondary as Q_{np} (see Section 2.4).

In the development of the workload, our emphasis is on properly introducing common memory locations between trace files so that the blocking parameters of traces can represent the real system's memory contention scenarios. For the reason of simplicity, artificial memory locations are inserted into trace files in many cases. With these artificial memory locations, the overlap between trace files can be introduced in a controlled manner. Contention measurement tools

[Kolski97] are used to help estimate the blocking parameters for the traces used in the simulation model.

The values of the blocking parameters used in our workload are:

$Q_{pp} = 0.036$

$Q_{np} = 0.17, 0.40, \text{ and } 0.59.$

$Q_{nn} = 0.0, 0.079, 0.14, \text{ and } 0.24$

For every Q value, there is a trace created to represent the corresponding memory contention scenario.

3.3.3 PROPHEt.config File

PROPHEt requires a formatted configuration file, called PROPHEt.config file, to set up the experiment engine for the simulator. The PROPHEt.config file contains parameters required to the experiment, such as warm up length, batch duration, batch number etc. This file is mandatory. Figure 3-5 shows an example of PROPHEt.config file used in our experiment. `primaryInterRequestTime`, `serialInterRequestTime`, and `gangInterrequestTime` are the configuration parameters that adjust the three types of message generation rates respectively. These, in turn, are adjusted to achieve a system utilization around 95%. `Algorithm` and `option` are where we set up the scheduling algorithm, i.e. serial, hybrid or gang with wait or non wait option. `PrimaryShare`, `SerialShare` and `GangShare` are the scheduler information necessary for engineering the processing resource. `ServiceTime` is used for computing overheads such as the cost incurred in inter-process communication, resolving memory access conflicts, and performing a context switch that includes the cost of running the scheduler. `Number of PEs` is the system configuration parameter. `Rollback`

schemes and Number_Priorities relate to the rollback mechanisms used in the system. The discussion of the rollback mechanisms is beyond the scope of this thesis. Num_requests_warmup, num_request_batch and the num_batches are the experiment parameters that allow our simulation model to reach a steady state before the final statistics get collected, and reduces the bias introduced by starting with an empty system.

Number of experiments	1	;	
primaryInterRequestTime	6.7	;	Mean time between requests with exponential distribution
serialInterRequesTime	50.5	;	
gangInterRequestTime	30.5	;	
algorithm	1	;	0==serial, 1==gang, 2==hybrid
option	0	;	0==wait, 1==nowait
primaryShare	6030	;	
serialShare	0	;	
gangShare	730	;	
serviceTime	17.6	;	cost for computing overheads
numberOfPEs	5	;	Number of PEs
Rollback_scheme	0	;	0==Bounded, 1==Age-Based
Number_Priorities	2	;	If Bounded rollback, range from 0 to n
NUM_REQUESTS_WARMUP	3000	;	Length of warmup
NUM_REQUESTS_BATCH	300	;	Length of batch
NUM_BATCHES	30	;	Number of batches of experiment

Figure 3-5 An example of PROPHET.config file

3.4 Output of the simulation model

3.4.1 Experiment Report File

The outputs of the simulation model are presented in the format of a PROPHET experiment report file. The report file gives a set of statistics collected automatically by PROPHET, called the system

statistics, and a set of statistics defined by users, called the user statistics. System statistics include statistics on processor utilization, message queues and scheduler ready queues, etc., while user statistics can be anything that users are interested in.

3.4.2 Validation Process

At the time when this research was conducted the implementation of the multiprocessor based system was incomplete. Therefore the modeling process has gone through the following steps of validation and verification:

1. Primary domain validation:

By disabling serial and gang domains and enabling primary domain only, the validation is made against a previously developed model used for a primary process domain contention study. Because our model extends the previously developed model to include serial process domain and gang process domain, and because the previous model has been very well validated and verified, it is therefore the baseline against which our simulation model validates. The PROPHET result reports show that the two models produce almost identical output.

2. Secondary process serialization validation:

By disabling gang domain and enabling primary and serial domains, this step of validation is made against an analytical model built for this project [Drweiga96.1]. During the validation, the serial scheduling algorithm has been activated. The validation criteria is the measure of serial idle which has been suggested by NORTEL experts. Table 3-2 provides the result of validation. The differences between our simulation result and analytical model prediction are less than 10%.

Table 3-2 Serial idle from experiment versus analytical prediction

Number of PE	2	3	4	5	6	7	8	9	10
Share for serial	20%	20%	20%	20%	17%	14%	13%	11%	10%
Simulation result	3.1%	5.6%	8.8%	13.2%	14.2%	12.1%	10.5%	10.8%	9.1%
Analytical prediction	2.8%	5.3%	8.4%	11.4%	13.1%	10.9%	10.7%	10.2%	10.4%

3. Gang synchronization validation:

By disabling the serial domain and enabling the primary and gang domains, this step of validation compared the experts' intuition on the synchronization idle in this multiprocessor based call control system with the simulator output. During the validation, only the gang algorithm was activated. The results of synchronization idle from experiments were found to be reasonable by Nortel experts working in this area.

Chapter 4

Results of Experiment

This chapter presents the experimental results of the simulation conducted for investigating the performance of serial, gang and hybrid scheduling algorithms. It is organized into five sections. Section 4.1 defines the performance index used in the experiments. Section 4.2 evaluates the performance of three scheduling algorithms in the scenarios of high Qnp. Section 4.3 evaluates the performance of three scheduling algorithms in the scenarios of low Qnp. Section 4.4 discusses the impact of variation in workload parameters. Section 4.5 discusses a number of additional issues on the overhead associated with the three scheduling algorithms.

4.1 Performance Index and Contention Matrix

The performance of a scheduling algorithm is measured in terms of Primary Capacity and Overall Capacity in this chapter.

- **Primary Capacity:** Primary Capacity is defined as useful processor seconds per second spent doing primary domain work. It is expressed in number of DMS SuperNode processors which is denoted by SuperNode_PEs in this thesis.

It can be presented as

$$\text{Primary Capacity} = (\text{Primary_commit} * \text{SuperNode_primary_AWT}) / \text{Elapsed_time}$$

where Primary_commit is the number of primary transaction commits during one experiment.

Elapsed_Time is the total elapsed time of the experiment.

SuperNode_primary_AWT is average work time per primary domain transaction measured on DMS SuperNode

- Overall Capacity: Overall capacity is defined as useful processor seconds per second spent doing both primary and secondary domains work. It is also measured in terms of SuperNode_PEs

It can be described as

$$\text{Overall Capacity} = \text{total_commit} * \text{SuperNode_primary_AWT} / \text{Elapsed_time}$$

where total_commit is the number of transaction commits during one experiment.

Elapsed_Time is the total elapsed time of the experiment.

SuperNode_primary_AWT is average work time per primary domain transaction measured on DMS SuperNode. Note that we used the average work time per primary domain transaction because we did not have separate data for secondary transactions.

- Overhead: Different types of overheads were described in Section 2.5. The simulation reports these overheads in terms of percentages of the total processing time.

The input of the simulator consists of three streams of the traces and a PROPHET.config file. The traces provide memory access contention scenarios for the simulator and the PROPHET.config file sets up the experimental configuration for the simulator. The memory contention scenario is described in terms of a contention matrix.

