Scheduling Issues in Real-Time Systems [1]

Chia-Mei Chen
Institute for Advanced Computer Studies
Systems Design and Analysis Group
Department of Computer Science
University of Maryland, College Park, MD 20742

June 19, 1995

[1]This work is supported in part by ARPA and Philips Labs under contract DASG-92-0055 to Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, PL, or the U.S. Government.

Chapter 1

Introduction

Real-time systems (RTS) have drawn a lot of attention and become an active area of research, because of the importance of their applications, such as defense, avionics, robotics, industrial automation, and stock trading. In particular, one of the most intensive research area in RTS is the domain of scheduling [AGM88, Bur90, CJL89, CL90b].

The result of a real-time application is valid only if the application functions correctly even with underlying faults and its real-time constraints are satisfied. Besides real-time constraints, tasks usually have resource requirements, such as exclusive access of certain resource or inter-task communication. Some real-time applications may have intensive data processing; we call such system real-time database system (RTDBS) and the application running on RTDBS is called transaction. Transactions must have data access requirements because of their nature. Besides resource requirements, transactions also have real-time constraints.

Scheduling tasks in RTS or scheduling transactions in RTDBS needs to consider real-time constraints, resource requirements, and fault-tolerance goals. In this dissertation, we address some scheduling issues, namely, scheduling tasks with resource requirements in RTS, scheduling transactions with resource requirements in RTDBS, scheduling tasks with fault-tolerance goals.

1.1 Motivation and Approaches

To schedule tasks with resource requirements in HRTS, resource synchronization or control protocol is employed to permit exclusive access to shared resources, while preventing deadlocks and guaranteeing that timing constraints are satisfied.

One approach to synchronization involves extending priority-driven protocols. In this class of protocols, each task has an associated priority that is used to determine access to shared resources (including the processor). When synchronization is permitted, priority-driven protocols are susceptible to potentially unpredictable delays due to priority inversion. Priority inversion [SRL87] occurs where a higher priority task is forced to wait for lower priority task. Some amount of priority inversion is unavoidable to guarantee mutual exclusion; however, it must be bounded to allow schedulability analysis and minimized to improve processor utilization bounds.

With the exception of Multiprocessor Priority Ceiling Protocol (MPCP) [RSL88], most resource synchronization protocols, in the context of preemptive priority-driven scheduling, have been developed solely for uniprocessor systems. MPCP does not allow nested accesses to global resources, i.e., it does not allow a task to simultaneously lock more than one global resource. A global resource is one that may be accessed by tasks assigned to different processors. This limitation on the use of global resources may not satisfy varying resource access requirements, and may lead to unnecessary blocking. For example, some tasks may only need access to a small unit of global data, while other tasks may need to lock the entire resource. With MPCP, all tasks are forced to lock the entire global resource to guarantee consistency. The situation is analogous to using file locking, when record locking would suffice. We know that a finer granularity of synchronization allows a greater degree of concurrency, while coarser granularity imposes less overhead. A balanced application of fine granularity can gain the advantages of parallelism in return for a reasonable overhead cost. The insights motivate us to propose a priority ceiling based resource synchronization protocol, for multiprocessor HRTS, that allows a task to simultaneously lock multiple global resources.

Scheduling algorithms for data intensive real-time applications (transactions) need to accompany with concurrency control to enforce data consistency and to satisfy timing constraints simultaneously. We call such system real-time database systems (RTDBS). One major objective in traditional database systems is to minimize the average response time of a transaction, while in RTDBS minimizing the number of transactions that miss their deadlines is a major concern. Transaction scheduling and concurrency control in RTDBS try to fulfill this objective while maintaining data integrity and consistency.

Priority-driven concurrency control methods usually favor high priority transactions. However, higher priority transactions do not necessarily have urgent timing constraints. For example, suppose a system predefines that every occurence of transaction H has higher priority than that of transaction X and both have the same relative deadline. Suppose an occurence of X arrives earlier than that of H. Therefore, low priority transaction X has tigher timing contraint than H. It might be the case that low priority transaction X with a tight timing constraint conflicts with higher priority transaction H with a looser timing constraint and has to wait or restart. After all, the low priority transaction X might miss its deadline. Let E_A and D_A be the execution time and the deadline of transaction A and L_A be the latest time to start execution of A, i.e. $L_A = (D_A - E_A)$. A transaction A is said to have a tighter timing constraint than transaction B, or be more urgent than B, if L_A is less than L_B, i.e, the latest time to start execution of A is earlier than that of B. A transaction A is *restartable* if its latest time to start execution is later than the current time. By the concept of *restartability*, we can discard late (non-restartable) transactions before their deadlines expire and save resources. These observations motivate our research to examine if the knowledge of execution time can help us make better data resolution decision.

The deployment of execution time information to improve the performance of optimistic concurrency control algorithms is not trivial. One possibility is to give preferential treatment to short transaction in data conflicts, for example, waiting for short transactions to commit if the validating transaction conflicts with short transactions. However, for the transaction with urgent timing constraint, such waiting might cause it to miss the deadline. Therefore, the number of transactions that miss their deadlines might increase. Besides, giving preferential treatment to short or long transactions might not satisfy certain properties, such as fairness and freedom of starvation. Therefore, this research studies the significance of the knowledge of the execution time in optimistic concurrency control.

RTS should perform applications correctly and meet the deadline requirements even with underlying failures. Hence, fault-tolerance is an inherent requirement for RTS. Faults can be classified as hardware and software faults. A software fault refers to a design or coding fault in a software system; a hardware fault refers to any deviation of a machine state from correct state. Hardware faults can be caused by transient disturbance, environmental disturbance[1] [KS89] and permanent failure of a component. The fault model for our research considers transient and permanent faults.

The simplest way of achieving fault-tolerance is to use fully redundant hardware or to replicate all the tasks in the system. These naive approaches increase the communication cost and need a voting mechanism to get the results of execution. Hence, the overhead and the resource usage are quite large in these cases. Besides, some systems might not have enough resources to replicate all tasks, or does not have fully redundant hardware architecture. But, they have fault-tolerance requirement. In non-real-time systems, a task can rollback many times to skip a transient fault, since it does not have timing constraint. Tasks in RTS might not have such luxury to roll back several times or even once. We are interested in static fault-tolerant scheduling which can pre-analyze the execution profiles of tasks. We may only need to replicate some tasks which have stringent timing constraints and can achieve very high degree of system reliability.

1.2 Contributions

To summarize, the major contributions of this dissertation are:

- We present resource synchronization protocols for multiprocessor HRTS and the simulation results show that the performance of the proposed protocols is superior to that of the existing protocols.
- We study the performance of optimistic concurrency control protocols with the knowledge of execution time and the simulation results illustrate that proper use of execution time information can improve the system performance.
- We present a static fault-tolerant scheduling algorithm which adaptively and dynamically puts temporal and spatial redundancy into the schedule in order to improve

[1] Environmental disturbance, such as electromagnetic noise and radiation, often cause correlated transient failures

system reliability.

- We utilize Markov chain models to estimate reliability for RTS using static scheduling and demonstrate the correctness of the proposed reliability models by simulation.
- We demonstrate that our proposed fault-tolerant scheduling method provide more reliable systems than the basic fault-tolerant scheduling schemes, replication and rollback.

1.3 Organization of the Dissertation

Chapter 2 is devoted to an overview of issues related to scheduling in RTS. This chapter contains a survey of published research on the subject of scheduling algorithms and synchronization protocols for RTS or RTDBS. Chapter 3 presents the proposed resource synchronization protocols for RTS and evaluates the performance of the proposed and existing protocols. Chapter 4 presents a set of optimistic concurrency control protocols using the knowledge of execution time. In this chapter, we give a review on the mechanism of optimistic concurrency control and the foundmental structure of the proposed protocols. We also demonstrate the significance of the knowledge of execution time in improving system performance. In Chapter 5, we propose a fault-tolerant scheduling algorithm. An abstract system model, applicable to the proposed scheduling algorithm and reliability models, is stated. The reliability models used for estimating system reliability are presented in Chapter 6. In Chapter 7, we present the basic fault-tolerant scheduling algorithms which are used as the baseline for evaluating the performance of the proposed scheduling algorithm. Then, we evaluate and compare the performance of the proposed and basic scheduling algorithms. Finally, Chapter 8 summarizes the results obtained in this dissertation and outlines future avenues to explore.

Chapter 2

Related Work

The objective of RTS design is to produce outputs correctly and within deadlines, in spite of occasional uncertainty. Most of research in real-time scheduling has been based on the assumption of periodic workload with static characteristics. Scheduling can be carried out off-line or on-line. Off-line (static) scheduling can provide predictable execution behavior for a fixed set of tasks; while on-line scheduling can jointly accept periodic and aperiodic tasks. To schedule tasks with hard deadlines, off-line scheduling is a frequent approach to meeting timing constraints. For a system with dynamic workload, on-line scheduling is a common proposal, which can adaptively adjust the system workload and reject tasks when their constraints cannot be met.

In order to provide predictable behavior for real-time processing, the requirements or constraints of the tasks must be known *a priori*. For scheduling tasks with exclusive accesses to shared resources, resource control protocol is deployed to coordinate the accesses of resources in maintaining data consistency and deadlock avoidance. For scheduling tasks with fault-tolerance requirements, fault-tolerance scheme is used to improve system reliability. Our research focuses on the issues on scheduling tasks with resource or fault-tolerance requirements. Hence, in this chapter, we review published work in real-time scheduling which considers resource synchronization, concurrency control, and fault-tolerance.

2.1 Scheduling with Resource Synchronization

It is a fundamental feature for RTS to allow a set of tasks to exclusively access physical or logic shared resources. Real-time applications are hard to design on a system without such feature. For preemptive scheduling, one frequently used strategy for resource control, to have predictive execution behavior, is priority-driven protocols. This class of protocols employs blocking to enforce synchronization. The advantages of priority-driven protocols are that theoretically they provide worse case bounds for blocking time and schedulability tests; the shortcomings are that they are conservative in estimating blocking time and ignore the overhead of context switches. The former shortcoming is the price we need to pay for schedulability tests. There is rare work on the issue of estimating the impact of context switches.

Sha *et al.* introduced the concept of priority inheritance protocols to solve the priority inversion problem [SRL87]. One of the more attractive protocols they proposed, the priority ceiling protocol (PCP), prevents both deadlock and transitive blocking. Transitive blocking occurs where a job J is blocked by job J_i which is in turn blocked by another job. They also developed sufficient schedulability conditions for a set of periodic tasks to be scheduled via the PCP algorithm on a uniprocessor system. Rajkumar *et al.* subsequently developed multiprocessor and distributed versions of PCP [RSL88].

Chen and Lin developed the dynamic priority ceiling protocol (DPCP) to enhance EDF scheduling algorithm [CL90a]. Baker proposed a stack-based resource allocation policy (SRP) which can be applied to either RM or EDF scheduling algorithms [Bak90]. PCP for multiple-instance resources is also developed in [CL91]. Chen and Lin summarized the schedulability conditions of several priority-driven protocols, and proposed a set of sufficient schedulability conditions for EDF-based resource control protocols [CL90b].

A review of resource control for HRTS is published in [Aud91]. It identifies the possible techniques for uniprocessor and multiprocessor systems.

For non-preemptive scheduling, mutually exclusive access of shared resources is solved implicitly, since the access of shared resources will not be interrupted. Zhao and Ramamritham proposed an on-line scheduling algorithm based on branch and bound approach to scheduling tasks that arrive dynamically and have timing and resource constraints [ZR87]. Heuristics were devised to shrink the search space. Off-line scheduling algorithms can also be developed by means of branch and bound approach. Xu and Parnas proposed such algorithm for finding an optimal schedule on uniprocessor systems [XP90]. Herhoosel and Hammer considered the problem of scheduling periodic tasks which have deadlines, resource requirements, and precedence constraints in distributed HRTS [VLH91]. Instead of using branch and bound approach, they adopted heuristic backtracking for searching feasible schedule, since the search space is large.

2.2 Scheduling with Concurrency Control

Scheduling in RTDBS needs concurrency control protocol to enforce data consistency and to satisfy timing constraints. Lin and Son gave a survey on the issues of schedulability and serializability in RTDBS [LS93]. Since the objectives of RTDBS are different from those of conventional database systems, scheduling and concurrency control used for conventional systems should be re-evaluated and might need to be extended for RTDBS. Several research efforts have been carried out in evaluating the performance of concurrency control and the impact of real-time constraints in RTDBS [AGM88, AGM89, HSTR89, Son91]. Buchmann *et al.* presented a framework for integrating real-time scheduling and concurrency control and a summary of real-time concurrency control algorithms [BMHD89].

Several priority ceiling based protocols were proposed; these protocols provide bounded blocking times imposed by accessing data objects and schedulability tests for scheduling a set of periodic tasks in uniprocessor RTDBS [SRSC91, Nak93]. Haritsa *et al.* proposed a priority-based optimistic concurrency control algorithm WAIT-50 [HCL90] which performs better than the traditional optimistic algorithm, OPT-BC [MN82, Rob82]. They concluded that priority information can improve the performance of optimistic concurrency control in RTDBS. Ulusoy proposed several locking and timestamp based concurrency control protocols for RTDBS [Ulu92]. He also developed single-site and distributed RTDBS models for evaluating the proposed protocols.

Most of the proposed real-time concurrency control protocols mentioned above incorporate priority information into data resolution decision. None of the aforementioned work makes use of execution time information to support the decision policies of concurrency control. Our research investigates the impact of execution time on data resolution decision of concurrency control.

2.3 Scheduling with Fault-Tolerance

Rollback and redundancy are dominant approaches to fault-tolerance. Rollback schemes often tightly connect to the checkpoint insertion methods. Several researchers have investigated the problem of selecting a checkpoint interval which is optimal with respect to a certain objective [CR72, Upa90, US86, Gel79, NK83, GRW88].

The earliest attempt to obtain highly reliable system was through redundancy [Neu56]. Redundancy schemes use several identical components operating in parallel and usually need a voting mechanism to get the results of execution. Redundancy schemes can be classified as hardware redundancy and software redundancy. Triple modular redundancy (TMR) is an example of hardware redundancy [Mar67, Pie65]. A slight different way of hardware redundancy is a hybrid system which uses TMR and standby spares switched in when needed [MA70]. Much work has been done on fault-tolerant architectures using redundancy strategy. Many of the techniques required either a number of spare processors [BB87, Bat80, BCH91, KJC89, Ros92, RBK90], or a switching mechanism assumed to be immune to faults [BB87, Bat80, BCH91, RBK90]. Walter *et al.* presented a multicomputer architecture for fault-tolerance (MAFT) which was designed for RTS requiring both high performance and reliability [WKF85]. System overhead tasks are executed at each node by a special purpose device, operations controller (OC), and the application tasks are performed on application processors (AP). A number of APs are connected together through OCs. Multiple copies of a task can run on different APs; OCs will take care of consistency and communication problems.

Software redundancy is to have multiple copies or versions of a software program running on different processors. Chen and Cherkassky devised k-circular shifting algorithm to allocate tasks to processors statically and redundantly, so that if some processors fail during the execution all tasks can be completed on the remaining processors [CC90].

Rollback strategy can be considered as temporal redundancy in which multiple copies of a task are run in different time intervals. Typically transient errors subsides quickly; one rollback often can skip an independent transient error. A permanent error might be detected by encounting the same error on multiple retries. Usually a task does not roll back from the beginning of the task, it rolls back to the most recent checkpoint and consumes resources only when it needs to. However, for redundancy strategy, multiple copies of a task run and consume resources regardless of errors. Hence, rollback strategy can save more resources than redundancy strategy. In another point of view, rollback strategy can accept larger workload. Rollback technique has time and space overheads for saving and reloading system state, while redundancy techniques often have communication overhead and time overhead on voting to get the results of execution. Redundancy strategy is more deterministic in which redundant copies run no matter what happens. Hence, such system has more predictable behavior and it is easy to meet timing constraints in RTS.

These two strategies have their own advantages and disadvantages, usually they are complementary. Few efforts have been made on combining these two strategies to get a reliable real-time system with the guarantee of satisfying timing requirements. This research attempts to develop a hybrid technique to extract the benefits of both to build up a more reliable system than that using only rollback or redundancy.

Chapter 3

Resource Synchronization Protocols

Scheduling tasks with needs to exclusively access some resources must consider resource synchronization problem. The common approach to synchronization in real-time systems (RTS) is priority-driven protocols. The order of accessing resources is according to the priority of the task, that is, higher priority task gets the priority to access resources. Priority inversion [SRL87] happens when a higher priority task tries to access a resource which is currently accessing by a low priority task. The higher priority task has to be blocked in order to maintain data integrity. Such blocking causes discontinuity in scheduling, results in unpredictable behavior, and degrades schedulability. The proposed protocol tries to minimize the priority inversion in order to obtain higher schedulability and system throughput.

In this chapter, we propose a synchronization approach based on priority ceiling protocol for multiprocessor hard real-time systems (HRTS). The proposed approach can be used to enhance rate monotonic (RM) or earliest deadline first (EDF) scheduling algorithms. In addition, we extend MPCP [RSL88] to an EDF-based resource synchronization protocol. We present the results of performance analysis of the proposed approach and MPCP and show that the proposed approach improves schedulability. This improvement is due to the fact that our approach allows a greater degree of parallelism.

In the next section, we state the assumptions of the proposed approach and present the notation used in the rest of the chapter. Section 3.2 presents a new version of multiprocessor synchronization protocol, along with its properties and schedulability analysis. MPCP is investigated and extended in Section 3.3. Section 3.4 compares the performance of the proposed approach and the existing approach.

3.1 Overview and Notation

Let $\mathcal{P}_1$,$\mathcal{P}_2$, ...,$\mathcal{P}_m$ be the processors of the system. Each task is assigned to a specific processor. A resource is any object that requires serialized access. Each resource is associated with a binary semaphore which is used to guarantee mutual exclusion. A resource may be either global or local. Global resources can be accessed by tasks assigned to some (possibly complete) subset of the processors, while local resources are only accessible to the tasks on that particular single processor. A set of n_i periodic tasks is associated with each processor $\mathcal{P}_i$. Each task T can be described by a triple (w, e, L), where w is the period of the task; e is the execution time of the task; and L is a list of resources accessed by the task. Access to a shared resource occurs within the corresponding critical section (i.e. a sequence of instructions preceded by a lock operation of the associated semaphore and followed by an accompanying release operation). To distinguish between critical sections that access global resources and those that access local resources, we call the former global critical sections and the latter local critical sections. Furthermore, semaphores that guard global resources are called global semaphores and those that guard local resources are called local semaphores. We use the terms resource, critical section, or semaphore interchangeablely, depending on the context.

For each task, an instance of the task, called a job, is generated for every period of the task. The release time of a job is the beginning of the period and the deadline of a job is the end of the period. $P_t(J)$ denotes the priority of the job J at time t and $P(J)$ denotes the priority of the job J at the current time. $C_t(S)$ denotes the priority ceiling of the semaphore S at time t and similarly $C(S)$ refers to the priority ceiling of the semaphore S at the current time. When priorities are static, the subscript t is dropped. The remote processors of processor $\mathcal{P}$ are the processors which share global resources with $\mathcal{P}$. Only jobs assigned to remote processors of processor $\mathcal{P}$ can interfere with the jobs on processor $\mathcal{P}$. Let J be assigned to processor $\mathcal{P}$. We define the remote jobs of J, or of $\mathcal{P}$, as the jobs assigned to the remote processors of processor $\mathcal{P}$, and similarly the local jobs as the jobs assigned to $\mathcal{P}$. Jobs $J_1, J_2, \ldots, J_n$ are listed conventionally in descending order of priority with J_1 having the highest priority, i.e., $P(J_1) > P(J_2) > \ldots > P(J_n)$. The jth critical section of job J_i is denoted by $z_{i,j}$.

Even though the priority ceiling protocol (PCP) [SRL87] is based on RM scheduling and the dynamic priority ceiling protocol (DPCP) [CL90a] is based on EDF, the concepts underlying both protocols are similar, varying primarily in their definitions of priority. Both protocols assign priority ceilings to semaphores which are used to defer some requests that could potentially be granted. The purpose of deferring some requests is to reduce and bound blocking due to priority inversion. The priority ceiling of a semaphore S, $C(S)$, is defined as the maximum priority of all jobs that are currently locking or will lock S. For a particular job J, S^* denotes the semaphore with the highest priority ceiling that is currently locked by a job other than J. Whenever a job J wants to access a resource, it must first acquire an exclusive lock on the semaphore associated with that resource. The lock is granted only if $P(J) > C(S^*)$; otherwise job J is blocked until the lock may be granted. The job holding semaphore S^* inherits the priority of the highest priority job that is blocked by S^*. When a job exits from a critical section, it releases the semaphore, and the highest priority job waiting for the semaphore can then lock the semaphore. There are two types of blocking[1]. A job J is blocked if it attempts to lock some semaphore S, while some lower priority job J_L has locked a semaphore S' whose priority ceiling exceeds the priority of J, $C(S') \geq P(J)$.

[1] We use the term blocking to refer to situations when a higher priority job is temporarily denied a resource (including the processor) due to a lower priority task.

The other form of blocking occurs when there is a higher priority job J_H that is already blocked due to some lower priority job J_L. Though the concepts do not change substantially, these protocols require careful modification to be extended to multiprocessor systems.

3.2 Multiprocessor Dynamic Priority Ceiling Protocol C (MDPCP-C)

In this section, we present a dynamic priority multiprocessor version of the priority ceiling protocol (MDPCP-C) based upon EDF scheduling. The protocol imposes a few restrictions which we believe are often acceptable. MDPCP-C only allows global (local) critical sections to be nested within other global (local) critical sections; in other words it is not possible for a job to simultaneously lock both a global and a local semaphores. An outermost global critical section and its nested inner global critical sections are viewed as a unit and can be shared by a group of processors. Any global semaphores that are ever locked simultaneously by the same job, (i.e. that ever appear in the same nested critical section), must be shared by exactly the same set of processors. If a global semaphore S is shared by processors $\mathcal{P}_i$ and $\mathcal{P}_j$, we say that S is a global semaphore common to $\mathcal{P}_i$ and $\mathcal{P}_j$. In the following subsections, we present the fundamental concepts behind MDPCP-C, give proofs of some of its useful properties, and analyze conditions under which a feasible schedule may be assured.

3.2.1 Basic Idea Behind MDPCP-C

Rajkumar *et al.* defined remote blocking as the blocking caused by remote jobs, regardless of their priorities [RSL88]. They then proved that the remove blocking time of a job blocked while attempting to enter a global critical section is a function of the access time of critical sections if and only if a job within the global critical section cannot be preempted by jobs executing outside critical sections. To ensure that blocking times are predictable, MDPCP-C will not allow any job to preempt a job executing within a global critical section. Since local critical sections may not overlap or nest with global critical sections, we can use DPCP to synchronize access to local resources. Events which may affect global resources such as locking or releasing a global semaphore or the arrival of a new job require more attention as will be discussed shortly. Recall from the previous section that the priority ceiling of a global semaphore S, $C(S)$, is the priority of the highest priority job that is currently holding or will hold the semaphore S at the current time. $C(S)$ may vary with time as priorities are reassigned according to the EDF scheduling discipline. Let job J be bound to processor $\mathcal{P}_j$.

1. The highest priority job eligible to execute on processor $\mathcal{P}_j$ is assigned the processor if no local job with lower priority is in a global critical section.

2. Before a job J enters a global critical section, it must lock the associated semaphore S. Let $\mathcal{SS}_j^*$ denote the *set* of global semaphores accessible from processor $\mathcal{P}_j$ that are currently locked by remote jobs of J. Job J is granted the lock and may enter the

critical section if it satisfies the locking condition:

$$P(J) > \max_{s \in \mathcal{SS}_j^*}(C(s)).$$

Otherwise, it is blocked and joins the queue for semaphore S. The queue is priority-ordered, i.e., the job with the highest priority waiting in the queue locks the semaphore when it is released.

3. If a job J is blocked and it has not locked any global semaphores, then a remote job J_r with the priority lower than J may lock a global semaphore whose priority ceiling is greater than or equal to $P(J)$ only if J_r was executing within a global critical section when J blocked. This restriction holds even if the remote job satisfies the locking condition described in rule 2.

4. A job J uses its original priority, unless it is in a critical section and blocks higher priority jobs. In that case, it inherits the priority of the highest priority job that it blocks. The original priority is restored upon exiting the critical section.

MDPCP-C gives rise to four distinct types of synchronization delay: indirect and direct blocking, remote and implicit preemption. If a job J_L is in a global critical section, it will block other local jobs with higher priorities (see rule 1). We use the term *indirect blocking* to refer to such blocking. The blocking enforced by the locking condition described in rule 2 is called *direct blocking* or remote preemption, especially when the blocking is caused by higher priority jobs. The blocking enforced by rule 3 is called implicit preemption. Note that the blocking caused by synchronization of local resources is also called direct blocking since the uniprocessor protocol uses the same locking condition described in rule 2.

Rule 1 implies that at most one job on each processor can be within a global critical section. Hence, each semaphore in $\mathcal{SS}_j^*$ must be locked by a remote job of the blocked job. In addition, the priority of a job that has locked a semaphore in $\mathcal{SS}_j^*$ may be either higher or lower than that of the blocked job.

Because of rule 2, a job J may be remotely preempted by a higher priority remote job. For example, suppose J_r is a remote job of J with higher priority than J and both are waiting for the same global semaphore S_g. J_r will lock the semaphore S_g before J, since its priority is higher than that of J. J is directly blocked by J_r when J_r locks S_g. We call this special case of direct blocking *remote preemption.* In uniprocessor systems, conventionally, blocking occurs when a job is blocked by a lower priority job. Therefore, to conform with the definition of blocking used in uniprocessor protocols, we will in the future only use the term direct blocking when discussing blocking caused by lower priority jobs. We will refer to blocking due to a higher priority remote job as remote preemption.

Implicit preemption ensures that a higher priority job will not be infinitely blocked by lower priority remote jobs. The following two examples show the need for implicit preemption.

Figure 3.1 shows the configuration of the system used in the next two examples. $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ are the processors of the system and S_1, S_2, and S_3 are the global semaphores.

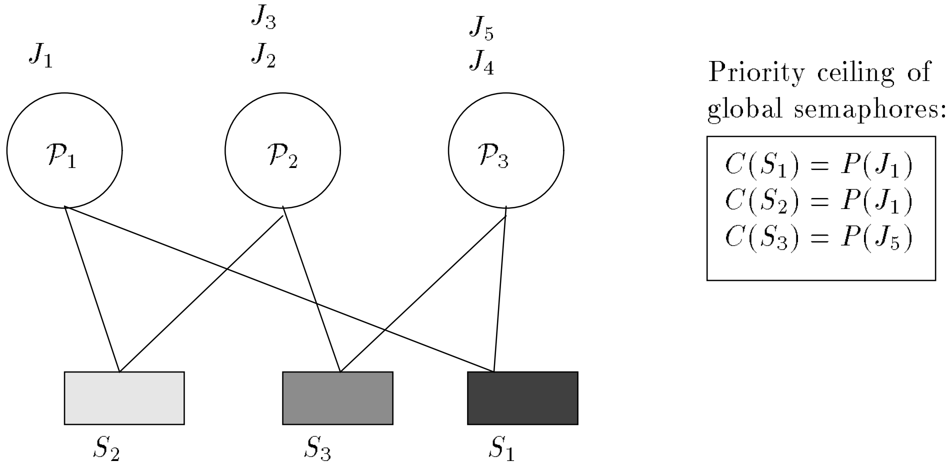


Figure 3.1: Architecture used in examples.

(We don't show any local semaphores.) S_1 is accessible from $\mathcal{P}_1$ and $\mathcal{P}_3$. S_2 is accessible from $\mathcal{P}_1$ and $\mathcal{P}_2$, and S_3 is accessible from $\mathcal{P}_2$ and $\mathcal{P}_3$. Job J_1 is assigned to processor $\mathcal{P}_1$, jobs J_2 and J_3 to $\mathcal{P}_2$, and jobs J_4 and J_5 to $\mathcal{P}_3$. S_1 is accessed by jobs J_1, J_4, and J_5, and S_2 is accessed by J_1, J_2, and J_3. No job accesses S_3.

Figure 3.2 illustrates how a job can be infinitely blocked by lower priority jobs, if we do not impose rule 3. Consider the following sequence of events. Suppose that J_1 attempts to lock global semaphore S_1 while J_2 has locked global semaphore S_2. In this case, J_2 directly blocks J_1. Before J_2 releases semaphore S_2, J_4 locks S_1, since $\mathcal{SS}_3^*$ is empty. Therefore, J_1 is directly blocked once again. The same scenario can happen on processor $\mathcal{P}_2$. Before J_4 releases S_1, J_3 locks S_2, and hence it blocks J_1. This sequence of events can repeat indefinitely, and cause J_1 to be infinitely blocked. Figure 3.3 shows how, under the same circumstances, job J_1 will not be infinitely blocked if rule 3 is enforced.

Synchronization of global resources contributes new blocking factors that do not arise in uniprocessor systems. To facilitate computation of the worse case blocking time induced by this protocol, we define two kinds of blocking sets: α and β. We also define three sets of jobs, $GP(J)$, $LP(J)$ and $LLP(J)$, useful in computing α and β. Each set of jobs contributes different synchronization delays.

Let $GP(J)$ be the set of remote jobs of job J whose priorities are higher than that of J, and let $LP(J)$ be the set of remote jobs of job J whose priorities are lower than that of J. Finally, $LLP(J)$ is defined to be the set of the local jobs of job J with priorities lower than J.

- $\beta_{i,L}$ denotes the set of the critical sections of the lower priority job J_L which can directly block J_i. $\beta_{i,LP(J_i)}$ denotes the set of critical sections of jobs in $LP(J_i)$ that can directly block J_i. $\beta_{i,LLP(J_i)}$ is defined similarly. Let β_i be the set of critical sections of all lower priority jobs of J_i that can directly block J_i. $\beta_i = \beta_{i,LP(J_i)} \cup \beta_{i,LLP(J_i)}$ and $\beta_{i,LP(J_i)} \cap \beta_{i,LLP(J_i)} = \emptyset$. Let $\mathcal{P}$ be the processor to which job J_i is bound. Elements

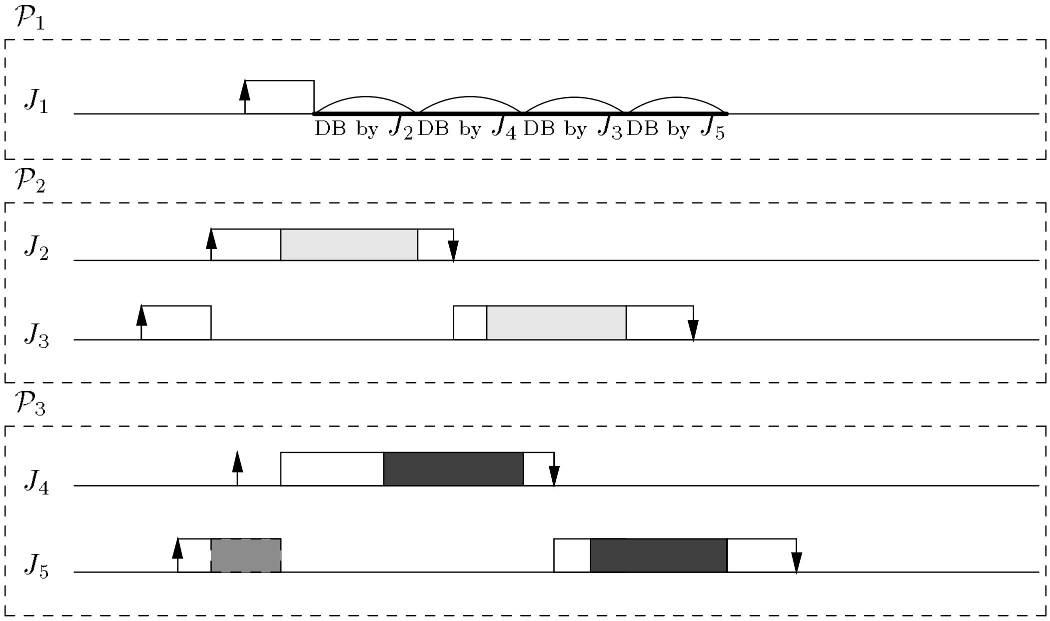


Figure 3.2: An example without implicit preemption.

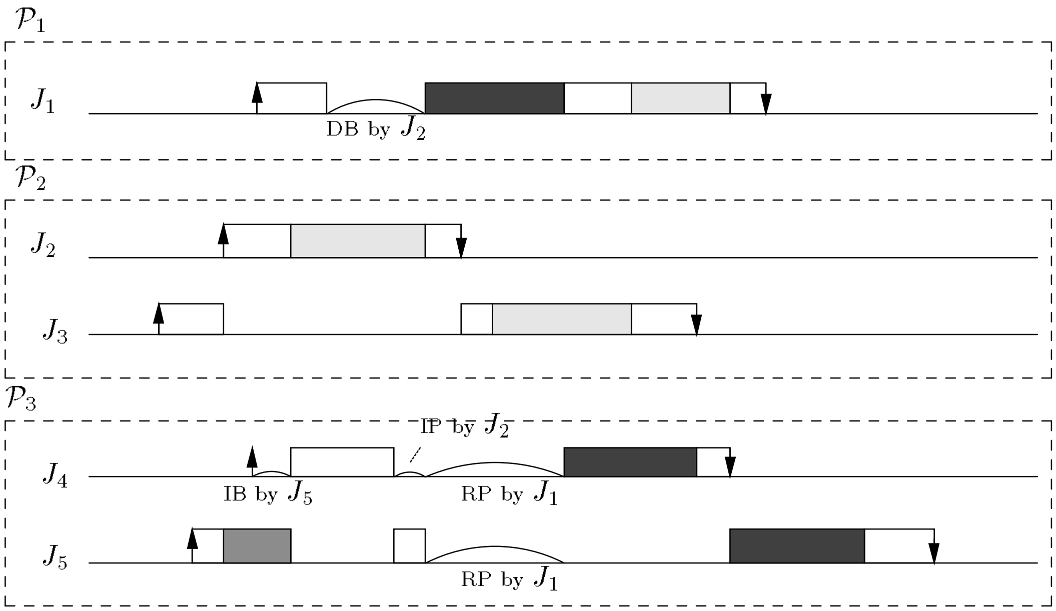


Figure 3.3: An example with implicit preemption. (IP = implicit preemption. RP = remote preemption.)

in $\beta_{i,LLP(J_i)}$ are the local critical sections of processor $\mathcal{P}$, while elements in $\beta_{i,LP(J_i)}$ are the global critical sections of $\mathcal{P}$.