- **Contention Matrix:** Contention Matrix is a combination of blocking parameters in the form of $\{Q_{pp}, Q_{np}, Q_{nn}\}$. Recall that Q_{pp} is the probability of blocking when a primary process runs concurrently with another primary process. Q_{np} is the probability of blocking when a secondary process runs concurrently with a primary process. Q_{nn} is the probability of blocking when a secondary process runs concurrently with another secondary process. In our experiments Q_{pp} is fixed at 0.036 while both Q_{np} and Q_{nn} vary from experiment to experiment. One contention matrix represents a shared data contention scenario estimated for a product. The contention matrices used in the experiments were selected in consultation with Nortel personnel.

We are interested in the performance of scheduling algorithms when the system is running at a high capacity. For all the simulation experiments described in the thesis, the arrival rates for the three generators shown in Figure 3-2 are chosen in such a way that an approximately 95% average CPU utilization is achieved. The average execution time AET_p is fixed at 20.6 time units.

4.2 High Contention Between Primary And Secondary

In this section, we present the experimental results to evaluate the performance of each scheduling algorithm in the scenario of high contention between primary and secondary. The contention matrix used in the workload are $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.079\}$ and $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14\}$. The performance indices used in this section are primary capacity and overall capacity.

4.2.1 Primary Capacity

The primary capacity achieved under contention scenario $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14\}$ for each of the algorithms is presented in Figure 4.1. A numeric value of capacity as well as a percentage capacity improvement (compared against the serial algorithm) is also included in the figure for convenient reference. Note that in the experimental results the serial algorithm is presented as the hybrid algorithm at $s:g = 1:0$, and the gang algorithm is presented as the hybrid algorithm at $s:g = 0:1$.

	s:g=1:0	s:g=3:1	s:g=1:1	s:g=0:1
PE=3	1.86	1.84/-1.1%	1.92 /+3.2%	1.96/+5.4%
PE=5	2.49	2.49/+0%	2.75/+10.4%	2.90/+16.5%
PE=7	3.74	3.72/-0.5%	4.05/+8.3%	4.10/+9.6%
PE=9	4.37	4.37/+0%	4.82/+10.3%	5.00/+14.1%
PE=10	4.44	4.41/-0.7%	4.90/+10.4%	5.30/+19.4%

Figure 4-1 Primary Capacity in Scenario $Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14$

For any given algorithm, the primary capacity increases initially with an increase in the number of processors. However the increase in performance diminishes as the system reaches 9 PEs. The rationale for this behavior is presented. Although more work can be done with a higher number

PEs, the memory contention is observed to increase significantly as the number of PEs is raised. As a result the net improvement in performance is diminished.

For any given number of PEs, the gang algorithm (case $s:g = 0:1$) demonstrates the best performance, followed by the hybrid algorithm with $s:g = 1:1$, then the serial algorithm. For both gang algorithm and the hybrid algorithm the primary capacity improves more at higher number PE configurations. For the gang algorithm, a 5.4% primary capacity improvement is observed at 3 PE configuration, and 19.4% primary capacity improvement is observed at 10 PE configuration. For the hybrid algorithm with $s:g = 1:1$, a 3.2% primary capacity improvement is observed at 3 PE configuration, 10.4% primary capacity improvement is observed at 10 PE configuration. For the hybrid algorithm with $s:g = 3:1$ no performance improvement is observed at all PE configurations. Note that an improvement in capacity (primary capacity) by even few percents through the use of an effective scheduling technique is important. Nortel spends approximately \$55,000 in terms of manpower for producing 1% improvement in capacity [Carroll99].

Knowing that higher number PE system results in more collisions, this result indicates that if primary to secondary contention is high, the blocking between primary and secondary is the dominant obstacle to achieving a desirable system performance. The gang algorithm has the best ability in reducing shared data contention, followed by hybrid algorithm then serial algorithm. Therefore the way a scheduling algorithm works in reducing the interaction between primary and secondary determines its performance. The serial algorithm limits the interaction between secondary and primary processes in such a way that, at any time in the whole system, there is only one running secondary process; no parallel execution is allowed among secondary domain processes. This eliminates the contention between two secondary processes but gives rise to some

degree of interaction between primary and secondary domains. The gang algorithm eliminates the interaction between secondary and primary domains by scheduling primary processes and secondary processes in totally separate batches, isolating the memory access contention between the two process domains. There is collision among primary processes or collision among secondary processes but no contention across domains. Of the three algorithms, gang algorithm is the most effective one in eliminating interference between primary and secondary. As a result it shows up to 19.4% performance improvement over serial algorithm. However there is a disadvantage of the gang algorithm: process scheduling latency. For maintaining a healthy switch some processes cannot tolerate a long delay from being scheduled in. But with gang scheduling they may have to wait for their batch to run. Scheduling latency is not covered in this thesis, but it is an important factor in considering a scheduling algorithm. The hybrid algorithm attempts to compensate for the scheduling latency incurred in gang algorithm by engineering a portion of secondary processes into a gang domain and the other portion of secondary processes into the serial domain. Candidate processes for the serial domain are processes that are either sensitive to scheduling latency or contend too much with other secondary processes. Candidate processes for the gang domain either contend too much with primary domain to be able to run in parallel with primary processes or contend very little with other gang domain processes, so that they can run in parallel without many rollbacks. The Hybrid algorithm is more effective in reducing the interaction between primary and secondary domains than serial algorithm. As a result, at 1:1 serial to gang ratio it shows up to 10.2% performance improvement over serial algorithm. The problem with hybrid algorithm is that it has both serial idle and synchronization idle. The overhead is so high that at 3:1 serial to gang ratio, hybrid scheduling shows the worst performance among the three algorithms.

The effectiveness of reducing the interaction between primary and secondary contributes to improving system capacity from two aspects.

1. Reduce wasted processing power caused by rollback.

Whenever a collision occurs, the processing time spent on the process rolled back is wasted. Besides, the system has to perform a variety of activities such as arbitrating one process to be rolled back, putting the rolled back process in the ready queue, scheduling in another process etc. to resolve the collision. This extra processing time spent on the system's reacting upon a collision forms another type of system overhead. Table 4-1 gives an example of the percentage of rollback overhead, the result is obtained under contention scenario $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14\}$ which results in the worst contention. The result shows that the rollback overhead increases in the following order: gang $\rightarrow$ hybrid $\rightarrow$ serial, reflecting their ability in reducing shared data contention.

Table 4-1 Rollback Overhead

Number of PEs	3	5	7	9	10
Serial algorithm	14.6%	20.2%	23.7%	25.9%	27.1%
Hybrid algorithm					
s:g = 3:1	13.7%	18.1%	21.7%	23.1%	24.2%
s:g = 1:1	12.8%	15.9%	18.4%	20.3%	22.7%
Gang algorithm	10.2%	13.4%	15.7%	18.7%	19.8%

2. Reduce rollback serialization idle

If a process is rolled back consecutively for a certain number of times, the system will take special care of it. First, the processes that reach the maximum rollback number allowed are serialized among themselves to avoid contending with each other; Second, the processes having maximum rollback numbers always win arbitration if they collide with processes having less rollback numbers. This rollback strategy introduces a new system overhead. Sometimes the system arrives at a situation in which a processor can not find any new work to do, at the same time there is a list of processes with the maximum number of rollbacks waiting there to be executed. Even so, processors have to set themselves idle to wait for rollback serialization constraints to be free. This type of idle is called rollback serialization idle. Rollback serialization idle reflects the memory contention level of the system. The higher the rollback serialization idle is, the more the contention has occurred. Table 4-2 shows the experimental results of rollback serialization idle under contention scenario $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14\}$ which again is the worst contention scenario. The rollback serialization idle increases in the order of gang $\rightarrow$ hybrid $\rightarrow$ serial, indicating that gang scheduling algorithm produces the least collisions, and serial scheduling algorithm produces the most collisions.