- $\alpha_{i,L}$ denotes the set of all critical sections of local job J_L which can indirectly block J_i. α_i is defined similarly.

3.2.2 The Properties of MDPCP-C

Both DPCP and PCP prevent transitive blocking and deadlock. These useful properties are preserved in MDPCP-C.

Lemma 3.1 *Suppose J_H is directly blocked by a remote job J_L on global semaphore S. Then under MDPCP-C, J_H is not within any critical section.*

Suppose J_H is within a critical section guarded by S' when it is directly blocked by J_L on S. S' must be a global semaphore and S' and S are common to the processors to which J_H and J_L are bound. J_H must lock S' either before or after J_L locks S. (1) First, suppose J_L locks S after J_H has locked S'. Then $P(J_L) > C(S')$, and hence $P(J_H) > P(J_L) > C(S')$. However, $C(S') \geq P(J_H)$ due to the definition of priority ceiling. (2) Now, suppose J_H locks S' after J_L has locked S. Thus $P(J_H) > C(S)$. But since J_H is blocked by S, $P(J_H) \leq C(S)$. Since both cases lead to contradictions, the lemma follows.

□

Since global critical sections can be nested within other global critical sections, a job can be within several critical sections simultaneously. The above lemma implies that once a job enters an outermost critical section, it will not be blocked before it exits from that critical section. Hence, no matter how many nested critical sections are entered during the time the job is within the outermost critical section, the job will not be blocked by any lower priority job. Note that the job still can be preempted by a higher priority job.

Theorem 3.2 *MDPCP-C prevents transitive blocking.*

Let J_{i3} directly or indirectly block J_{i2} and suppose that J_{i2} directly or indirectly blocks J_{i1}. If J_{i3} directly blocks J_{i2}, by Lemma 3.1, J_{i2} is not within any of its critical sections. By the definition of MDPCP-C, J_{i2} cannot block any jobs indirectly or directly. If J_{i3} indirectly blocks J_{i2}, by the definition of MDPCP-C, J_{i2} has not started execution. Hence, J_{i2} cannot directly or indirectly block a job.

□

Theorem 3.3 *MDPCP-C prevents deadlocks.*

Suppose deadlock may occur and let $\{J_1, J_2, \ldots, J_n\}$ be a set of jobs that cause a waiting cycle. Since, by Theorem 3.2 there is no transitive blocking, at most two jobs can be in the waiting cycle. The rest of the proof is similar to the proof of Theorem 3.2.

□

3.2.3 Schedulability Analysis of MDPCP-C

In this subsection, we develop a set of sufficient conditions which, when satisfied, guarantee that m sets of periodic tasks assigned to m processors will complete execution within their periods when scheduled using MDPCP-C. Liu and Layland proved the schedulability condition for uniprocessor EDF scheduling [LL73]. A set of n periodic tasks can be scheduled by EDF algorithm if

$$\frac{e_1}{w_1} + \frac{e_2}{w_2} + \ldots + \frac{e_n}{w_n} \leq 1. \tag{3.1}$$

If we find upper bounds for the blocking factors of MDPCP-C, we can then derive sufficient schedulability conditions using equation 3.1. The blocking factors can be better understood with the aid of the following lemma.

Lemma 3.4 *Whenever J_H attempts to enter an outermost global critical section, it can be directly blocked by lower priority remote jobs for at most the duration of the global critical section with the longest access time in $\beta_{H,LP(J_H)}$.*

Let V be the set of remote jobs which are currently within global critical sections in $\beta_{H,LP(J_H)}$ and that block J_H at the time J_H requests a lock on a semaphore corresponding to an outermost global critical section. During the blocking by jobs in V, a remote job with the priority lower than J_H cannot enter a critical section. The jobs in $LP(J_H) - V$ are the remote jobs with priorities lower than J_H; hence, they cannot contribute any blocking. Therefore, J_H can be directly blocked by a remote job with lower priority for at most the duration of the longest global critical section in $\beta_{H,LP(J_H)}$.

□

The above lemma provides an upper bound for the direct blocking caused by remote jobs with lower priorities, for each time that a job attempts to enter an outermost global critical section.

We now address the computation of blocking factors. First, we define additional notation. Let $LB_{k,i}$ be the worse case direct blocking time of job $J_{k,i}$ induced by one of its lower priority local jobs (i.e., the local blocking time of $J_{k,i}$). Let $GB_{k,i}$ be the worse case direct blocking time of job $J_{k,i}$ induced by its lower priority remote jobs (i.e., the remote blocking time of $J_{k,i}$) each time that $J_{k,i}$ attempts to enter an outermost global critical section. Let $IB_{k,i}$ be the worse case indirect blocking time of job $J_{k,i}$. Finally, we define $d_{k,i}$ as the number of times that $J_{k,i}$ enters an outermost global critical section.

By the definition of MDPCP-C, indirect blocking can only occur once during the execution of a job. Lemma 3.4 showed that each time a job $J_{k,i}$ attempts to enter an outermost critical section, it can be directly blocked by its remote jobs with lower priority. Thus, the worse case blocking time induced by indirect blocking and remote blocking is $IB_{k,i} + d_{k,i} * GB_{k,i}$. As for local blocking, every time $J_{k,i}$ suspends itself when it tries to enter a global critical section, its local jobs with lower priorities might enter local critical sections which can later cause $J_{k,i}$ to be blocked. Local blocking factors contribute $d_{k,i} * LB_{k,i}$ in the worse case.

In addition to the above blocking factors, remote and implicit preemption will also occur. When a job $J_{k,i}$ attempts to lock a global semaphore, it might be remotely preempted by its remote jobs with higher priorities. In the worse case, it has to wait for all its remote jobs with higher priorities accessing the global critical sections common to processor $\mathcal{P}_k$. So, in the worse case the blocking time caused by the remote preemption, $RP_{k,i}$, is $\sum_{J_{j,h} \in GP(J_{k,i})} c_{k;j,h} * \lceil w_{k,i}/w_{j,h} \rceil$, where $c_{k;j,h}$ is the total access time that job $J_{j,h}$ spends in the global critical sections common to $\mathcal{P}_k$.

Implicit preemption occurs when a job $J_{k,i}$ wants to lock a global semaphore S and finds that one of its higher priority remote jobs, $J_{j,h}$, is directly blocked by $\mathcal{SS}_j^*$ and $C(S) \geq P(J_{j,h})$. According to the definition of MDPCP-C, $J_{k,i}$ is implicitly preempted by $J_{j,h}$. Each time job $J_{k,i}$ attempts to lock a global semaphore whose priority ceiling is higher than or equal to the priority of one of its higher priority remote jobs, $P(J_{j,h})$, it is implicitly preempted. In other words, each time when $J_{k,i}$ wants to lock a global semaphore S, $J_{k,i}$ can be implicitly preempted for at most $\max_{J_{j,h} \in GP(J_{k,i})} GB_{j,h}$, where $C(S) \geq P(J_{j,h})$. To simplify the computation of the total implicit preemption time, we express the worse case of implicit preemption time of job $J_{k,i}$, $IP_{k,i}$, as $d_{k,i} * \max_{J_{j,h} \in GP(J_{k,i})} GB_{j,h}$. Hence, the worse case total blocking time of a job $J_{k,i}$ induced by MDPCP-C, $B_{k,i}^{MDPCP-C}$, can be expressed as follows.

$$\begin{aligned} B_{k,i}^{MDPCP-C} &= IB_{k,i} + d_{k,i} * (GB_{k,i} + LB_{k,i}) + \\ &\quad \sum_{J_{j,h} \in GP(J_{k,i})} c_{k;j,h} * \lceil w_{k,i}/w_{j,h} \rceil + \\ &\quad d_{k,i} * \max_{J_{j,h} \in GP(J_{k,i})} GB_{j,h}. \end{aligned} \tag{3.2}$$

We need to know the elements of the sets $LLP(J_{k,i})$, $LP(J_{k,i})$, and $GP(J_{k,i})$ for each job $J_{k,i}$ to compute the blocking sets and preemption factors. The set of local jobs with lower priorities, $LLP(J_{k,i})$, of job $J_{k,i}$ is the same as the set of the lower priority jobs of $J_{k,i}$ defined in uniprocessor systems, since both refer to the jobs on a single processor. A job $J_{k,l}$ in $LLP(J_{k,i})$ must arrive earlier and lock a local semaphore. It is preempted by $J_{k,i}$ when it is holding the semaphore such that later $J_{k,i}$ will be blocked. Since $J_{k,l}$ is preempted, it must have a later deadline. Consequently, the period of a job in $LLP(J_{k,i}$ must be longer than that of $J_{k,i}$.

However, the set of remote jobs with lower priorities, $LP(J_{k,i})$, of job $J_{k,i}$ does not possess the same nice properties as $LLP(J_{k,i})$. Jobs in $LP(J_{k,i})$ are the jobs whose deadlines are later than that of $J_{k,i}$, but not necessarily with earlier arrival times. So, they can be any jobs on the remote processors of processor $\mathcal{P}_k$. The same theory applies to the set $GP(J_{k,i})$. Consequently, $GP(J_{k,i}) = LP(J_{k,i}) = GP(J_{k,j}) = LP(J_{k,j})$. The sufficient schedulability conditions can be stated as follows:

Theorem 3.5 *Given m sets of periodic tasks on a system with m processors, where a set of n_k periodic tasks is assigned to processor $\mathcal{P}_k$. The sets of tasks can be scheduled by EDF with MDPCP-C as the resource control protocol, if the following conditions are satisfied:*

$$\forall k, 1 \leq k \leq m,$$

syn protocol \ scheduling scheme	RM	EDF
CT approach	MSPCP-C	MDPCP-C
RSL approach	MSPCP-R (=MPCP)	MDPCP-R

Figure 3.4: The mapping of the abbreviations for different approaches.

$$\frac{e_{k,1} + B_{k,1}}{w_{k,1}} + \frac{e_{k,2} + B_{k,2}}{w_{k,2}} + \ldots + \frac{e_{k,n_k} + B_{k,n_k}}{w_{k,n_k}} \leq 1. \tag{3.3}$$

□

3.3 Multiprocessor Dynamic Priority Ceiling Protocol R (MDPCP-R)

The multiprocessor dynamic priority ceiling protocol R (MDPCP-R) is based on a previously developed static priority multiprocessor protocol known as MPCP [RSL88]. To clearly distinguish our dynamic protocol MDPCP-C from the static protocol MPCP, we will subsequently use the acronym MSPCP-R to refer to the original MPCP protocol. MSPCP is short for multiprocessor static priority ceiling protocol. The suffix R indicates that the same resource control scheme is used as in MDPCP-R, and does not signify a revision to the original MPCP protocol. MDPCP-R relies upon an EDF scheduling policy, but is otherwise identical to MSPCP-R.

Figure 3.4 shows a brief mapping of the proposed protocols to the scheduling schemes, where RSL approach means the approach adopted by [RSL88]. Unlike CT approach (our approach), theirs does not allow nested global critical sections. Otherwise, the assumptions made by theirs do not differ from those required by ours.

Due to the varying restrictions concerning nested global critical sections, Both approaches take dramatically different policies to controlling access to global resources. To allow nested global critical sections, our approach relies upon priority ceilings to reflect the importance of global resources throughout the system. Thus ours requires a lock checking protocol that is similar to that used in uniprocessor systems, but must be much more complex. By contrast, due to its prohibition against nested global critical sections, RSL approach can use simple efficient atomic operations, such as test-and-set, to implement global locking. Both approaches may use a uniprocessor synchronization protocol to manage local resources.

3.3.1 Basic Idea of MDPCP-R

We use a slightly different definition of priority ceiling that we used in MDPCP-C. The priority ceiling of a local semaphore S_L, $C(S_L)$, is defined to be the priority of the highest priority job that is accessing or will access the semaphore at the current time. This is the same definition used in the (uniprocessor) DPCP. Recall from Section 3.2.1 that, in order to easily bound remote blocking times, it is necessary to prevent jobs executing within global critical sections from being preempted by jobs executing outside of critical sections. Consequently, a job within a global critical section must have a priority higher than every job executing outside of global critical sections. This is easily handled by introducing the concept of the *base priority* to denote the priority that is higher than the highest priority job in the entire system.

A job J_i bound to $\mathcal{P}_j$ is assigned a new priority, $\tilde{P}(J_i, S)$, when it locks a semaphore S, and reverts to its previous priority when it releases the semaphore. The *extended priority* $\tilde{P}_{J_i,S}$ is defined to be $P(J_i)$ if S is a local semaphore, and $P(J_i)$ plus the base priority if S is a global semaphore. Since the remote priority ceilings of all global semaphores have been increased by the base priority, a job executing within a global critical section has a higher priority than any job outside of a global critical section. This is assured that a job that has locked a global semaphore may only be preempted by a local job that locks another global semaphore that has a higher remote priority ceiling.

The protocol can be described as follows:

1. When job J wants to access a local critical section, it uses DPCP to see if it can lock the associated semaphore. i.e., J can seize the lock, only if $P(J) > C(S_L^*)$, where S_L^* denotes the semaphore with the highest priority ceiling of all local semaphores currently locked by jobs other than J. DPCP is used to synchronize access to local resources.

2. If job J attempts to access a global critical section, it locks the associated semaphore S if no other job has already locked S. Otherwise, it joins the priority-ordered queue associated with S using its original priority $P(J)$,

3. A job J locking a global semaphore S_g inherits the extended priority $\tilde{P}_{J,S_g}$, and reverts to its previous priority upon releasing S_g.

4. A job J can lock S_g and preempt another job J' within another global critical section guarded by S', if $\tilde{P}(J, S_g) > \tilde{P}(J', S_g')$.

5. Whenever a global semaphore is released, it will be given to the highest priority job waiting if the associated queue is not empty.

While a job has locked any global semaphore, it cannot attempt to lock any other semaphore, whether it is local or global. Thus a job cannot deadlock while holding a lock for a global semaphore. Jobs can simultaneously lock multiple local semaphores, but since MDPCP-R uses DPCP to manage the access of local critical sections, a job cannot become deadlocked within a local critical section. So MDPCP-R is deadlock free.

3.3.2 Schedulability Analysis of MDPCP-R

Blocking times in MDPCP-R depend upon one type of blocking that does not arise in MDPCP-C. In MDPCP-R, a job within a global critical section S can preempt a local job within another global critical section S'. Hence, it can induce another form of blocking delay to jobs that waits for S'. Blocking times in MDPCP-R fall into the following categories:

1. *Blocking by local jobs with lower priorities within local critical sections.* When a job $J_{k,i}$ attempts to lock a global semaphore S, it may suspend while waiting for some job to unlock S. In the meantime, one of its local jobs might lock a local semaphore which will later cause job $J_{k,i}$ to be blocked. Let $LLB_{k,i}$ be the worse case access time of a local semaphore accessed by a lower priority local job of job $J_{k,i}$ that can block $J_{k,i}$. Let $d_{k,i}$ denote the number of times that $J_{k,i}$ locks a global semaphore. The worse case blocking time caused by this scenario can be expressed as $d_{k,i} * LLB_{k,i}$.

2. *Blocking by local jobs with lower priorities accessing global critical sections.* This type of blocking is similar to the previous one, except that, in this case, a lower priority local job can lock or be waiting for a global semaphore that might later cause $J_{k,i}$ to be blocked when $J_{k,i}$ executes outside of a critical section. Let $GLB_{k,i,l}$ be the worse case access time of a global semaphore accessed by the lower priority job $J_{k,l}$ that can block $J_{k,i}$. For every lower priority job $J_{k,l}$ of job $J_{k,i}$, this form of blocking can contribute at most $\min(d_{k,l}, d_{k,i}+1) * GLB_{k,i,l}$ blocking delay.

3. *Blocking by remote jobs with lower priorities.* When job $J_{k,i}$ attempts to lock a global semaphore, that semaphore might already be locked by a lower priority remote job. Let $GRB_{k,i}$ be the worse case access time of the global semaphore that is accessed by job $J_{k,i}$ and a lower priority remote job. Then job $J_{k,i}$ can experience at most $d_{k,i} * GRB_{k,i}$ blocking delay caused by this situation.

4. *Blocking by remote jobs with higher priorities.* When job $J_{k,i}$ tries to lock a global semaphore, that semaphore might be locked or a higher priority remote job might be waiting for it. Let $d^C_{k,i;m,h}$ be the number of times that job $J_{m,h}$ locks the global semaphores which will be also accessed by $J_{k,i}$. Let $GHB_{k,i;m,h}$ be the worse case access time of the global semaphore accessed by $J_{k,i}$ and $J_{m,h}$. We call this form of blocking remote preemption. The worse case blocking time caused by remote preemption is $d_{k,i;m,h} * \lceil w_{k,i}/w_{m,h} \rceil * GHB_{k,i;m,h}$, for each remote job $J_{m,h}$, with higher priority, of job $J_{k,i}$.

5. *Blocking by a remote job accessing a global critical section.* Suppose job $J_{k,i}$ is blocked by a remote job $J_{m,j}$ accessing a global semaphore S_{g1}. Meanwhile, suppose another remote job $J_{m,x}$ inherits a higher extended priority and preempts $J_{m,j}$, when $J_{m,x}$ locks another global semaphore S_{g2}. Not only does job $J_{k,i}$ experience the blocking delay caused by the semaphore S_{g1} that it tried to lock; it also experiences a blocking delay due to S_{g2}. The blocking by the former semaphore is considered above; the blocking by the latter is considered here. Let $d^H_{k,i;m,x}$ be the number of times that job $J_{m,x}$ locks the global semaphores with higher remote priority ceilings than a global

semaphore that is accessed by another local job of $J_{m,x}$ and that can block $J_{k,i}$. We use the notation $GHB_{k,i;m,x}$ to refer to the worse case access time of the global semaphore as described above. This type of the blocking time can be bound by the expression $d^H_{k,i;m,x} * \lceil w_{k,i}/w_{m,x} \rceil * GHB_{k,i;m,x}$, for every remote job $J_{m,x}$ of job $J_{k,i}$.

The total blocking time of a job $J_{k,i}$ induced by MDPCP-R, $B^{MDPCP-R}_{k,i}$, is the summation of the blocking factors described above.

$$
\begin{aligned}
B^{MDPCP-R}_{k,i} = \; & d_{i,k} * LLB_{k,i} + \\
& \sum_{J_{k,l} \in LLP(J_{k,i})} \min(d_{k,l}, d_{k,i} + 1) * GLB_{k,i,l} + \\
& d_{k,i} * GRB_{k,i} + \\
& \sum_{J_{m,h} \in GP(J_{k,i})} d_{k,i;m,h} * \lceil w_{k,i}/w_{m,h} \rceil GHB_{k,i;m,h} + \\
& \sum_{\forall J_{m,x} m \neq k} d^H_{k,i;m,x} * \lceil w_{k,i}/w_{m,x} \rceil * GHB_{k,i;m,x}.
\end{aligned} \tag{3.4}
$$

The definitions of the set of remote jobs with lower priorities and higher priorities and the set of local jobs with lower priorities, $LP(J_{k,i})$, $GP(J_{k,i})$, and $LLP(J_{k,i})$, for a job $J_{k,i}$ are the same as those defined in MDPCP-C, since both use dynamic priorities.

3.4 Performance Comparisons for Multiprocessor Priority Ceiling Based Protocols

This section compares the performance of two static priority protocols, MSPCP-C and MSPCP-R, and two dynamic priority protocols, MDPCP-C and MDPCP-R. The multiprocessor static priority ceiling protocol C, MSPCP-C is a variation of MDPCP-C which uses a RM scheduling algorithm. The primary differences between MSPCP-C and MDPCP-C are the definitions for priorities and priority ceilings. MDPCP-C defines $P(J)$ and $C(S)$ in exactly the same fashion as uniprocessor PCP. MDPCP-C also preserves the useful MDPCP-C properties: freedom from deadlock and prevention of transitive blocking. The proofs are analogous to those for MDPCP-C. The blocking factors induced by MSPCP-C are similar as well: indirect blocking, local and remote blocking, remote preemption, and implicit preemption. Hence, the expression for the worse case blocking time of a job $J_{k,i}$, $B^{MSPCP-C}_{k,i}$ is the same as that in MDPCP-C, $B^{MDPCP-C}_{k,i}$. The only difference is the definitions of the sets of lower (and higher) priority remote jobs, i.e., $LP(J_{k,i})$ and $GP(J_{k,i})$. $LP(J_{k,i})$ is the set of remote jobs with longer periods than $J_{k,i}$, and $GP(J_{k,i})$ is the set of remote jobs with shorter periods than $J_{k,i}$. A set of periodic tasks can be scheduled by RM if the following condition is satisfied[LL73]:

$$
\sum_{i=1}^{n} \frac{e_i}{w_i} \leq n(2^{\frac{1}{n}} - 1). \tag{3.5}
$$

The metric used to compare the schedulability of these protocols is the maximum estimated consumed processor power ($MECPP$). Inequality 3.1 shows the intuition behind this measurement. When an EDF scheduling policy is used in a single processor system, the utilization of the processor must be less than 1 in order to meet all the deadlines of the periodic tasks in the system. The left-hand side of the inequality is the upper bound of the processor utilization consumed by the tasks in the system. This upper bound, called the estimated consumed processor power (ECPP), can be viewed as a measure of schedulability. For a multiprocessor system, each processor $\mathcal{P}_k$ has its associated ECPP value, $ECPP_k$. The deadlines of all tasks in the system will be met if all the ECPP values are less than 1; or equivalently, if the maximum of the ECPP values is less than 1. Consequently, the maximum ECPP value (MECPP) is a natural performance metric for the schedulability of multiprocessor hard real-time systems.

Given m sets of n tasks each assigned to an m multiprocessor system and each processor accepts a set of n tasks. The estimated consumed processor power of processor $\mathcal{P}_k$ ($ECPP_k$) is defined as $\sum_{i=0}^{n} \frac{e_{k,j}}{w_{k,j}} + \max_{1 \leq i \leq n} \frac{B_{k,i}}{w_{k,i}}$, if RM scheduling is used and $\sum_{j=0}^{n} \frac{e_{k,j}+B_{k,j}}{w_{k,j}}$, if EDF is used, where $B_{k,j}$ is the worse case blocking time of job $J_{k.j}$ induced by the corresponding resource synchronization protocol. The computation of $B_{k,j}$ is described in Sections 3.2.3 and 3.3.2. $MECPP$ is defined as $\max_{1 \leq k \leq m} ECPP_k$. Note that utilization for real work (processor utilization without synchronization delay) is the same for both CT and RSL approaches, i.e., $\sum_{i=0}^{n} \frac{e_{k,j}}{w_{k,j}}$ for processor $\mathcal{P}_k$. Therefore, the major factor of schedulability depends on synchronization delay. According to the schedulability condition proved in Theorem 3.5, If $ECPP_k$ is smaller, we have greater chance that the set of tasks assigned to processor $\mathcal{P}_k$ is schedulable. Therefore, we say that a protocol performs better, if the protocol leads to a smaller $MECPP$.

3.4.1 Simulation Design

The simulator models a system with m processors and shared memory. It consists of two components, the configuration model and the task model. The configuration model produces global critical sections shared with different sets of processors and local critical sections for each processor. The task model generates m sets of tasks; each set contains n periodic tasks. The configuration model generates nested global critical sections since MSPCP-C and MDPCP-C allow them. However, for MSPCP-R and MDPCP-R, a job must use a coarser level of granularity for global semaphores.

The following parameters control the configuration of the simulated system:

- m is the number of processors in the system.
- n is the number of tasks accepted by each processor.
- $NumLCS$ is the maximum number of local semaphores for a processor. In our simulation, $NumLCS$ is 4.
- $NumGCS$ is the maximum number of global semaphores for a processor. In our simulation, $NumGCS$ is 4.

- $TotalGCS$ is the total number of global semaphores in the whole system. In our simulation, $TotalGCS$ is 8.
- $CSAccessTime$ is the maximum access time of a critical section. In our simulation, $CSAccessTime$ is 4 time units. We assume that all tasks have the same access time for executing the same global critical section.
- $DegreeSharing$ is the probability that a global critical section is accessible from a processor. Setting $DegreeSharing$ to 0 means that no global critical sections are shared by different processors (all global semaphores become local in this case), while a value of 1 indicates that all global critical sections are accessible from every processor. The greater the degree of sharing, the more frequently jobs interfere with each other.

The task model determines the attributes of the various tasks. The following parameters control the task model.

- $MinPeriod$ and $PeriodIncrement$ defines the periods of tasks. For a task $T_{k,i}$, the period of task $T_{k,i}$, $w_{k,i}$ can be computed by

  $$w_{k,i} = \begin{cases} MinPeriod + PeriodIncrement * R_w & \text{if } i = 1 \\ w_{k,i-1} + PeriodIncrement * R_w & \text{otherwise,} \end{cases}$$

  where R_w is a number uniformly distributed over (0,1].
- $LbTaskConsumedPower$ and $UbTaskConsumedPower$ define the lower and upper bounds of the task consumed processor power, the rate of the execution time to the period of a task. The execution time of a task $T_{k,i}$, $e_{k,i}$, is defined as

  $$e_{k,i} = w_{k,i} * R_e,$$

  where R_e is uniformly distributed between $LbTaskConsumedPower$ and $UbTaskConsumedPowe$
- The list of critical sections accessed by a task is represented as a bit map which is generated by a random number and uniformly distributed between 0 and the maximum number of possible bit map patterns.

Our simulation does not generate workload based on processor utilization; different workloads are generated by varying run-time parameters: the number of processors, the number of tasks, and the degree of sharing. Under a fixed setting of the parameters, we want to see how the protocols perform on various utilization. We define processor workload as processor utilization of the real work. Therefore, the average workload for a processor can be expressed as $\frac{n}{2}(LbTaskConsumedPower + UbTaskConsumedPower)$.

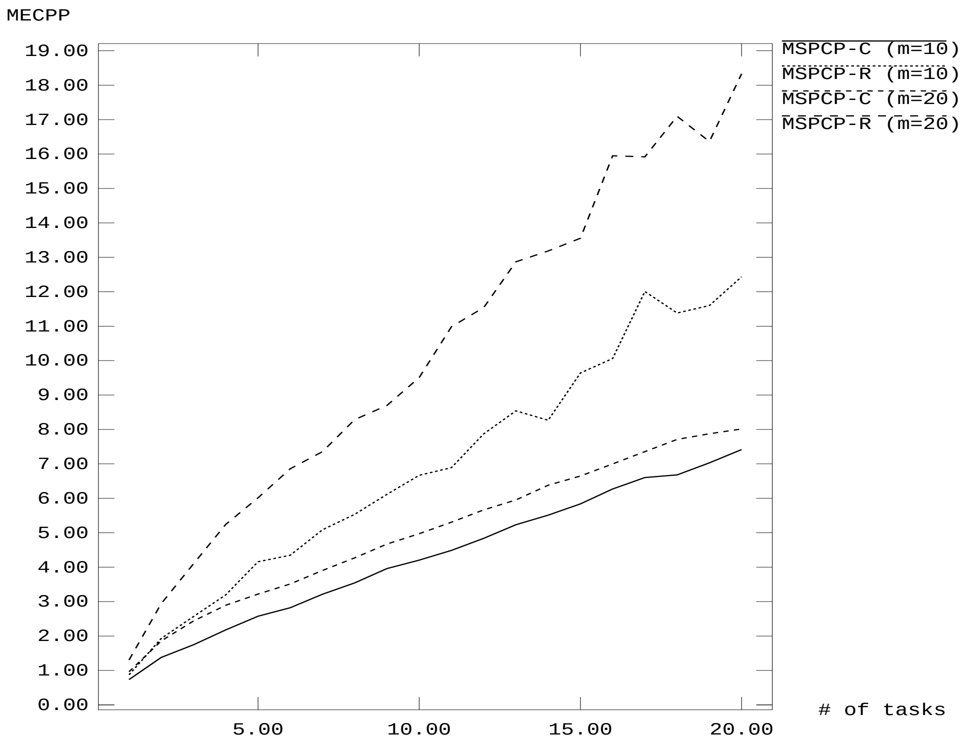
Three run-time parameters are used to simulate various system workloads. Varying the number of tasks provides a way to see how the protocol behaves as the processor workload increases, and varying the number of processors shows how the protocol behaves as the system size scales. Changing the degree of sharing illustrates the impact of synchronization for access to global resources. In the following section, we give the results of several simulation experiments. In each experiment, we varied only a single system parameter, and held all the others constant.

3.4.2 Simulation Results

In order to see the impact of increases in the processor workload on performance, we varied the number of tasks for a fixed number of processors. The results are shown in Figure 3.5. We also changed the degree of sharing when varying processor workload, and found similar results. Therefore, we only present the results for the case when the degree of sharing was set equal to 0.5. Clearly, resource contention and blocking time increase as the workload increases. The rate of increase of blocking by MSPCP-R and MDPCP-R are greater, because blocking factors 4 and 5, described in Section 3.3.2, increase significantly as the number of tasks increases. The slopes of MSPCP-C with m equal to 10 and 20 are almost identical. The increased MECPP values for the case where m is 20 are primarily due to remote preemption, and remain constant throughout the various processor loads. A job will have more remote jobs with higher priorities when m is 20, which increases the amount of more remote preemption for the job. The rate of increase of the number of higher priority remote jobs remains stable as the processor workload increases; there is negligible difference between the cases where m equals 10 and m equals 20. Consequently, the increase in MECPP values is almost identical for the two cases. However, MDPCP-C behaves differently since the blocking time of each task affects the MECPP; it is unlike MSPCP-C where only a single blocking time matters. As processor workload increases, the discrepancies between MSPCP-C and MSPCP-R (or between MDPCP-C and MDPCP-R) become significant. MSPCP-C and MDPCP-C perform better under a wide range of processor workloads.

Figure 3.6 and Figure 3.7 show the simulated performance results as the degree of sharing changes. For MDPCP-C and MSPCP-C, the increased concurrency allowed by the fine granularity of resources becomes more significant as the degree of sharing increases. Two jobs can simultaneously access different critical sections, in cases where MDPCP-R and MSPCP-R would force them to be serialized. In general, the MECPP increases as the degree of sharing increases. However, it increases much faster under MSPCP-R or MDPCP-R than under MSPCP-C or MDPCP-C. Again, we see that MSPCP-C and MDPCP-C allow better performance.

To study the effect of resource contention among processors, we varied the number of processors while holding other system parameters fixed. We present the simulation results from those experiments in Figure 3.8. MSPCP-C and MDPCP-C are less sensitive to changes in system size than MSPCP-R and MDPCP-R, and have better performance as well. These results have one similarity with the results from the experiments of varying processor workload. The differences of the MECPP values for MSPCP-C remain constant, while the differences of the MECPP values for MDPCP-C continue to increase. The cause of such behavior is the same in both cases.



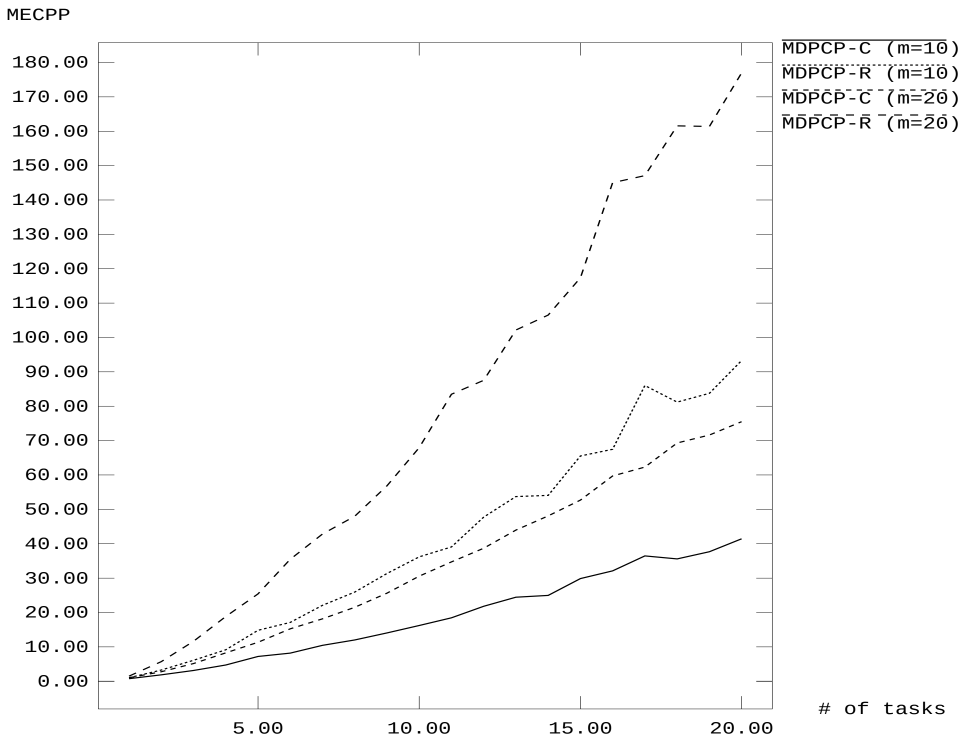
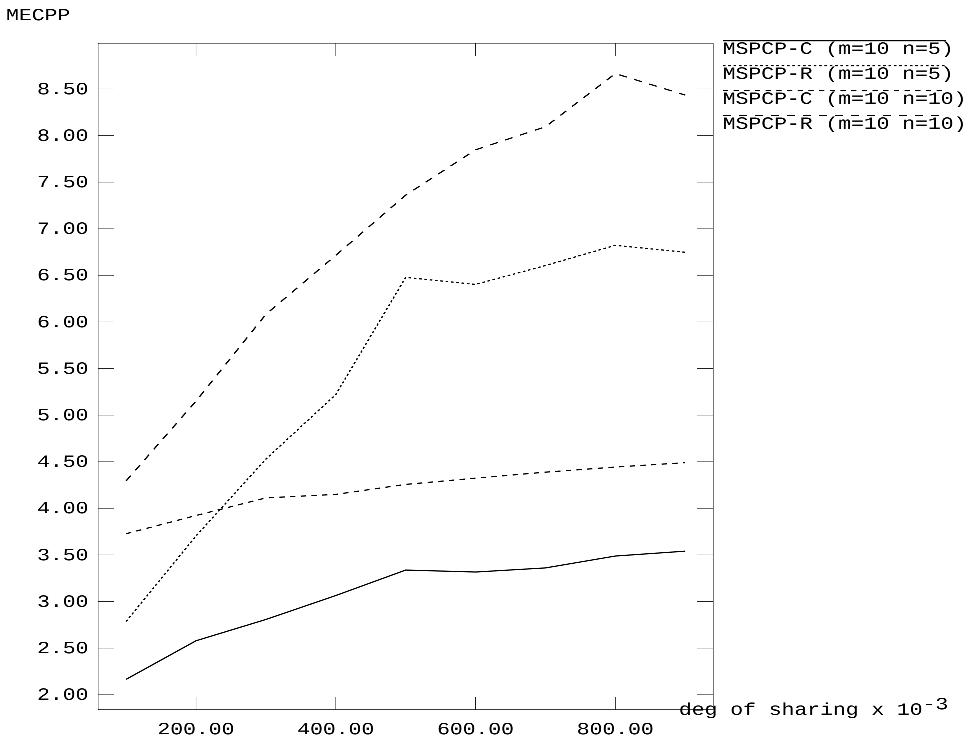


Figure 3.5: Varying processor workload.