Table 4 - 2 Rollback Serialization Idle

Number of PEs	3	5	7	9	10
Serial algorithm	0%	0.2%	3.8%	11.0%	12.4%
Hybrid algorithm					
s:g = 3:1	0%	0.2%	2.4%	6.4%	10.8%
s:g = 1:1	0%	0.2%	1.8%	5.2%	7.7%
Gang algorithm	0%	0.0%	1.2%	3.7%	6.8%

The result obtained with another high Q_{np} scenario $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.079\}$ is presented in Figure 4-2. The relative performance of the scheduling algorithms are found to be similar to those captured in Figure 4-1. Because of a lower Q_{nn} value, a slightly higher primary capacity in comparison to Figure 4-1 has been achieved in some cases.

	s:g=1:0	s:g=3:1	s:g=1:1	s:g=0:1
PE=3	1.86	1.84/-1.1%	1.94 /+4.3%	1.97/+5.9%
PE=5	2.49	2.54/+2.0%	2.81/+12.9%	3.1/+24.5%
PE=7	3.74	3.73/-0.3%	4.15/+11.0%	4.3/+15.0%
PE=9	4.37	4.4/+0.7%	4.84/+10.8%	5.2/+19.0%
PE=10	4.44	4.42/-0.5%	4.90/+10.4%	5.41/+21.8%

Figure 4-2 Primary Capacity in Scenario $Q_{np}=0.59$ $Q_{nn}=0.079$

4.2.2 Overall Capacity

Figure 4-3 and Figure 4-4 present the overall capacity achieved with scenario $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14\}$ and scenario $\{Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.079\}$ respectively. As expected, the overall capacity for a given algorithm, number of PE's and Scenario is higher than the

	s:g=1:0	s:g=3:1	s:g=1:1	s:g=0:1
PE=3	2.3	2.29/-0.4%	2.37 /+3.0%	2.41/+4.8%
PE=5	2.96	3.01/+1.7%	3.29/+11.1%	3.50/+18.2%
PE=7	4.13	4.18/+1.2%	4.54/+9.9%	4.88/+18.2%
PE=9	4.64	4.81/+3.7%	5.32/+14.7%	5.80/+25.0%
PE=10	4.79	4.88/+1.9%	5.44/+13.6%	6.30/+31.5%

Figure 4-3 Overall Capacity in Scenario Qnp=0.59 Qnn=0.14

	s:g=1:0	s:g=3:1	s:g=1:1	s:g=0:1
PE=3	2.34	2.34/+0%	2.42/+3.4%	2.5/+6.8%
PE=5	3.02	3.02/+0%	3.35/+10.9%	3.7/+22.5%
PE=7	4.14	4.24/+2.4%	4.59/+10.9%	4.94/+19.3%
PE=9	4.72	4.85/+2.8%	5.39/+14.2%	5.89/+24.8%
PE=10	4.82	4.90/+1.5%	5.49/+13.9%	6.40/+32.8%

Figure 4-4 Overall Capacity in Scenario Qpp=0.036 Qnp=0.59 Qnn=0.079

corresponding primary capacity. The relative performance of the scheduling algorithms are the same as observed with the results on primary capacity discussed in the earlier section.

4.2.3 Sensitivity to Qnn

Another observation is that for both the gang and the hybrid algorithms, when Qnn varies from 0.14 to 0.079, the primary capacity for a given number of PE's is not affected by a noticeable degree: see Figure 4-1 and 4-2. The reason is that, 1) the variation of Qnn is not big enough; 2) even if the number of rollbacks increase in proportion with the Qnn change, the processing time allocated to gang process domain is small - one PE worth of processing time at most. Given such a small share for the gang process domain, the variation of Qnn could not have significant impact over the whole system. This observation suggests a strategy for disjointing domain overlap – disjoint the overlap between the primary domain process and the secondary domain process, leave the gang domain the way it is since the change in Qnn from one scenario to another has a minor impact on the system performance.

4.2.4 Capacity vs. Number of PEs

Table 4-3 and Table 4-4 list the speedup in primary capacity for three algorithms under contention scenario {Qpp=0.036, Qnp=0.59 Qnn=0.14} and {Qpp=0.036, Qnp=0.59, Qnn=0.079}. We define the ratio of primary capacity to number of PEs in the system as primary capacity speedup. Under this definition, the primary capacity speedup is the best with the 3 PE configuration, then getting worse as more PEs are added into the system. The exception point is the 5 PE configuration, where the primary capacity speedup drops abruptly.

The primary capacity speedup does not grow linearly as more processors are added into the system. To arbitrate memory contention and preserve the integrity of shared data, certain types of software control are necessary. These controls incur a system overhead – the cost for paying for memory contention resolution. As well as the contention resolution overhead, there are also scheduling idles which are really the side-effects brought by the scheduling algorithms. All of these overheads increase as more PEs are added. As a result the primary capacity speedup tends to decrease with an increase in the number of PE's.

Of all PE configurations, the 5 PE is somewhat exceptional – its capacity speedup is the same as or even lower than 7 PE configuration's. The reason for this exception is that, with 5 PE's the secondary domain share reaches its maximum of 20%. With 20% of secondary domain share serial algorithm results in the highest serial idle compare to other PE configurations, leading to a lower capacity speedup.

Table 4-3 Primary Capacity Speedup under scenario $Q_{pp}=0,036$ $Q_{np}=0.59$ $Q_{nn}=0.14$

Number of PE		3	5	7	9	10
Serial algorithm	primary capacity	1.86	2.49	3.74	4.37	4.44
	capacity speedup	62%	49%	53%	49%	44%
hybrid algorithm (s:g=1:1)	primary capacity	1.94	2.75	4.05	4.82	4.90
	capacity speedup	65%	55%	58%	54%	49%
gang algorithm	primary capacity	1.89	2.9	4.1	5	5.3
	capacity speedup	63%	58%	59%	56%	53%

Table 4-4 Primary Capacity Speedup for Scenario $Q_{pp}=0.036$ $Q_{np}=0.59$ $Q_{nn}=0.079$

Number of PEs		3	5	7	9	10
Serial algorithm	primary capacity	1.86	2.49	3.74	4.37	4.44
	capacity speedup	62%	49%	53%	49%	44%
hybrid algorithm (s:g =1:1)	primary capacity	1.92	2.92	4.15	4.84	4.90
	capacity speedup	64%	58%	59%	54%	49%
gang algorithm	primary capacity	1.97	3.1	4.33	5.2	5.45
	capacity speedup	66%	62%	62%	58%	55%

4.2.5 Summary

- The high Q_{np} is the key obstacle that hurts the system capacity. As a result, the effectiveness in reducing the interaction between primary processes and secondary processes determines the performance of each of the three scheduling algorithms under high Q_{np} . The gang algorithm performs the best, followed by the hybrid algorithm, then serial algorithm.
- If the total secondary domain share does not exceed one PE worth of processing time, disjointing the secondary domain for reducing Q_{nn} is not recommended because the primary capacity is not severely impacted by the change of Q_{nn} .
- Under high Q_{np} (0.59), 3 PE configuration gives the best capacity speedup of 63%. As more PEs are added into the system, the capacity speedup is observed to drop.