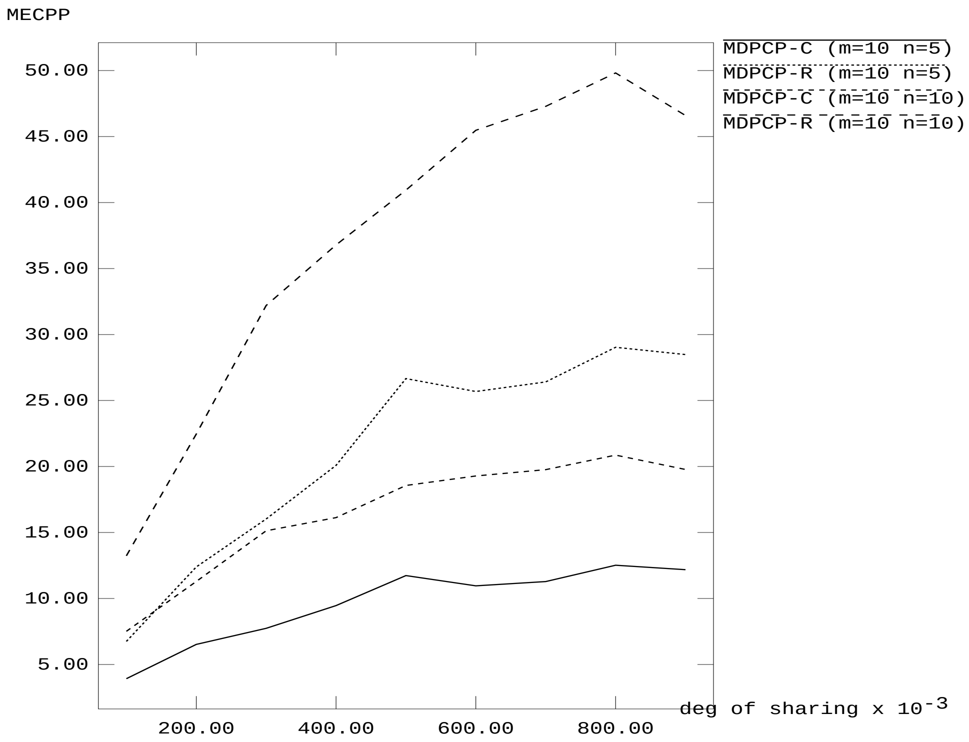


Figure 3.6: Varying the degree of sharing when $m = 10$.

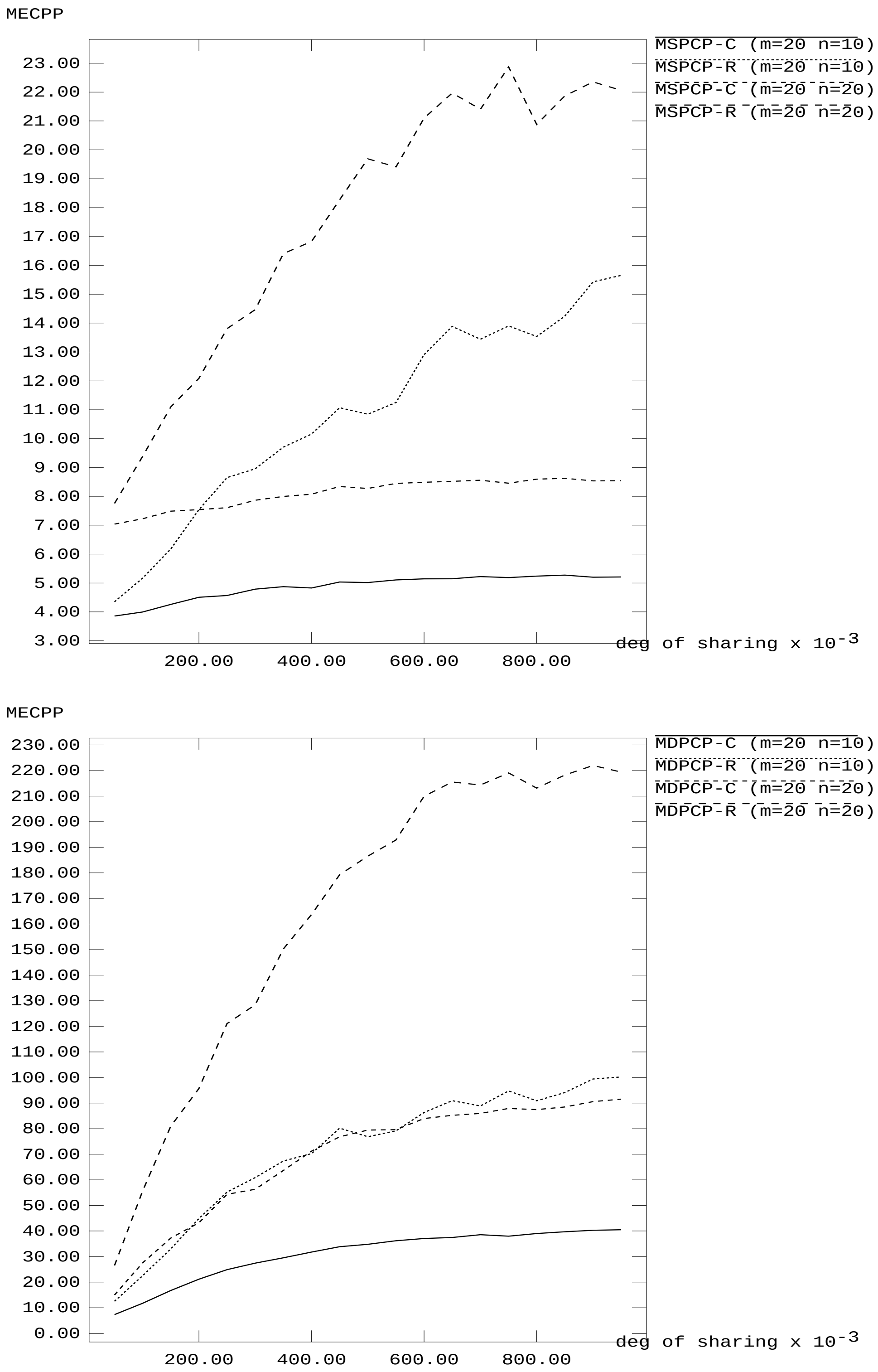


Figure 3.7: Varying the degree of sharing when $m = 20$.

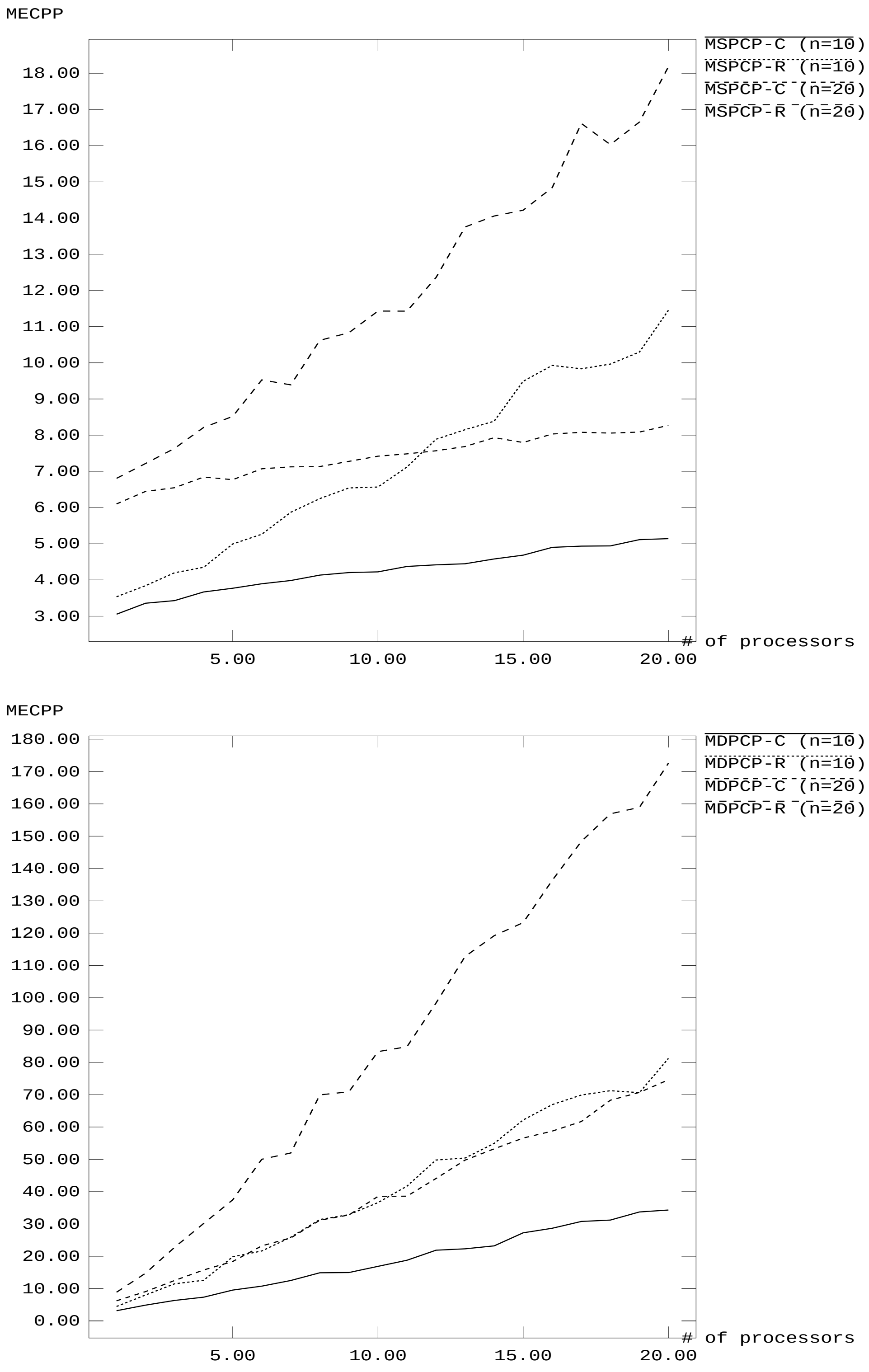


Figure 3.8: Varying system size.

Chapter 4

Optimistic Concurrency Control Protocols

Optimistic concurrency control (OCC) mechanism assumes that data conflicts will be rare. Its efficiency relies on the hope that conflicts among transactions will not occur. The execution of a transaction under OCC mechanism can be divided into three phases: read phase, validation phase, and possibly update phase. This is based on the observation that reads are completely unrestricted but writes are severely restricted. Reading a value from a data object does not lose data integrity; however, changing the value of an object needs to be validated to see if data consistency is maintained. The first phase reads data objects, processes the data objects that have been read, and prepares the data objects to be written. Note that in this phase all writes take place on local copies of the objects. The second phase invokes concurrency control algorithm, making sure that the changes made by the validating transaction will not lose data integrity. A write phase is enabled only if the validation phase succeeds (data integrity is maintained). Otherwise, the transaction might restart as a new transaction or wait for the data conflicts being removed. In the write phase, the local copies of the changes are made global.

A widely used criterion for validating the correctness of concurrent execution of transactions is called serializability [EGLT76]. An interleaved execution sequence of transactions is correct if it is serializable. If transaction A conflicts with transaction B, defined below, we said that the interleaved execution sequence of these two transactions violates serializability [Har84].

Each data object is associated with two locks: read and write locks. A data object can only be write-locked by one transaction at a time, while it can be read-locked by many transactions simultaneously. Let read set for transaction A, $\mathcal{R}_A$, be the set of data objects that are read by transaction A and write set for transaction A, $\mathcal{W}_A$, be the set of data objects that are written by transaction A. Transaction A conflicts with transaction B if

$$\mathcal{W}_A \cap \mathcal{R}_B \neq \phi.$$

The conflict set of transaction A is the set of the transactions with which transaction A conflicts.

Our baseline OCC, OPT-BC (stands for Broadcast Commit), is classified as forward oriented optimistic concurrency control (FOCC) [Har84] which checks whether the write set of the validating transaction intersects with the read set of any transactions having not yet finished their read phase. The idea of OPT-BC [KR81, MN82] can be described as follows. When a transaction commits, it notifies all the transactions with which it conflicts and restarts all the conflicted transactions immediately. A transaction is restarted once it is found to be in conflict with a validating transaction. Such restart is useful, not *wasted restart* [HCL90], because a validating transaction is guaranteed to commit when it restarts all the transactions in its conflict set. Wasted restart occurs when a transaction restarts another one and later it misses its deadline. *Mutual restart* problem [1] [HCL90] will not happen in OPT-BC, because a validating transaction commits right after it restarts the conflicting transactions. The validating transaction will not restart once it is in validation phase.

The remaining chapter is organized as follows. A set of OCC algorithms using the knowledge of execution time is described in the following section. In Section 4.2, we describe the RTDBS model used for performance evaluation and compare the proposed algorithms with the baseline algorithm. The simulation results show that a proper use of execution time information can improve system performance.

4.1 The Proposed Concurrency Control Algorithms

Based on the observations mentioned in Section 1.1, we might be able to improve the baseline OCC by making use of the knowledge of execution time. We will examine the possible alternatives of conflict resolution and develop OCC algorithms which make conflict resolution decision based on the knowledge of execution time. Although priority information can be used to resolve conflicts, in this paper, we focus on the effects of the knowledge of execution information and hence factor out the priority of transactions.

The proposed OCC algorithms belong to FOCC which checks if the write set of validating transaction conflicts with the read sets of active transactions. Since conflicting transactions have not yet committed, there are several alternatives on resolving data conflicts.

- *Wait for conflicting transactions*
  A validating transaction waits for the transactions in its conflict set to complete. The validating transaction is deferred and the validation needs to be retried later. As it waits, new active transactions might cause data conflicts and could lengthen the wait time. According to the assumption of OCC (conflicts happen rarely), waiting eventually ends and the validating transaction commits.

- *Remove conflicting transactions and commit*
  Since the conflicting transactions have not yet committed, data conflicts can be removed by either restarting or aborting them. Although resources having been consumed by the conflicting transactions are wasted, this strategy guarantees that the

[1] Mutual restart describes the phenomenon that two transactions restart each other.

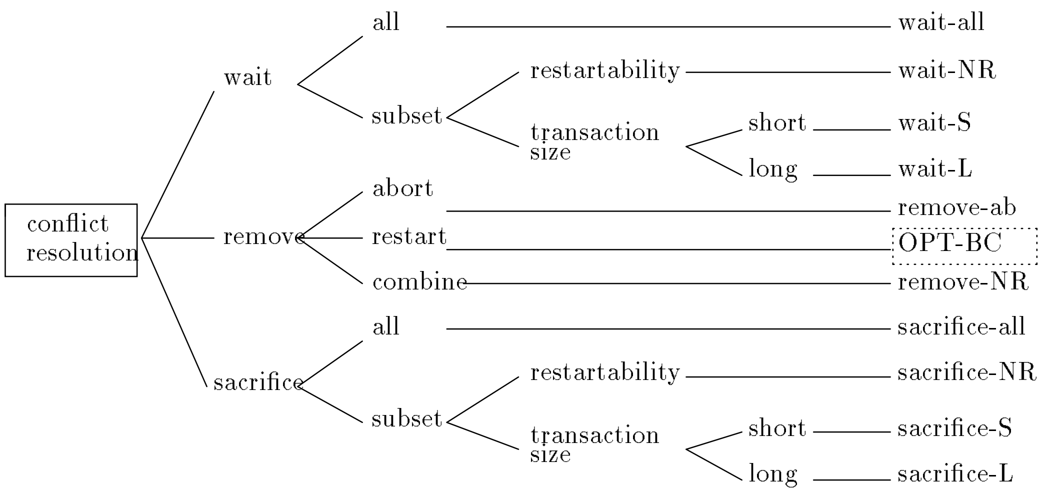


Figure 4.1: The family tree of the proposed OCC algorithms.

validating transaction can commit, not like the previous one where it might miss its deadline or wait infinitely.

- *Sacrifice validating transaction*
  In some cases, it might be beneficial that we sacrifice (restart) validating transaction. For example, when a validating transaction conflicts with many transactions, waiting for them might encounter more conflicts and lengthen the wait time, and removing them might not compensate the performance gain (committing one transaction).

Based on the knowledge of execution time, we have two schemes to classify conflicting transactions. One classification scheme uses restartability described in Section 1.1 and the other classifies transactions by the length of execution time. The proposed OCC algorithms make conflict resolution decision based on the classification of conflicting transactions. A hybrid strategy of two conflicting resolution alternatives might be adopted in order to make distinct decisions on different classes of conflict transactions. For the continuity of the performance evaluation, we also develop and evaluate OCC algorithms without the use of execution time.

Figure 4.1 shows the combinations of the conflict resolution alternatives and the classification schemes. They constitute the possible protocols for FOCC with or without the knowledge of execution time. In the following subsections, we will describe the proposed algorithms and discuss their potential shortcomings and strengths in an intuitive manner.

4.1.1 WAIT-ALL

WAIT-ALL does not utilize execution time information to classify conflicting transactions. It waits for all conflicting transactions to complete. It depends on the assumption that conflicts will occur rarely. Otherwise, it might wait infinitely, if it keeps conflicting with new active transactions.

4.1.2 WAIT-NR

WAIT-NR stands for waiting for non-restartable transactions to commit. The decision for conflict resolution depends on whether the conflicting transactions are restartable or not. It waits whenever there is a non-restartable transaction in the conflict set. Such waiting is worthwhile in the sense that if it does not wait, the conflicting non-restartable transactions must miss their deadlines and the resources having been consumed by them are wasted. Hence, it gives preferential treatment to non-restartable transactions. If the transactions in the conflict set are all restartable, the algorithm restarts them and commits. This algorithm is an example of using a hybrid strategy of two resolution alternatives, wait and remove.

4.1.3 WAIT-S

The algorithm waits for the conflicting transactions which are shorter than the validating transaction. It attempts to minimize the wait time by only waiting for shorter transactions and restarting longer transactions. With the minimized wait time, the validating transaction could have a chance to commit earlier and hence the algorithm might achieve lower miss ratio.

The algorithm biases on short transactions; hence its average response time should be small. The problem on the algorithm and the algorithms, described below, which use transaction size to classify conflicting transactions is fairness and starvation. For example, an extremely long transaction might wait for short transactions, could be restarted many times, and never has a chance to commit.

4.1.4 WAIT-L

This algorithm asserts that restarting long transactions might waste more resources than restarting short transactions, so it waits for long transactions to complete and restarts short ones instead. The average wait time of this algorithm might be longer than that of WAIT-S, because it always waits for long transactions. In high data contention, conflicts might increase substantially as the wait time increases. More data conflicts keeps a validating transaction waiting or lets it restart more transactions in the conflict set.

4.1.5 REMOVE-AB

The algorithm resolves conflicts by aborting the conflicting transactions. Like OPT-BC, its efficiency depends on the assumption that the data conflicts happen rarely. Otherwise, the miss ratio could be high. Without careful evaluation, we cannot tell OPT-BC or REMOVE-AB is better for RTDBS. We will be able to answer this question after the simulation.

4.1.6 REMOVE-NR

REMOVE-NR does not wait. It aborts the conflicting transactions which are non-restartable, restarts the rest of the conflicting transactions (which are restartable), and then commits without waiting. This algorithm prefers to commit validating transaction as soon as possible such that the wait time is minimized (actually, it is zero). The side benefit of the early commit is the resource saving and the reduction of data contention. The resources saved from the early discarded non-restartable transactions could be utilized by other transactions. Further data contention might be reduced due to the less number of active transactions in the system.

4.1.7 SACRIFICE-ALL

The algorithm sacrifices validating transaction as long as there is a conflicting transaction. Such sacrifice is effective for limiting wait time, in case of many conflicting transactions. Although sacrificing can avoid long waiting, a validating transaction might be restarted many times, if a long transaction is in the conflict set.

4.1.8 SACRIFICE-NR

The algorithm sacrifices validating transaction *conditionally*. Non-restartable transactions can not meet deadlines, if they are restarted. Therefore, SACRIFICE-NR restarts the validating transaction, if the conflict set contains non-restartable transactions. It restarts the conflicting transactions if they are all restartable and commits the validating transaction.

4.1.9 SACRIFICE-S

The algorithm asserts that short transactions are more possible to complete than long transactions, even if they are restarted. The algorithm sacrifices (short) validating transaction if it conflicts with longer transactions. SACRIFICE-S commits the validating transaction only if the transactions in the conflict set are all shorter than the validating transaction and they are restarted.

The potential weakness results from that long transactions last long in the system. Hence, a validating transaction sacrificed once might be sacrificed again and later it misses the deadline, due to the long duration of such transactions. Such sacrifice wastes resources and causes performance degradation.

4.1.10 SACRIFICE-L

SACRIFICE-L has the same assertion as SACRIFICE-S, but it makes contrary decisions. It prefers to let short transactions commit, rather than restart them. Hence, it sacrifices validating transaction if it conflicts with short transactions. Its efficiency relies on the hope that the sacrifice of a validating transaction can help many short transactions complete. The algorithm restarts the conflicting transactions if they are all longer than the validating transaction and commits the (short) validating transaction.

Restarting long transactions in a conflict set, in order to commit a validating transaction, should be justified. One can argue that long transactions might have already consumed a large amount of resources and it is not wise to sacrifice such transactions. Without a performance study, we cannot distinguish which one, SACRIFICE-S or SACRIFICE-L, makes a better conflict decision.

4.2 Performance Study for Real-Time Optimistic Concurrency Control

4.2.1 Simulation Design

Our real-time database system model simulates a multiprocessor system with disk resident database. The simulation is written in C, using a process-oriented simulation package CSIM [Sch90]. The database is modeled as a collection of pages evenly spread over the disks in the simulated system. We assume that there is no buffer management associated with the database. In real case, buffer management is essential to a database system, while it is important to know the worst case performance in a real-time system. Our simulation attempts to simulate the worst case performance, so the buffer management is excluded. Hence, each read/write access involves disk I/O activity. We also assume that each read/write page access is associated with a fixed period of time to access CPU for the processing of the page, but disk write operations are deferred to update phase as mentioned in Section 4. The same page and CPU access pattern occurs if a transaction restarts. If a transaction cannot make its deadline, it is aborted and discarded.

Transaction arrivals are simulated as a Poisson distribution and each transaction is associated with an execution time and a deadline. The calculation of the execution time of a transaction is based on the number of read/write accesses made by the transaction and the deadline depends on a randomly generated slack rate. The detail formula for the deadline computation will be described later.

The simulation system mainly consists of three components: workload generator, RT-DBS simulator, and statistic data collector as shown in Figure 4.2. Workload generator is used to generate a variety of transactions with different database access patterns and transaction sizes. It also controls the size of the simulated database. The system simulator in RTDBS simulator is the main body of the simulation which accepts input transactions, schedules and executes them, simulates the consumption of resources, and invokes OCC to resolve data conflicts. To evaluate different OCC algorithms, we only need to change the subcomponent, OCC simulation. Statistic data collector gets the statistic results for performance analysis. The statistic information can help us to explain the behavior of the algorithms, including the average wait time, the average response time for a committed transaction, the average number of active transactions in a system, and the number of data conflicts for a transaction. Note that an active transaction is visible to our statistic data collector only after it makes the first read or write operation. The purpose of counting the number of active transactions is to measure the level of concurrency, not to the level of multiprogramming. Therefore, if an transaction does not make any database access, we do not consider it as an active transaction in our simulated system.

Resource simulation considers two resources: CPUs and disks. There is only one queue for all CPUs and separate queue for each disk. Earliest deadline first (EDF) policy is used to schedule both resources. However, the service discipline of CPUs is preemptive-resume, while that of disks is non-preemptive. From the observation addressed by Abbott and Garcia-Molina [AGM92], we know that processor scheduling policy, such as first come first serve (FCFS), EDF, least slack time first, does not have much influence on the performance of different concurrency control schemes. Our simulation also conduct the experiments with FCFS as processor scheduling policy and have the similar observation: the relative performance of the proposed concurrency control strategies remains under different processor scheduling policies. Since the aim of our simulation is to evaluate the effect of execution time information on concurrency control, not to evaluate that in various scheduling policies, therefore we choose a simple and reasonable scheduling policy for the resources. The following parameters control the resource simulation:

- $NumCPU$ is the number of CPUs in the simulation system.
- $NumDisks$ is the number of disks in the simulation system.
- $CPUAccessTime$ is the access time of CPU for processing a page.
- $DiskAccessTime$ is the access time of disk for accessing a page on disk.

Workload generator is responsible for generating various system workloads to evaluate concurrency control algorithms. The following parameters are used by the workload generator to create transactions with various access patterns and the associated timing constraints.

- $DBSize$ defines the number of pages in the database. The data pages of the database are uniformly spread over the disks.
- $NumPages$ specifies the average number of pages to be accessed by a transaction. The total number of pages accessed by transaction X, $TotalPages_X$, is computed by

$$TotalPages_X = NumPages * U_p,$$

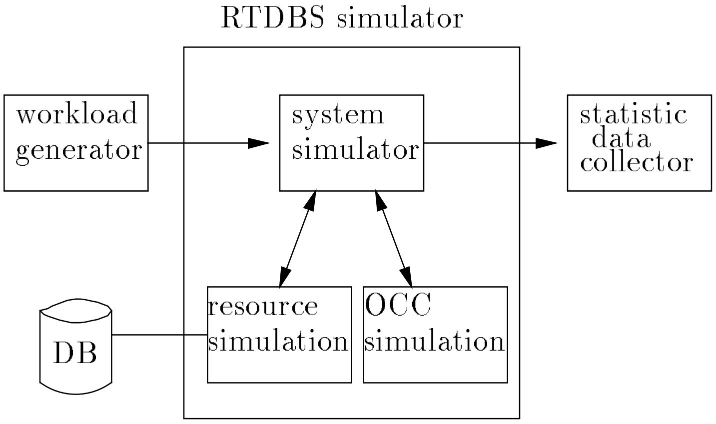


Figure 4.2: Structure of the simulation.

where U_p is uniformly distributed over an interval, such as an interval between 0.5 and 1.5. The page requests of a transaction is a uniform distribution spreading over the entire database. The execution time can be calculated based on the total number of page accesses.

- Wp is the probability that a transaction writes a page.
- $LbSlackRate$ and $UbSlackRate$, the lower bound and upper bound of the slack rate, defines the range of the slack rate which is used to tune up the tightness of the real-time constraints. Let A_X be the arrival time and E_X be the execution time of transaction X. The formula for computing the deadline of transaction X, D_X, is

$$D_X = A_X + E_X * U_d,$$

where U_d is uniformly distributed between $LbSlackRate$ and $UbSlackRate$.

- $InterArrivalTime$ specifies the mean inter-arrival time (IAT) in millisecond between two consecutive transactions which have an exponential distribution.

Workload of a simulated system can be computed as follows. We define resource usage of a transaction X as $\frac{E_X}{D_X - A_X}$, since the execution time counts CPU and disk usage. The average workload for a system can be defined as the average resource usage times transaction arrival rate.

4.2.2 Simulation Results

This section presents the simulation results for the performance evaluation of the proposed OCC algorithms under various system workloads and different levels of data conflict. For each experiment, 20 runs with the same parameter settings are performed. Each run of the simulation continues until at least 20000 transactions are executed. The statistic data of an

Table 4.1: Common settings for the simulation.

parameter	value
$NumCPU$	10
$NumDisks$	20
$CPUAccessTime$	10 ms
$DiskAccessTime$	20 ms
$DBsize$	1000 pages
$NumPage$	16 pages

experiment is then collected and averaged over the 20 runs. Totally, we have 11 algorithms to be compared. In order to make the comparison more readable, we separate them into two groups. The first group, G1, contains four variations using wait strategy (namely WAIT-ALL, WAIT-NR, WAIT-S, and WAIT-L) and REMOVE-NR; The other group, G2, has the baseline algorithm OPT-BC, REMOVE-AB, and the variations using sacrifice. We will present the simulation results for each group. The performance comparison of the best algorithms will be shown in the conclusion. Since most proposed algorithms make use of the knowledge of the execution time, it is necessary to investigate the effect of the error on the estimation of the execution time. In the following, we will discuss how we conduct the experiments for the estimation error and their results.

Our primary performance metric is miss ratio, defined the number of transactions that miss the deadlines over the total number of transactions, which is a major concern in RTDBS. The parameter settings which are common to the evaluation of the proposed algorithms are listed in Table 4.1. We intend to choose the values which can simulate a general multiprocessor RTDBS and can display the performance differences among the proposed algorithms. We conduct the experiments with different setting on the write probability. Due to space limitations, we only show a subset of the results which best illustrate the behavior and the performance of the algorithms.

Comparisons on the First Group

In this section, we discuss the performance of the algorithms in the first group. We vary the inter-arrival time to evaluate the algorithms under various workloads and data conflict rates. High write probability results in high data conflict rate. Our simulation assumes that high load settings refer to the IAT up to 120 ms, that normal load settings refer to IAT between 120 and 160 ms, and that low load settings refer to IAT greater than 160 ms. Figures 4.3 to 4.5 show the miss ratios against the inter-arrival time with three different write probabilities. The graphs are plotted based on the experiments with the slack rate ranging from 3 to 3.5. The results for looser slack rates are experimented. Figures 4.6 and 4.7 present the results for the slack rate ranging from 3 to 12. Since the various slack rates we experiment convey the similar behavior, unless stated otherwise, we only show the simulation results with the slack rate ranging from 3 to 3.5.

Figure 4.3 graphs the results for low data conflict rate ($W_p = 0.25$). As expected, all the strategies yield larger miss ratios as the workload increases. REMOVE-NR is superior for nearly all load settings, but its performance margin over the others narrows, as the workload decreases. WAIT-L misses the most deadlines for nearly all loads. This is not surprising because it lets the validating transaction commit only if all the conflicting transactions are short and restarts all these short transactions which might complete soon. Such inefficiency lengthens the response time. To confirm this observation, Figure 4.8 plots the average response time in the corresponding experiments and shows that WAIT-L has the longest response time over various loads.

Observing Figure 4.4, we see that REMOVE-NR has the best performance and that its relative performance to the other algorithms remains. As the conflict rate is high, in Figure 4.5, REMOVE-NR still performs best; WAIT-L performs poorly at heavy and normal loads; WAIT-ALL yields larger performance gap between the others than the previous two cases ($W_p = 0.25$ and 0.50). In the following, we will examine the statistic data collected during the experiments to further investigate the behavior of the algorithms.

The graphs of the statistic data for different write probabilities are similar, so we only illustrate the graphs for $W_p = 0.75$. Figure 4.9 confirms our expectation that WAIT-L has longer response time than the others. Except WAIT-L, all the other algorithms have comparable response time. They have convex curves: short response time at high and low loads and long response time at normal loads. In high loaded systems, high resource contention causes that the committed transactions most likely complete without being restarted and the restarted transactions might miss the deadlines. Note that the average response time only counts for the committed transactions. Therefore the response time is short under heavy loads. As the load decreases from high to normal loads, the restarted transactions become possibly to commit and they contribute longer response times. At low load settings, transactions arrive sparsely and data contention becomes less significant; transactions can complete without being restarted or conflicting transactions, so the response time becomes short again. WAIT-NR and REMOVE-NR have slightly longer response time than WAIT-S and WAIT-ALL. It implies that the former strategies, using restartability concept, commit more number of transactions and have lower miss ratio.

Figure 4.10 shows that, as expected, the number of active transactions increases when the workload increases. WAIT-ALL has the most number of active transactions. Unlike the others, WAIT-ALL does not restart, but rather waits for the conflicting transactions to complete. It keeps transactions in the system until they either commit or miss the deadlines. Therefore it has more active transactions than the others where restarts might occur. In the figure, we see that WAIT-L has the least number of active transactions. It is primary because WAIT-L waits for long transactions, while long transactions stay at the system long. Therefore, in WAIT-L, transactions in the system tend to be long and the level of concurrency is low. Such system results in less active transactions in the system.

Figure 4.11 graphs the wait time of the algorithms, excluding REMOVE-NR which has no wait. In REMOVE-NR, the active transactions either utilize the resources or wait for the resources, while, in the other algorithms, the active transactions might wait for some transactions to complete and let the resources set idle there. However, large number of active transactions does not mean low miss ratio, since wait time is another factor. We expect that WAIT-ALL has the longest wait time, because it waits for *every* conflicting transaction, and Figure 4.11 ensures our observation. This is the reason why it cannot perform well, even though it has the highest level of concurrency. The graph in this figure has the pattern that the wait time increases as the workload decreases, but it drops at lowest loads. In the previous paragraph, we learn the reasons that the response time is short at high loads. This figure confirms the reason that short response time results from short wait time. WAIT-L yields an approximate 40 percent decrease in wait time over WAIT-ALL. As expected, WAIT-S and WAIT-NR have less wait time. The average wait time of WAIT-NR becomes smaller as the arrival rate decreases, since the transactions are more possibly restartable at low loads.

The statistic data shown in the above figures illustrates that WAIT-S has short response time and short wait time. In Section 4.1, we learn that it tends to commit short transactions, which matches our finding in the above figures.