4.3 Low Contention between Primary and Secondary

In this section, we present the experimental results to evaluate the performance of three scheduling algorithms under low Q_{np} scenarios. The contention scenarios used are $\{Q_{pp} = 0.036, Q_{np} = 0.17, Q_{nn} = 0.14\}$ and $\{Q_{pp} = 0.036, Q_{np} = 0.17, Q_{nn} = 0.079\}$. Again we use primary capacity and overall capacity as performance evaluation indices.

4.3.1 Primary Capacity

Figure 4-5 shows the performance comparison of three scheduling algorithms under low Q_{np} scenario $\{Q_{pp}=0.036, Q_{np}=0.17, Q_{nn}=0.079\}$. At 3 PE, 9 PE, and 10 PE configurations, the serial algorithm, the gang algorithm and the hybrid algorithm at $s:g = 1:1$ demonstrate comparable performance. At 5 PE and 7 PE configurations, the gang algorithm and the hybrid algorithm with $s:g = 1:1$ ratio perform better than the serial algorithm. At most PE configurations the hybrid algorithm at $s:g=3:1$ performs the worst in comparison with other scheduling algorithms. The rationale behind such a behavior is the increasing importance of the scheduling idles at lower Q_{np} scenarios. A short discussion of this is presented next.

From Table 4-5 we can observe that, at 5 PE and 7 PE configurations, the gang algorithm and the hybrid algorithm at $s:g=1:1$ results in a smaller scheduling overhead (i.e. serial idle + sync. idle) than the serial algorithm. This smaller scheduling overhead contribute to the primary capacity improvement seen in Figure 4-5 for the gang algorithm and the hybrid algorithm at $s:g=1:1$.

	s:g=0:1	s:g=3:1	s:g=1:1	s:g=0:1
PE=3	2.00	1.96/-2.0%	2.02 /+1.0%	2.01/+0.5%
PE=5	2.79	2.83/+1.4%	3.12/+11.8%	3.15/+12.9%
PE=7	4.25	4.29/+0.9%	4.47/+5.2%	4.52/+6.4%
PE=9	5.54	5.18/-6.5%	5.49/-0.9%	5.51/-0.5%
PE=10	6.05	5.80/-4.1%	6.03/-0.3%	6.08/+0%

Figure 4-5 Primary Capacity in Scenario $Q_{pp}=0.036$ $Q_{np}=0.17$ $Q_{nn}=0.079$

Table 4-5 Scheduling overhead observed in scenario $Q_{pp}=0.036$ $Q_{np}=0.17$ $Q_{nn}=0.14$

Number of PEs	3	5	7	9	10
Serial algorithm	4.8%	14.7%	12.2%	9.8%	9.1%
Hybrid algorithm					
s:g = 3:1	5.4%	14.6%	14.0%	14.5%	14.9%
s:g = 1:1	4.8%	13.3%	12.1%	14.1%	14.4%
Gang algorithm	3.8%	5.7%	6.6%	12.1%	12.8%

At 9 PE and 10 PE configurations, the hybrid algorithm at s:g=1:1 and the gang algorithm result in bigger scheduling overheads (sum of serial idle and sync. idle) than the serial algorithm. These bigger scheduling overheads compromise the capacity improvement gained from reduced number of rollbacks, leading to the comparable performance seen in Figure 4-5 for the three algorithms. The hybrid algorithm at s:g=3:1 results in the highest scheduling overheads at almost all configurations, therefore it has the lowest performance.

Table 4-6 Number of Collisions Observed From Scenario $Q_{pp}=0.036$ $Q_{np}=0.17$ $Q_{nn}=0.14$

Number of PEs	3	5	7	9	10
Serial algorithm	535	1161	1302	1575	1704
Hybrid algorithm					
s:g = 3:1	484	1109	1294	1501	1672
s:g = 1:1	476	986	1136	1445	1529
Gang algorithm	435	898	992	1301	1447

From Table 4-6 we can observe that for any given number of PE's, the serial algorithm produces the most number of rollbacks, followed by hybrid algorithm, then gang algorithm. This indicates that, under low Q_{np} (0.17) scenarios, the contention between primary and secondary domains is no longer the dominant factor that affects the performance of scheduling algorithms. The contribution of scheduling overhead becomes more important. Although the gang algorithm and the hybrid algorithm still reduce the interaction between the primary domain and the secondary domain effectively, the actual benefit gained from reducing the memory access contention may be offset by the loss of processing power caused by scheduling overhead. When that happens the performance of the hybrid algorithm and the gang algorithm drop to even lower than the serial

algorithm's. This is the reason why the hybrid algorithm with $s:g=1:1$ ratio and the gang algorithm perform no better than the serial algorithm at $PE=9$ and $PE=10$, even though they do reduce the interaction between the primary domain and the secondary domain.

The $PE=5$ configuration is an exceptional case in the low Qnp scenario. The primary capacity of serial algorithm is 10% lower than gang algorithm and hybrid algorithm with $s:g=1:1$. The reason for this is the serial algorithm results in a bigger scheduling overhead (serial idle) in comparison to other PE configurations due to the 20% share for the serial domain. While at higher number PE configurations the shares for the serial domain decrease to $1/N$, where N is the number of PE s in the system.

	$s:g=0:1$	$s:g=3:1$	$s:g=1:1$	$s:g=0:1$
PE=3	2.00	1.95/-2.5%	2.02 /+1.0%	2.01/+0.5%
PE=5	2.79	2.81/+0.7%	3.09/+10.8%	3.11/+11.5%
PE=7	4.25	4.26/+0.2%	4.39/+3.3%	4.44/+4.5%
PE=9	5.54	5.11/-7.8%	5.47/-1.3%	5.49/-0.9%
PE=10	6.05	5.73/-5.6%	6.03/-0.3%	6.05/+0%

Figure 4-6 Primary Capacity in Scenario $Qpp=0.036$ $Qnp=0.17$ $Qnn=0.14$

The result obtained with another low Qnp scenario {qpp=0.036 Qnp=0.17 Qnn=0.14} is presented in Figure 4-6. The relative performances of the scheduling algorithms are found to be similar to those Captured in Figure 4-5.

4.3.2 Overall Capacity

Figure 4-7 and Figure 4-8 shows the overall capacity of three scheduling algorithms under contention scenario {Qpp=0.036, Qnp=0.17, Qnn=0.079} and {Qpp=0.036, Qnp=0.17, Qnn=0.14} respectively. The overall capacity produced by the scheduling algorithm follows the same trend as the primary capacity. The rationale for this system behavior is discussed in detail in the previous section.

Figure 4-7 Overall Capacity in Scenario Qnp=0.17 Qnn=0.14

	s:g=0:1	s:g=3:1	s:g=1:1	s:g=0:1
PE=3	2.50	2.50/+0%	2.59 /+3.6%	269/+6.8%
PE=5	3.50	3.60/+2.9%	3.81/+8.9%	3.95/+12.9%
PE=7	4.88	4.88/+0%	5.11/+4.7%	5.24/+7.4%

Figure 4-8 Overall Capacity in Scenario $Q_{pp}=0.036$ $Q_{np}=0.17$ $Q_{nn}=0.079$

The results show that at 3, 5, 7 PE configurations, the gang algorithm performs the best, followed by the hybrid algorithm with $s:g = 1:1$, and then the serial algorithm. Since the primary focus of this thesis is on primary capacity that is of interest to Nortel and since the overall capacity may be of interested to applications other than a telephone switch, we run the overall capacity experiments at 3, 5, and 7 PE configurations only to reduce the simulation expense.

4.4 The Impact of Blocking Parameters

In this section, we study the sensitivity of three scheduling algorithms to the change of blocking parameters in the workload. The experiments are organized into two subsections, each focuses on one variation of the blocking parameters.

- 1) fix Q_{nn} at 0.14, vary Q_{np} .
- 2) fix Q_{np} at 0.59, vary Q_{nn} .

4.4.1 Variation of Q_{np}

The variation of Q_{np} would affect the system capacity – both primary capacity and overall capacity only if there is an interaction between primary processes and secondary processes. The impact of Q_{np} variation on primary capacity is presented in Figure 4-9 and Figure 4-10 that capture the performance change of the serial algorithm and the hybrid algorithm at $s:g=1:1$ respectively. The gang algorithm is not impacted by the variation in Q_{np} because of the separation of primary and secondary. It was observed from the results presented in Section 4.2 and Section 4.3 that the variation of Q_{np} has a significant impact on the performance of the serial algorithm and the hybrid algorithm. Here we further investigate this impact by studying the system for three different Q_{np} values.

Figure 4-9 The Impact of Q_{np} Variation for Serial Algorithm

Figure 4-10 The Impact of Qnp Variation for Hybrid Algorithm at s:g=1:1

Table 4-7 Primary capacity degradation when Qnp increase from 0.17, 0.40 to 0.59

		PE3	PE5	PE7	PE9	PE10
Serial algorithm	Qnp=0.17	2	2.79	4.25	5.54	6.05
	Qnp=0.40	1.94	2.68	4.04	5.04	5.37
	degradation	3%	3.9%	4.9%	9.0%	11.2%
	Qnp=0.59	1.86	2.49	3.74	4.37	4.44
	degradation	7%	10.8%	12%	21.1%	26.6%
Hybrid algorithm	Qnp=0.17	2.01	3.09	4.47	5.47	6.03
	Qnp=0.40	1.98	2.98	4.30	5.22	5.50
	degradation	1.5%	3.6%	3.8%	4.6%	8.8%
	Qnp=0.59	1.94	2.85	4.05	4.82	4.90
	degradation	3.5%	7.8%	8.1%	11.9%	18.7%

Table 4-7 shows primary capacity degradation when Q_{np} varies from 0.17 to 0.40 and to 0.59 for the serial algorithm and the hybrid algorithm with $s:g=1:1$. The contention scenarios are ($Q_{pp}=0.036$, $Q_{np}=0.17$, $Q_{nn}=0.14$), ($Q_{pp}=0.036$, $Q_{np}=0.40$, $Q_{nn}=0.14$) and ($Q_{pp}=0.036$, $Q_{np}=0.59$, $Q_{nn}=0.14$). The degradation is measured with respect to the primary capacity obtained under $Q_{np}=0.17$.

For any given number of PEs, the serial algorithm experiences the greatest impact – its primary capacity drops 26.6% when Q_{np} rises from 0.17 to 0.59 on a 10 PE system. The hybrid algorithm with $s:g=1:1$ experiences a smaller impact – only 18.7% primary capacity degradation is observed when Q_{np} rises from 0.17 to 0.59 on a 10 PE system. This indicates that by placing more secondary processes in the gang domain, the system becomes more robust to the sudden change of contention between primary and secondary processes, reducing the degradation of primary capacity.

4.4.2 Variation of Q_{nn}

The variation of Q_{nn} would affect system capacity only if there are secondary processes running in parallel. The serial algorithm serializes the execution of secondary processes so it is not impacted by the variation of Q_{nn} . We have observed from the results recorded in Section 4.2 and Section 4.3 that a variation of Q_{nn} from 0.14 to 0.079 does not necessarily affect the primary capacity. We concluded that the small change of Q_{nn} and the small share allocated to the gang domain make the system insensitive to the variation of Q_{nn} . Recall that the share allocation in this model is based on the assumption that the secondary domain gets less than one PE worth of processing power. However in real switch the secondary domain share changes from time to time, depending on the

phases in the project development [Carroll99]. Therefore the Qnn sensitivity analysis presented in this section is necessary. Here we further investigated the impact of Qnn by increasing the variation of Qnn from 0.24 down to 0.0. We consider a 5 PE system because this configuration obtains the maximum secondary domain share of 20%. It should therefore have the greatest impact in comparison to other PE configurations. The contention scenarios involved are {Qpp = 0.036, Qnp = 0.59, Qnn=0.0}, {Qpp=0.036, Qnp=0.59, Qnn=0.079} and {Qpp = 0.036, Qnp = 0.59, Qnn = 0.14}, serial to gang ratio are 1:0, 3:1, 1:1, 1:3 to 1:0. The reason we consider the s:g=1:3 case is to better understand at what point the system becomes insensitive to the change of Qnn.

Table 4-8 shows the primary capacity degradation when Qnn varies from 0.0, 0.079, 0.14 to 0.24 for systems with serial algorithm and hybrid algorithm. The degradation is compared against Qnn=0.0.

Table 4-8 primary capacity degradation when Qnn increase from 0.0, 0.079 to 0.14

	Serial algorithm	Hybrid algorithm			Gang algorithm
s:g	1:0	3:1	1:1	1:3	0:1
Qnn=0.0	2.49	2.58	2.98	3.17	3.4
Qnn=0.079	2.49	2.56	2.92	3.1	3.23
degradation	0%	0.8%	2.0%	2.2%	5%
Qnn=0.14	2.49	2.49	2.85	2.93	3.04
degradation	0%	3.5%	4.4%	7.5%	10.6%
Qnn=0.24	2.49	2.44	2.79	2.86	2.97
Degradation	0%	5.4%	6.4%	9.8%	12.6%

The gang algorithm is impacted most by the increase of Q_{nn} . A 5.9% drop in primary capacity is observed if Q_{nn} varies from 0.079 to 0.14, and a 10.6% drop in primary capacity is observed when Q_{nn} changes from 0.0 to 0.14. The hybrid algorithm with $s:g=1:1$ reduces the impact of a change in Q_{nn} significantly – only a 2.4% drop in primary capacity is observed when Q_{nn} changes from 0.079 to 0.14, and a 4.4 % drop in primary capacity is observed if Q_{nn} changes from 0.0 to 0.14.

4.4.3 Summary

The effect of Q_{np} variation is the highest for the serial algorithm, followed by the hybrid algorithm. A change in Q_{np} does not impact gang algorithm because there is no interaction between primary domain and secondary domain.

The impact of variation in Q_{nn} on the gang algorithm is the highest, followed by the hybrid algorithm. A change in Q_{nn} does not impact serial algorithm because there is no secondary process running in parallel with other secondary processes. The hybrid algorithm with a $s:g=1:1$ smoothes the impact caused by the swing of Q_{np} and Q_{nn} . It demonstrates the characteristic of robustness required for applications like telephone switches.