Comparisons on the Second Group

Figures 4.12 to 4.14 plot the miss ratios under various load settings and three different write probabilities. In Figure 4.12, the baseline algorithm OPT-BC performs worst at high and normal load settings, it becomes comparable at low loads, REMOVE-AB outperforms the baseline at most loads, and both graphs decrease steeply when the workload decreases. As we observed that both efficiency is based on the assumption that data conflicts are rare, therefore, they perform comparably at low arrival rates. The same observations hold true for different write probabilities as well (Figure 4.13 and 4.14). In Figure 4.12, SACRIFICE-L and SACRIFICE-NR have nearly identical performance and are superior to the others. SACRIFICE-S performs slightly better than SACRIFICE-ALL. Examining Figures 4.13 and 4.14, we see that the performance margin of SACRIFICE-L and SACRIFICE-NR over SACRIFICE-S and SACRIFICE-ALL becomes significant, that SACRIFICE-L is slightly better than SACRIFICE-NR at high loads, and that SACRIFICE-NR is slightly better than SACRIFICE-L at low loads. As the range of the slack rate becomes wide, (graphs are not shown) we find that SACRIFICE-NR performs best, that its performance gap over SACRIFICE-S and SACRIFICE-ALL is still significant, and that SACRIFICE-L becomes not as comparable as SACRIFICE-NR. We will explain the performance behavior later when we discuss the statistic data.

Like the graphs in the previous section, the curves of the response time, shown in Figure 4.15, are convex. The same reason holds true for this case. At high loads, OPT-BC has the least response time and REMOVE-AB has slightly larger response time, because the later algorithm has lower miss ratio and commits more transactions. As the workload decreases, from normal to low loads, the later algorithm has the least response time, since it does not restart, while the others do. SACRIFICE-S is expected to have long response time. Based on the assumption of SACRIFICE-S described in Section 4.1.9, a short transaction might be restarted several times before it commits, so the average response time tends to be long. The figure ensures our observation. The response time of SACRIFICE-L is the least among those algorithms with sacrifice. The reason for this behavior is similar to that for WAIT-S, described in the above section.

Figure 4.15 plots the level of concurrency under various loads. Except OPT-BC and REMOVE-AB, the average number of active transactions raises as the workload increases. REMOVE-AB has the least level of concurrency at normal and light loads, because of the nature of the algorithm: aborting transactions. At high loads, REMOVE-AB has higher level of concurrency than OPT-BC. This confirms our observation mentioned in the previous paragraph that REMOVE-AB commits more transactions at high loads. OPT-BC has comparably high number of active transactions at normal and low loads, because the restarted transactions keep in the system. SACRIFICE-ALL has the highest level of concurrency, mainly because it restarts only validating transaction and keeps all conflicting transactions in the system. Recalling the definition of the active transactions in Section 4.2.1, we can see that restart will decrease the level of concurrency at high loads. SACRIFICE-ALL does not restart as often as the other algorithms in this group, so it behaves as we expected. SACRIFICE-S has almost identical level of concurrency with SACRIFICE-ALL at high and normal loads and its level of concurrency is slightly lower at light loads. We know that long transactions last long in the system. At high and normal loads, it is more possible that a validating transaction conflicts with a long transaction, consequently SACRIFICE-S behaves like SACRIFICE-ALL. By contrast, SACRIFICE-L has the less level of concurrency at most loads.

Observing Figures 4.15 and 4.16, we learn that, SACRIFICE-L has shorter response time and lower level of concurrency than SACRIFICE-S. This is mainly because SACRIFICE-L favors short transactions and tends to complete short transactions. Hence it has lower miss ratio. By contrast SACRIFICE-NR does not bias on short or long transactions. Its response time and level of concurrency are between theirs. SACRIFICE-NR restarts the conflicting transactions only if they are restartable. Overall, it has the less miss ratio in this group, because of the concept of restartability.

The Effect of the Estimation Error

Most of the proposed concurrency control algorithms depend on the estimation of the execution time. To study how error of the estimation affects the performance of the algorithms, we devise three experiments. The first experiment adds a random error, choosing from -0.5 to 0.5, into the estimation; the second experiment biases the estimation in one direction such that the execution time is over-estimated (i.e., $E' = E * (1 + 0.5)$, where E is the execution time with no error and E' is the estimated execution time); the third experiment under-estimates the execution time with the same error (i.e., $E' = E * (1 - 0.5)$). The results of the experiments are compared with the results of the baseline experiment which has zero estimation error.

Figure 4.17 graphs the miss ratio of SACRIFICE-S on different estimation errors with $W_p = 0.75$. SACRIFICE-S yields performance changes in the first experiment, but the performance difference between the baseline and the first experiment is very little. This algorithm uses the relative length of transactions to resolve conflicts. Since the second and the third experiments bias the error in one direction and the relative length remains in such

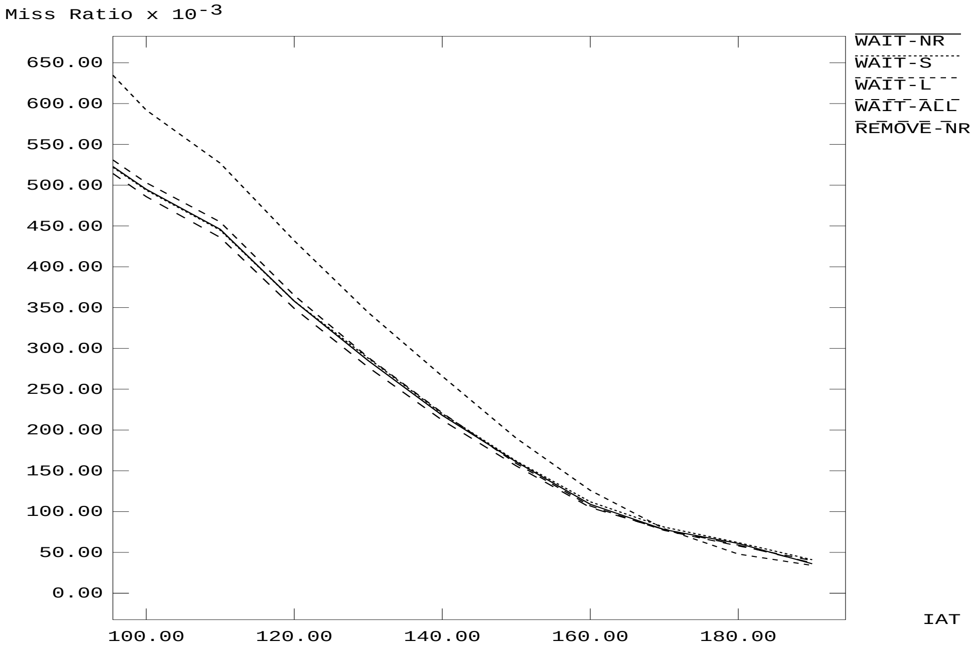


Figure 4.3: Miss ratios of G1 for $Wp = 0.25$.

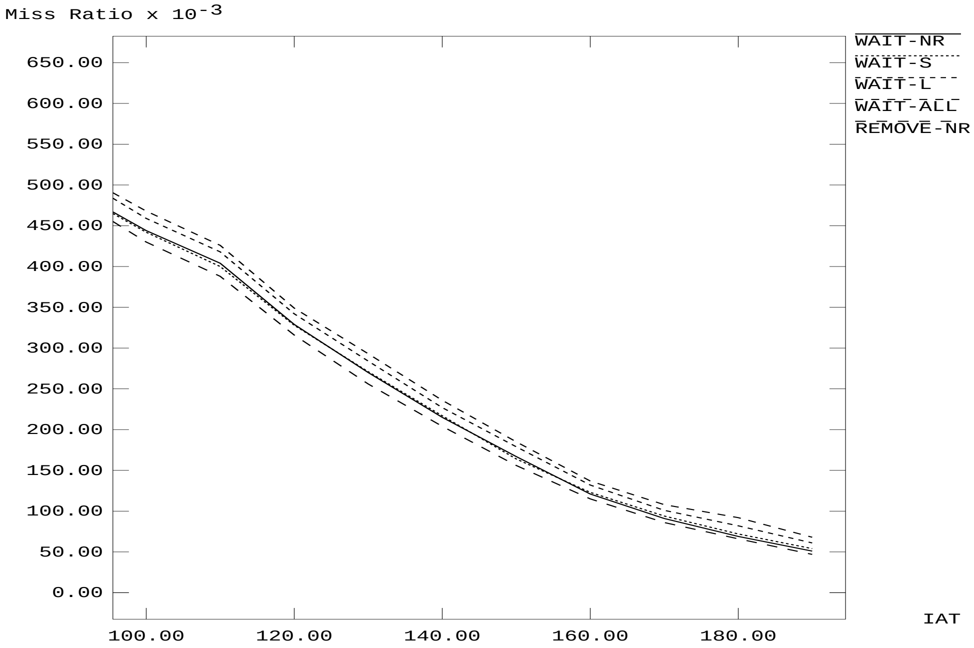


Figure 4.4: Miss ratios of G1 for $Wp = 0.50$.

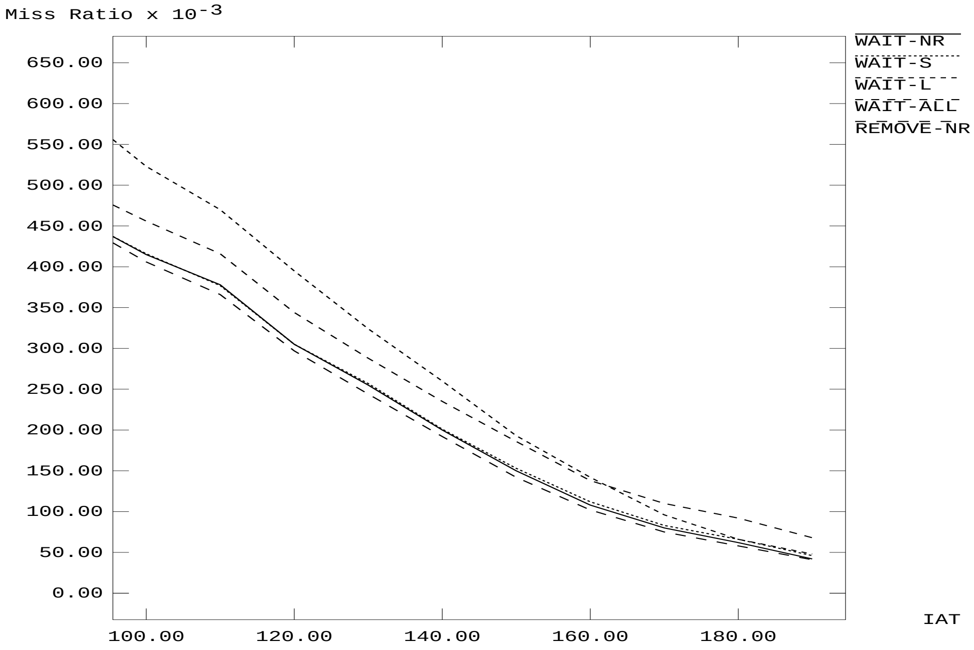


Figure 4.5: Miss ratios of G1 for $Wp = 0.75$.

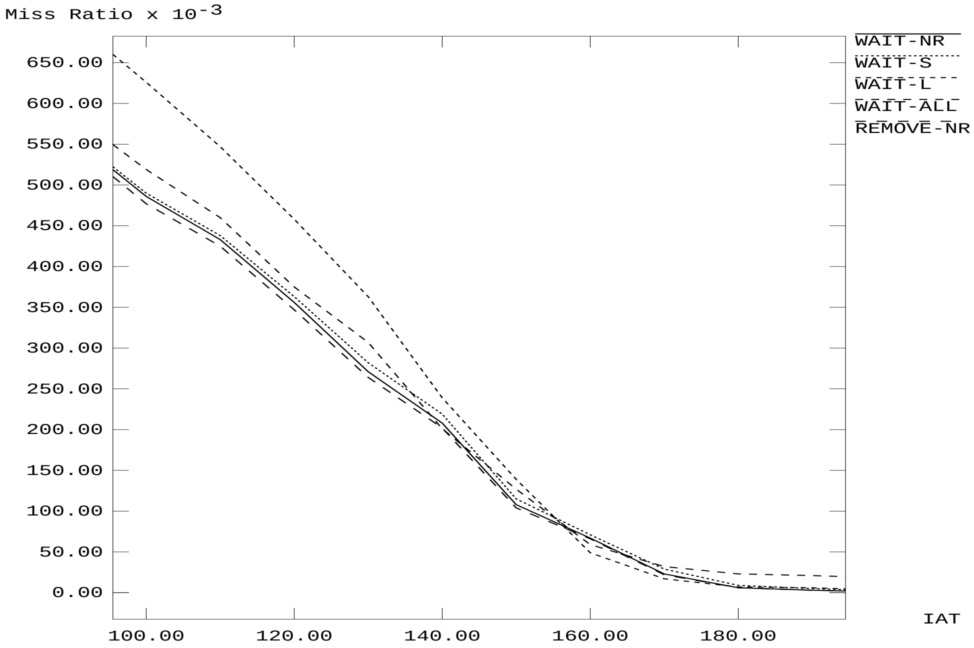


Figure 4.6: Miss ratios of G1 for $Wp = 0.25$ and the wide range of slack rate.

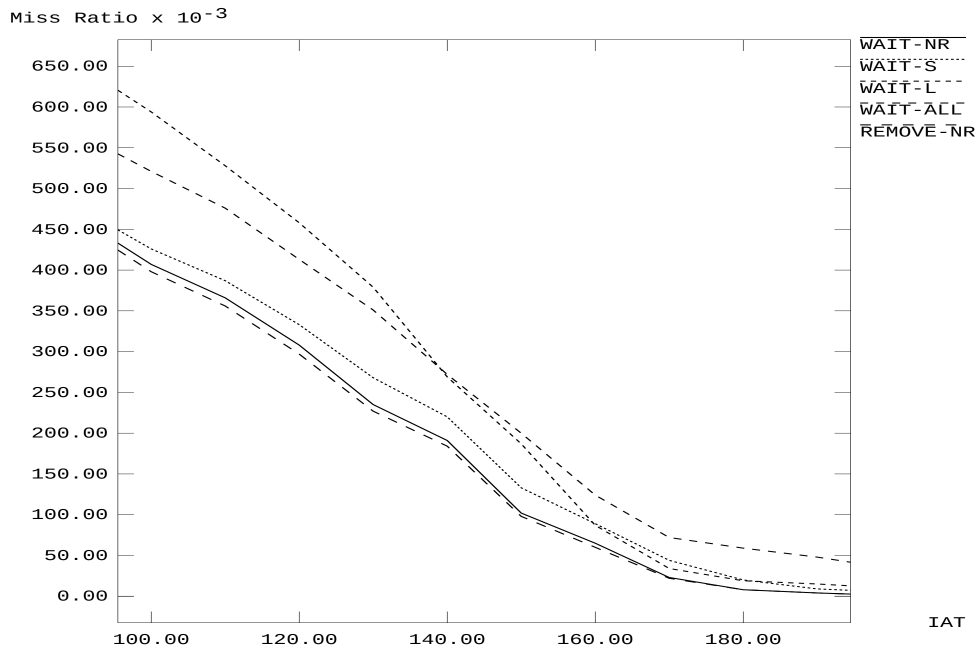


Figure 4.7: Miss ratios of G1 for $Wp = 0.75$ and the wide range of slack rate.

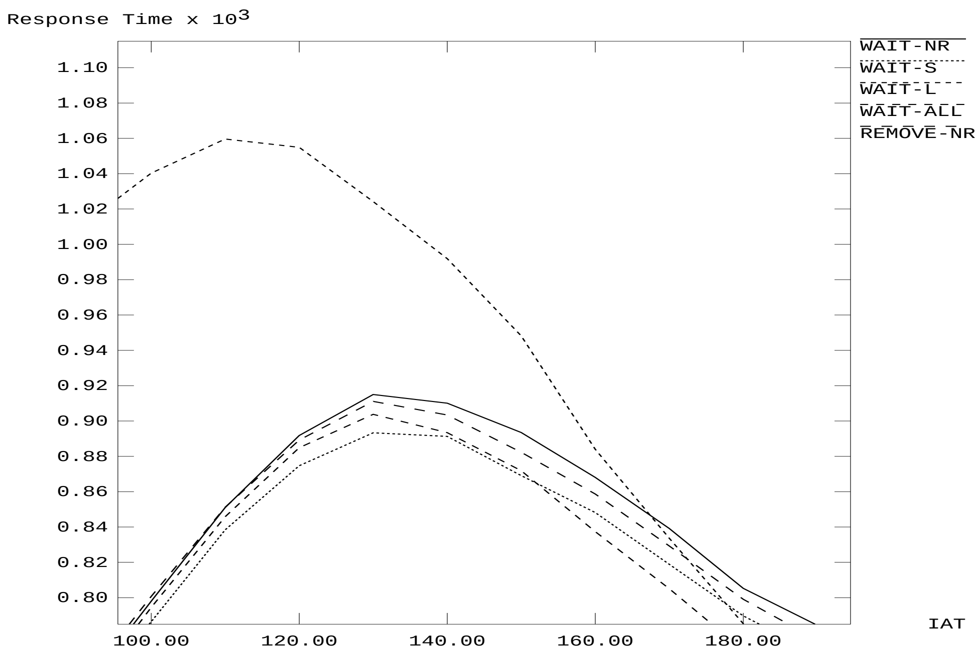


Figure 4.8: Response times of G1 for $Wp = 0.25$.