4.5 The Impact of Other Workload Parameters

In this section, we study in detail how workload parameters other than blocking parameters impact the performance of the scheduling algorithms. We organize this section into three subsections:

- 1) The impact of average execution time (AET) of transaction
- 2) The frequency of batch switching
- 3) Summary

4.5.1 Average Execution Time of Transactions

So far in the experiments of our study, the same Average Execution Time is used for both primary process domain transactions and secondary process domain transactions. This is because we do not have statistics about average execution time for the secondary domain as input for our simulation model, and it is very difficult to obtain one due to the extremely diverse nature of secondary domain software [Benniger98]. However, in a real system the Average Execution Time for secondary domain transactions may be longer than the Average Execution Time for primary domain transactions. In this section we use 2 times of average work time per primary transaction for secondary domain transactions to investigate the impact of average work time on the performance of the scheduling algorithms.

The average execution time of the primary transaction is denoted by AET_p, and the average execution time of the secondary transaction is denoted by AET_s. The contention scenario is {Q_{pp}=0.036, Q_{np}=0.59, Q_{nn}=0.14}. System configurations are PE=5 and PE=10.

Table 4-9 and Table 4-10 compare the degradation of primary capacity among three algorithms when increasing AET_s.

Table 4-9 Performance Degradation after Increasing AET_s for 5 PE Configuration

	Serial algorithm	Hybrid algorithm (s:g =3:1 s:g = 1:1)		Gang algorithm
AET _s = AET _p	2.49	2.49	2.85	3.05
AET _s = 2*AET _p	1.84	1.86	2.19	2.41
Capacity Degradation	26%	25.3%	23%	21%

Table 4-10 Performance Degradation after Increasing AET_s for 10 PE Configuration

	Serial algorithm	Hybrid algorithm (s:g =3:1 s:g = 1:1)		Gang algorithm
AET _s = AET _p	4.44	4.41	4.90	5.30
AET _s = 2*AET _p	3.11	3.09	3.70	4.07
Capacity Degradation	30%	29.9%	24.5%	23.2%

Under 5 PE configuration, there is a 26% drop in primary capacity for the serial algorithm, a 21% drop for the gang algorithm and a 23% drop for the hybrid algorithm with s:g=1:1. On a 10 PE system, there is a 30 % primary capacity degradation for serial algorithm, a 23.2% for gang algorithm and a 24.5 % for hybrid algorithm with 1:1 serial to gang ratio. These results demonstrate that the impact of increasing AET_s is significant for all three algorithms. Two factors contribute to the capacity degradation:

- Increasing scheduling idle. Table 4-11 shows the scheduling idle comparison before and after increasing AET_s. Under serial algorithm, a PE that is scheduled to execute secondary process

A has to wait until the current running secondary process B releases the serial token. The time spent on waiting depends on when the PE starts its waiting: at the beginning of process B's transaction or near the end of process B's transaction, and the time process B takes to finish its transaction. Usually the longer process B takes in processing, the longer the serial idle a PE experiences. Similarly under gang algorithm, all PEs that are scheduled to run processes from the next batch also have to wait until the last process in the current batch commits or rolls back. Usually the longer the transaction in the current batch, the bigger the synchronization idle. Both serial idle and synchronization idle experienced by the serial algorithm and the gang algorithm respectively are observed to increase when AET_s is increased. Both serial idle and synchronization idle increase for the hybrid algorithm when AET_s is increased.

Table 4-11 Scheduling Overhead Before and After Increasing AET_s

algorithm	no. of PEs	before		after	
		serial idle	sync. idle	serial idle	sync. idle
serial algorithm	5	13.8%	0%	23.5%	0%
	10	9.1%	0%	18.4%	0%
hybrid algorithm s:g=3:1	5	10.1%	5.0%	19.4%	5.4%
	10	7.7%	6.5%	18.5%	7.1%
hybrid algorithm s:g=1:1	5	6.4%	4.9%	12.1%	5.4%
	10	5.7%	6.3%	10.1%	6.9%
gang algorithm	5	0%	4.8%	0%	5.3%
	10	0%	6.1%	0%	6.9%

- Increasing number of rollbacks: Table 4-12 shows the number of rollbacks before and after increasing AET_s. We take 5 PE system as an example.

Table 4-12 Number of Rollbacks Before and After Increasing AET_s

algorithm	before	after
serial algorithm	3756	4384
hybrid algorithm s:g=3:1	3577	4159
hybrid algorithm s:g=1:1	3064	3433
gang algorithm	2557	2891

When AET_s is increased, the time a secondary process holds shared memory line is longer, causing more chances of collision among contentious processes.

4.5.2 Frequency of Batch Switch

Synchronization idle occurs at every batch switching with gang algorithm and hybrid algorithm. It is easy to understand that the smaller the frequency of batch switch, the smaller the synchronization idle which in turn leads to higher system capacity. However, the process scheduling latency requirement imposed on some processes has restricted the frequency of batch switch to be high.

In this subsection, we increase the batch time by 25% of the original. Other workload parameters remain the same as original. That is, APT_s = APT_p, Q_{pp} = 0.036, Q_{np} = 0.59 and Q_{nn} = 0.14. We chose the 10 PE configuration in this experiment because it results in the highest synchronization idle.

The result of this experiment is given in terms of primary capacity (see Table 4-13). Compared to the results achieved before reducing the frequency of batch switch, there is a 7.5% primary capacity improvement for gang algorithm and a 6.6% improvement for hybrid algorithm with s:g = 1:1 and a 3.4% for s:g = 3:1.

Table 4-13 the impact of frequency of batch switches

	Gang algorithm	Hybrid algorithm	
		s:g = 3:1	s:g=1:1
Before reducing the frequency of batch switch	5.30	4.41	4.44
After reducing the frequency of batch switch	5.70	4.56	4.70
Performance comparison	7.5%	3.4%	6.6%

Two factors contribute to the improvement of system performance after the reduction of the batch switch frequency:

- Synchronization idle. Table 4-14 shows the synchronization idle after / before reducing the frequency of batch switching. The reduction of synchronization idle brings back more processing power, increasing the system capacity.

Table 4-14 Sync. idle after / before reducing the frequency of batch switches

	Gang algorithm	Hybrid algorithm	
		s:g = 3:1	s:g=1:1
Before reducing the frequency of batch switch	6.1%	6.5%	6.3%
After reducing the frequency of batch switch	4.0%	5.2%	4.4%

- Keep all PEs occupied during gang batch. In section 4.2, we observed that a problem that had occurred during a gang batch: there was not enough work to keep all PEs occupied. The share for gang domain is too small, or the available processors are more than necessary at high PE configurations. This problem is particularly serious to the hybrid algorithm which has only a portion of the secondary processes in the gang domain. The hybrid algorithm is devised to address the problem of the process scheduling latency. With the hybrid algorithm we can keep on reducing the frequency of batch switch as long as the process scheduling latency requirement is not violated.

4.5.3 Summary

The increase of AET_s causes the performance of all three algorithms to drop by a significant degree. There is increased scheduling idle and an increased number of rollbacks. However, short AET_s implies more number of context switches, indicating increased scheduling overhead.

Reducing the frequency of batch switch will reduce the synchronization idle which in turn will improve the performance of the gang algorithm and the hybrid algorithm. Because of process scheduling latency constraints reducing frequency of batch switch may be meaningful only for hybrid algorithm.

4.6 Scheduling Algorithms with No Wait Option

In this section, we evaluate the performance of scheduling algorithms with no wait option. We

organize this section into three subsections:

- 1) Performance comparison of serial / hybrid algorithm with no serial idle option and serial / hybrid algorithm with serial idle option.
- 2) Performance comparison of gang / hybrid algorithm with no synchronization idle option and gang / hybrid algorithm with synchronization idle option.
- 3) Summary

4.6.1 Algorithms with No Serial Idle Option

Serial algorithm with wait option results in low processor utilization because of the serial idle – processors spin and wait for another running serial domain process to finish. This spin-wait is a solution for bringing a single processor system scheduler to the multiprocessor based call control system; however, it also brings the problem of low processor utilization which prevents fully realizing the available processing power. Simulation results indicate that serial idle tends to increase with an increase in the number of processors.

Serial algorithm without serial idle option tries to make good use of the processor by executing a primary process whenever the serialization constraint is not free. This non-wait proposal promises more efficient utilization of processors with the trade-off of additional effort in modifying existing strategy for processing time allocation across process domains, and with the drawback of introducing additional number of rollbacks which would not occur if PEs sit idle. In this subsection, we present the experimental results to identify the capacity potential of serial algorithm

and hybrid algorithm with no serial idle option; and to compare the performance of serial algorithm and hybrid algorithm with no serial idle option to serial algorithm and hybrid algorithm with serial idle option.

The workload parameters used in this subsection:

- Blocking parameters: $Q_{pp}=0.036$, $Q_{np}=0.59$, $Q_{nn}=0.14$.
- Processor configurations: PE=5 and PE=10.

Table 4-14 shows the primary capacity of serial algorithm and hybrid algorithm with no serial idle option, and the performance improvement over serial algorithm and hybrid algorithm with serial idle option.

Table 4-15 Primary Capacity With No Serial Idle Option vs. With Serial Idle Option

Algorithm	5 PE system	10 PE system
Serial with no serial idle	2.97	5.2
Serial with serial idle	2.49	4.44
Performance improvement	19.7%	17.1%
Hybrid (s:g=1:1) with no serial idle	3.05	5.27
Hybrid (s:g=1:1) with serial idle	2.75	4.90
Performance improvement	10.9%	7.6%

The performance improvement of scheduling algorithms with no serial idle option over scheduling algorithms with serial idle option is significant. The contribution is mainly from the better utilization of processors. Serial or hybrid scheduling algorithms with no serial idle option make use of 'serial idle' power and adds it into primary process domain. Not 100% of the 'wasted processing power' can be converted to do useful work though. When a PE skips spin-wait to execute primary processes, the 'extra' primary process may cause collisions that would not occur otherwise under scheduling algorithms with serial idle option. Nevertheless, the results encourage selecting the option of no serial idle.

4.6.2 Algorithms with No Synchronization Idle Option

Gang algorithm with wait option results in idling of processes – processors spin and wait for the last process from current process domain to finish before starting another process domain. This spin-wait helps in eliminating memory access contention between primary domain processes and secondary domain processes, but lowers the processor utilization which prevents fully realizing the available processing power. Gang algorithm with no sync. idle option tries to make use of the sync. idle time by allowing processors to start another process domain without waiting for the last process from current process domain to finish. This option introduces some degree of interaction between primary domain and secondary domain during batch switch. This subsection investigates whether the benefit of sync. idle elimination can compensate the loss caused by cross domain memory access collisions that otherwise would not occur in gang / hybrid algorithm with sync. idle option. The investigation is done by comparing the performance of gang/hybrid scheduling

algorithm with no synchronization idle with the performance of gang/hybrid scheduling algorithms with sync. idle.

The workload parameters used in this subsection:

- 1) $AET_s = AET_p$.
- 2) Blocking parameters: $Q_{pp}=0.036$, $Q_{np}=0.59$, $Q_{nn}=1$ but we do not count the commitment of gang process to exclude the effect of $Q_{nn}=1$.
- 3) Processor configurations: $PE=10$.

Table 4-15 shows the primary capacities of gang algorithm and hybrid algorithm with no sync. idle option, and gang/hybrid algorithms with synchronization idle option.

Table 4-16 Primary Capacity: No Sync. Idle option vs. Sync. idle option

Number of PEs	PE10
Gang algorithm without sync. idle	5.05
Gang algorithm with sync. idle	5.30
Performance comparison	-4.7%
Hybrid (s:g=1:1) without sync. idle	4.8
Hybrid (s:g=1:1) with sync. idle	4.90
Performance comparison	-2.0%

The performance of gang / hybrid algorithm with no synchronization idle option is lower than the performance of gang / hybrid algorithm with synchronization idle option, indicating that the system

would use more processing time than the synchronization idle to resolve the contention between primary processes and gang processes under high Qnp scenario.

4.6.3 Summary

The experimental results support the elimination of serial idle imposed on serial algorithm and hybrid algorithm. The elimination of serial idle makes better use of available multiprocessing power, and is able to compensate the loss of system processing time used for resolving the collision caused by the elimination of serial idle option for the scenario considered in Section 4.5.

For the scenario considered in Section 4.5, the experimental results do not support the elimination of synchronization idle imposed on gang algorithm and hybrid algorithm. The elimination of synchronization idle is not able to compensate the loss of system processing time used for resolving the collision caused by the elimination of the synchronization idle option.

Chapter 5

Conclusions

5.1 Summary

Memory access contention is one of the key problems that every shared memory multiprocessor faces. As discussed in Chapter 1, the most common way of dealing with memory access contention is disjointing shared data domains as much as possible. A second way is controlling the sharing of data in shared memory through appropriate scheduling mechanisms to avoid contentious processes being executed concurrently. Of these two approaches, controlling the sharing of data in shared memory is attractive especially when disjointing all of data domains is not feasible. Disjointing data domain requires re-design of existing applications, making the transition of legacy software written for single processor multitasking architecture to a multiprocessor parallel processing architecture difficult to realize at affordable cost. Controlling the sharing of data through scheduling may require only some change of operating system, keeping the vast amount of application software unchanged. This thesis has focused on the study of the scheduling algorithms that intend to control the sharing of data by avoiding contentious processes executing concurrently, on a shared memory multiprocessor based telephone switch. A trace driven simulation model has been implemented for investigating the performance of a number of scheduling algorithms. The key results of this thesis are presented.

1. As pointed out in Section 4.2, a capacity improvement of even a few percent is of great value to a telephone switch builder. This thesis demonstrates that controlling the concurrency of contentious processes by employing a proper scheduling algorithm can improve system capacity significantly. For example, with high Q_{np} , the capacity improvements produced by both hybrid and gang scheduling algorithms over the serial scheduling are 10% approximately.
2. For the scenarios investigated, Q_{nn} has a minor impact on the capacity of switch. This is because secondary domain gets a small portion of the machine (roughly 20% on DMS switch), and the potential blocking between secondary processes is small (Q_{nn} ranges from 0.079 to 0.14 only). Therefore disjointing secondary domain processes is not necessary in the early project development.
3. Hybrid algorithm demonstrates the characteristic of robustness in the presence of dynamic changes in Q_{np} and Q_{nn} . Gang algorithm performs the best when Q_{np} is high, but is sensitive to a change in Q_{nn} . Serial algorithm is not affected by a change in Q_{nn} but performs the worst when Q_{np} is high.
4. Overheads associated with a scheduling algorithm are important and affect system capacity. In order to control the concurrent executions of contentious processes, a scheduling algorithm may force processors to idle. Serial idle and synchronization idle are two types of scheduling overhead incurred in serial algorithm and gang algorithm respectively. As the systems grow, both serial idle and synchronization

idle increase. Up to 14% serial idle has been observed from the experiments. This led to the investigation of serial algorithm with no wait option. Serial algorithm with no wait option tries to eliminate the serial idle by scheduling in a primary process, and improves system's capacity by up to 17% over serial algorithm with wait option.

5.2 Contributions

This thesis contributes to the state of the art for controlling the sharing of data in a shared memory multiprocessor based system. The important contribution are summarized.

1. The thesis presents a comparative analysis of the performance of three scheduling algorithms for a multiprocessor based telephone switch under various contention scenarios.
2. Through the trace driven simulation, this thesis demonstrated the effectiveness of gang algorithm and hybrid algorithm in controlling memory contention in a shared memory multiprocessor based telephone switch.
3. The thesis provides a detailed analysis of the scheduling algorithm associated overheads, and estimates the effect of these overheads on system performance. It demonstrates that a serial algorithm with no wait (no serial idle) option can lead to a significant capacity improvement over the serial algorithm with wait option.
4. The research provides an understanding of the key factors that affect the performance of the scheduling algorithms. These factors include the number of PE's, average work time of transactions, and memory contention levels.

5.3 Future work

This thesis indicates that process scheduling is an effective way for controlling shared memory contention without incurring excessive application layer code re-writing. It hides the complexity of parallel programming from application layer by not allowing contentious processes executing concurrently. Investigating the performance of the scheduling algorithms on the real telephone switch is important.

All computing systems are continually challenged to deliver higher processing capacity for improving the performance of existing applications and realizing new applications of advanced services. To meet the challenge many manufactures are moving towards multiprocessing, an alternative especially suitable for memory access intensive system in which higher speed single processor would suffer from slow memory transactions. As stated in [Stenstorm97], shared memory multiprocessors promise natural evolution for extending a single processor based legacy system to a multiprocessor system. However, it is also likely to give rise to memory contention that was observed on the telephone switch. Extending the scheduling algorithms, discussed in this thesis, to other systems such as main memory based database forms a direction for future research in shared memory multiprocessor area.

References

- [Almasi94] G. S. Almasi and A. Gottlieb, "Highly Parallel Computing", Second Edition, Benjamin/Cummings, 1994.
- [Baker95] B. Baker and T. Newell, "Shared Memory Multi-processor Proposal for DMS", Nortel Networks, 1995.
- [Benniger98] B. Benniger, "DMS SuperNode XA-Core Guide to Multiprocessing", Nortel Networks, 1998.
- [Carroll98] B. Carroll, Nortel Networks, Personal Communication, September, 1999.
- [Catanzaro92] B. Catanzaro, "Multiprocessor System Architectures", Prentice Hall, 1992.
- [Drweiga96.1] T. Drweiga, "Shared memory Contention and its Impact on Multiprocessor Call Control Throughput", Proc. 15th International Teletraffic Congress, June 1997, pp. 22-27.
- [Drweiga97] T. Drweiga, "Formulas for Calculating Qnp", Nortel Networks, 1997.
- [Drweiga96.2] T. Drweiga, "Calculating Degradation by Memory Contention between Non-Payload and Payload", Nortel Networks, 1996.
- [Dandamudi97] S. P. Dandamudi "Reducing Hot-Spot Contention in Shared Memory Multiprocessor Systems", Technical Report TR-97-17, School of Computer Science, Carleton University, 1997.
- [Feitelson94] D. Feitelson "A Survey of Scheduling in Multi-programmed Parallel Systems", Research Report RC19790 (87657), IBM T.J. Watson Research Center, New York, 1994.
- [Feitelson96] D. Feitelson "Packing Schemes for Gang Scheduling", in job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, Springer Verlag, 1996, pp. 89-110.
- [Jain91] R. Jain, "The Art of Computer Systems Performance Analysis", John Wiley & Sons, 1991.
- [Kolski97] S. Kolski "Determining Data Domain Overlap-Analysis of GSM and NA100 Traffic Data", Nortel Networks, 1997.

- [Lenoski95] D. E. Lenoski, "Scalable Shared-Memory Multiprocessing", Morgan Kaufmann Publishers, 1995.
- [Lilja94] D.J. Lilja, "The Impact of Parallel Loop Scheduling Strategies on Prefetching in a Shared Memory Multiprocessor", IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 6, June 1994, pp. 573-584
- [Motorola96] "The Programming Environments for 32-bit Microprocessors", Motorola, 1996.
- [Majumdar99.1] S. Majumdar, D. Streibel, B. Benniger, B. Carroll, N. Verma, M. Liu, "Controlling Memory Contention on a Scalable Multiprocessor-Based Telephone Switch", Proc. ACM SIGMETRICS '99 Conference, (Extended Abstract), Atlanta, May 1999, pp. 228-229.
- [Majumdar99.2] S. Majumdar, "Scheduling Strategies for Concontrolling Resource Contention of Multiprocessor Systems", Proceedings of the 6th International Conference on High Performance Computing (HiPC'99), Calcutta, India, 1999, pp. 151-157.
- [Nanda96] A. Nanda and L. M. Ni, "MAD Kernels: "An Experimental Testbed to study Multiprocessor Memory System Behavior", IEEE transactions on parallel and distributed system, Vol. 7, No.2, February, 1996, pp. 207-215.
- [Nortel98] "DMS Programming Model", Nortel Networks, 1998.
- [Nortel94] "DMS SuperNode", Training Notes, Nortel Networks, 1994.
- [Nortel96] "PROPHET User Guide and Reference Manual", Release 2.0.3, Document Issue 1.3, Nortel Networks, 1996.
- [Silberschatz99] A. Silberschatz and P. B. Galvin, "Operating Systems Concepts (Fifth Edition)", Addison-wesley, 1999.
- [Stenstrom97] P. Stenstrom, E. Hagersten, D. J. Lilja, M. Martonosi, M. Venugopal, "Trends in Shared Memory Multiprocessing", IEEE Computer, Vol. 30, No.12, December, 1997, pp. 44-50.
- [Sequent99] "Requirements and Performance of Enterprise Computer Solutions: SMP, Clustered SMP, and MPP", Sequent Computer Systems, Inc., 1999.
- [Stallings98] W. Stallings, Operating Systems (Third Edition), Prentice Hall, 1998.

- [Streibel97] D. Streibel, "Simulation Study of a Single Class Multiprocessor System", Nortel Networks, 1997.
- [Torrie96] E. Torrie, M. Martonosi, C.W. Tseng, M.W. Hall, "Characterizing the Memory Behavior of Compiler-Parallelized Applications", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No.12, December, 1996, pp. 1224-1229.
- [Thomasian93] A. Thomasian, "Two-Phase Locking Performance and its Thrashing Behavior", ACM Transactions on Database Systems, Vol.18, No.4, December, 1993, pp. 579-625.
- [Tay85] Y. C. Tay, R. Suri and N. Goodman, "A Mean Value Performance Model for Locking in Databases: The No-Waiting Case", Journal of the Association for Computing Machinery, Vol.32, No.3, July 1985, pp. 618-651.
- [VanderSteen96] A. J. VanderSteen and J. J. Dongarra, "Overview of Recent Supercomputers", NHSC Review, 1996 Volume, First Issue, July 1996, <http://www.crpc.rice.edu/NHSEreview/ORS/>
- [Wilson96.1] P. Wilson, "Multi-processing Development Environment", Architecture Document", Nortel Networks, 1996.
- [Wilson96.2] P. Wilson, "Call Processing User Guide for XA-Core", Nortel Networks, 1996.