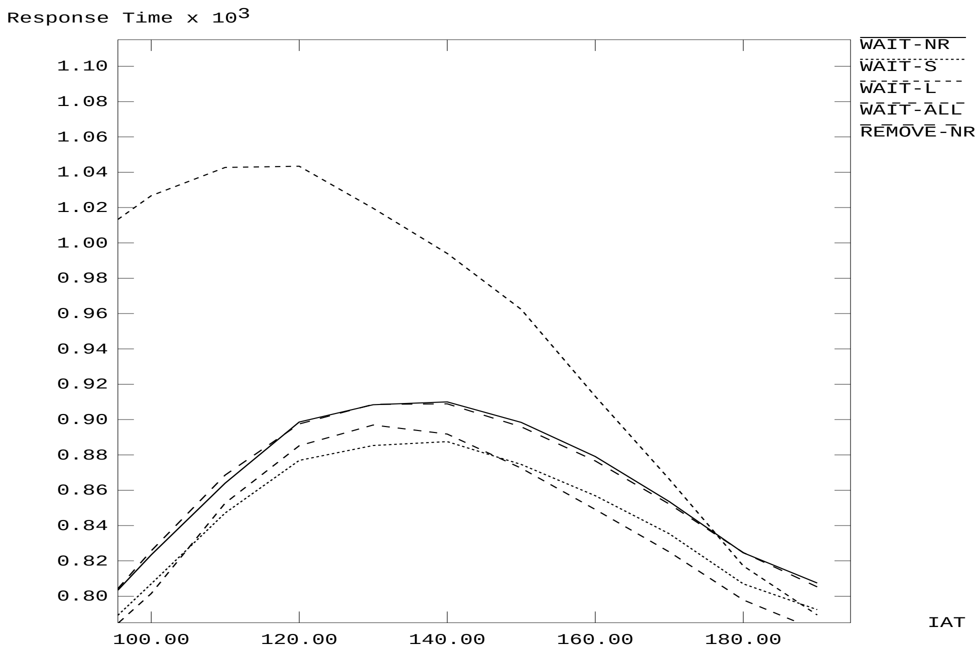


Figure 4.9: Response times of G1 for $Wp = 0.75$.

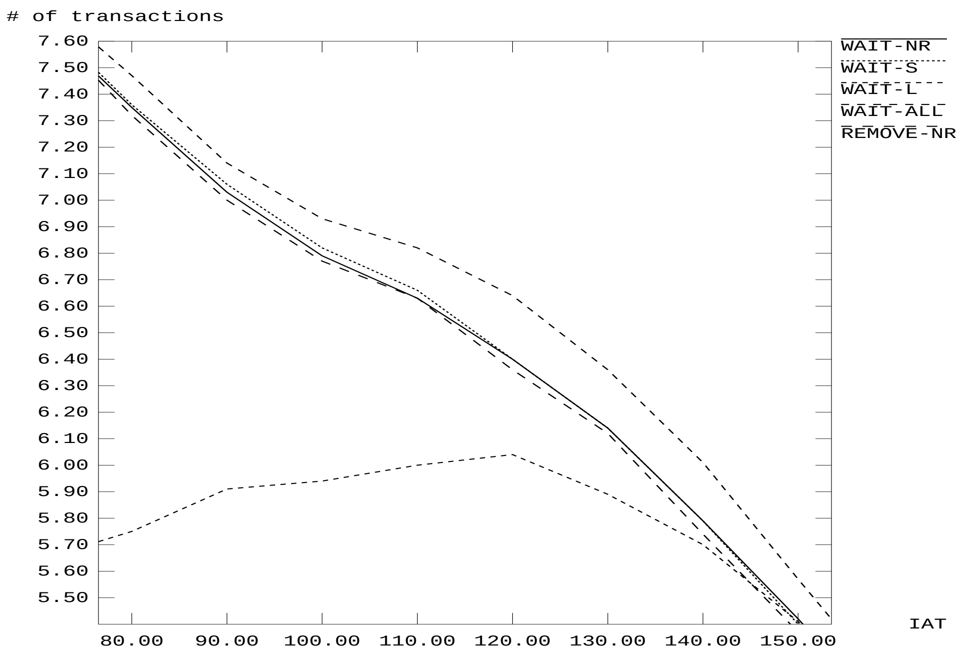


Figure 4.10: Concurrency level of G1 for $Wp = 0.75$.

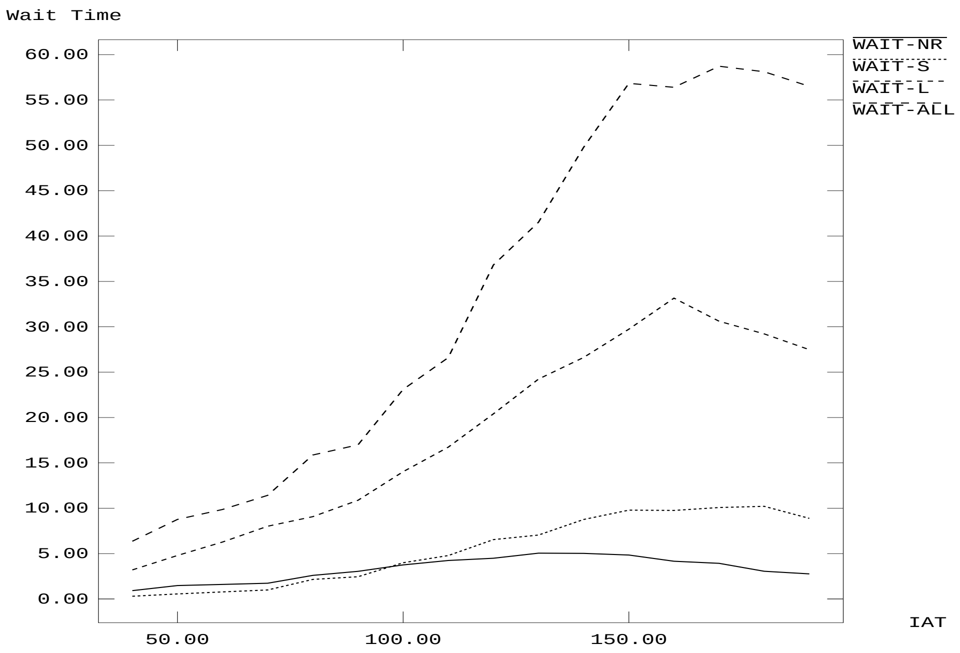


Figure 4.11: Wait time of G1 for $Wp = 0.75$.

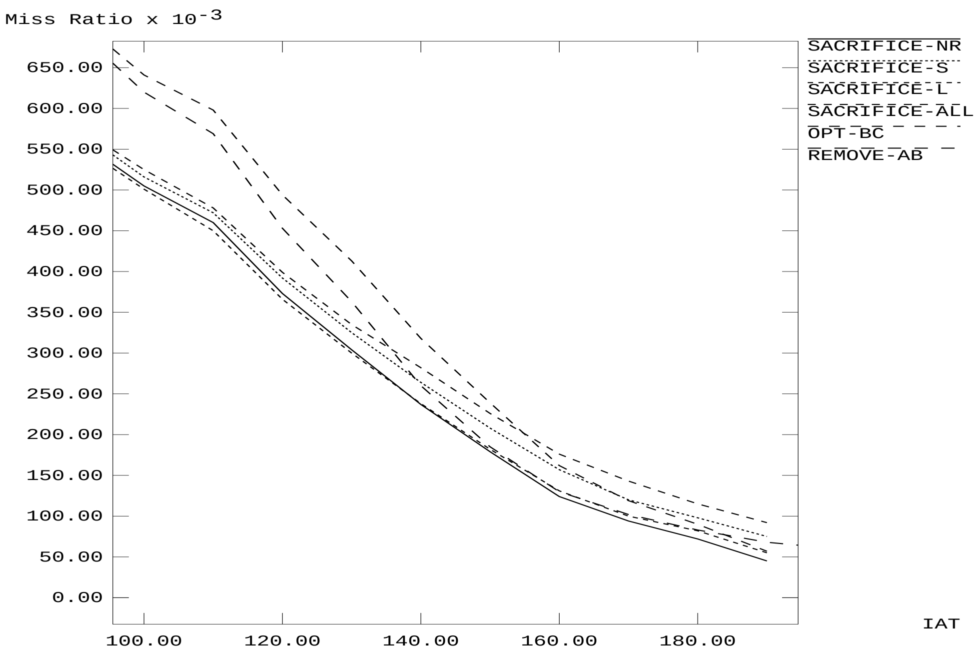


Figure 4.12: Miss ratios of G2 for $Wp = 0.25$.

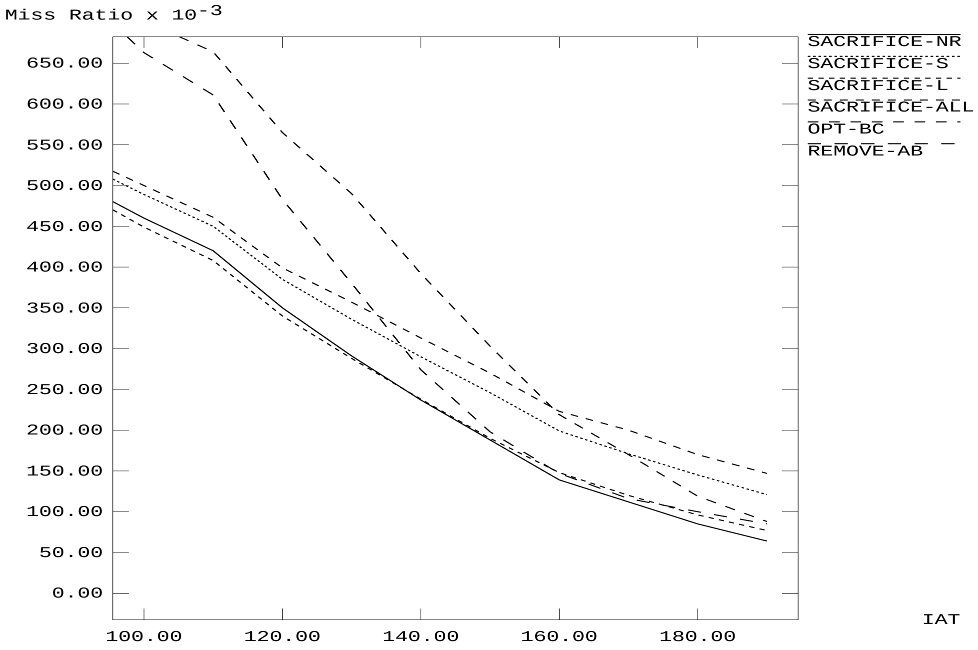


Figure 4.13: Miss ratios of G2 for $Wp = 0.50$.

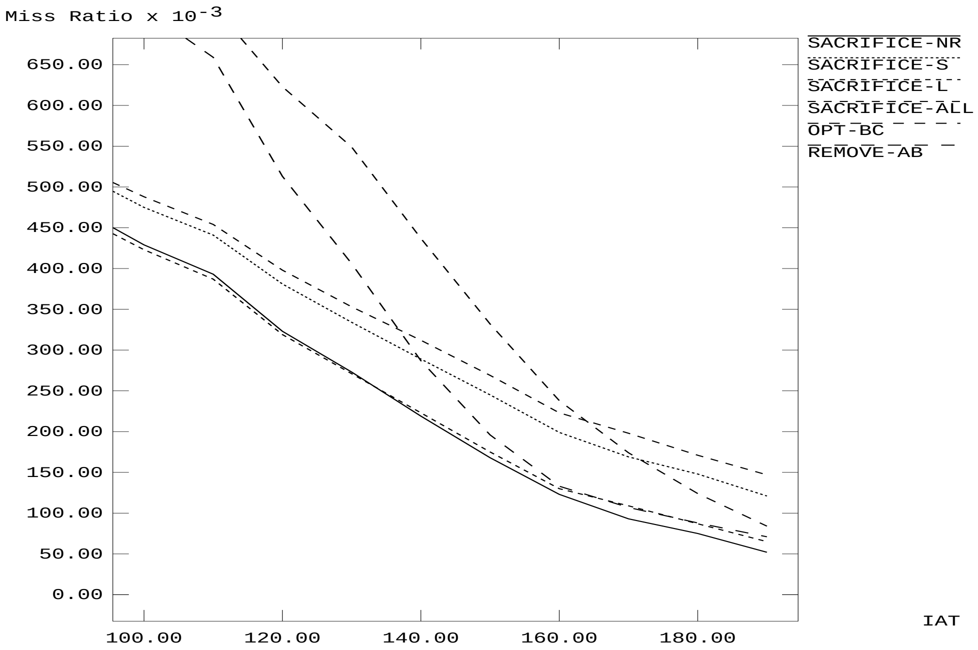


Figure 4.14: Miss ratios of G2 for $Wp = 0.75$.

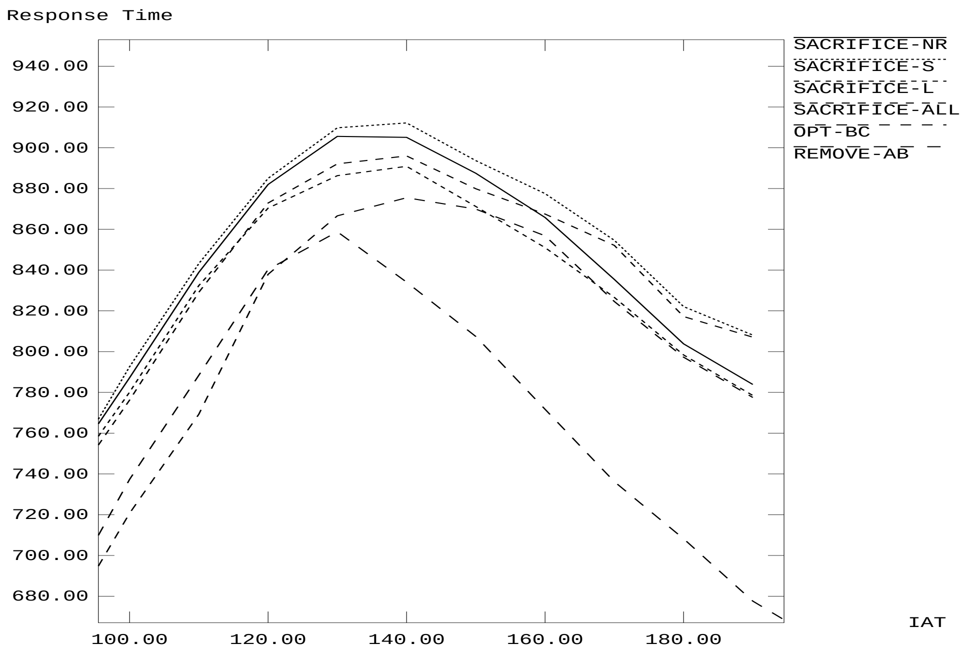


Figure 4.15: Response times of G2 for $Wp = 0.75$.

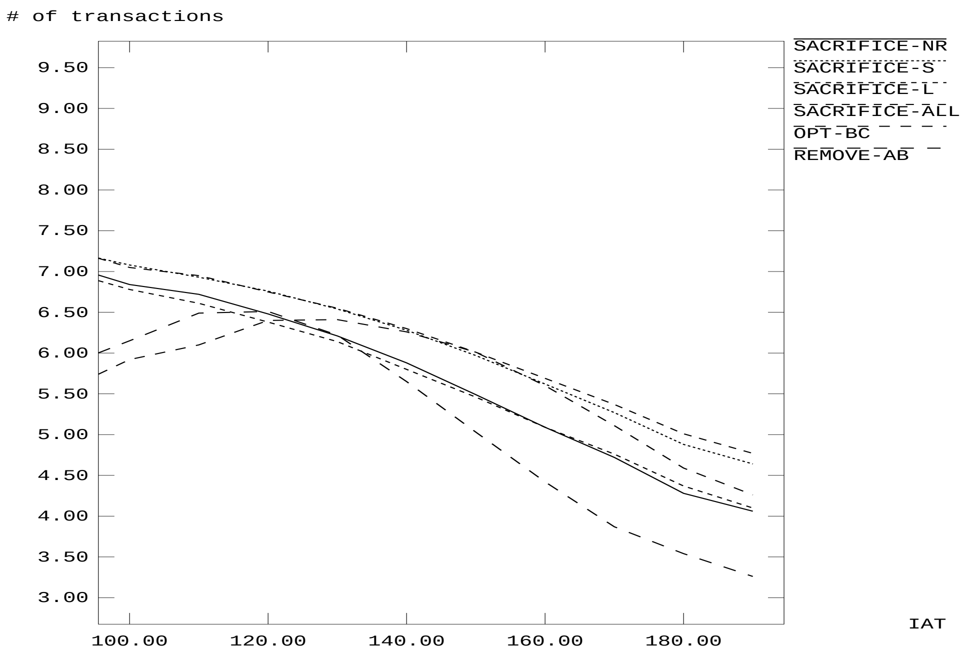


Figure 4.16: Concurrency level of G2 for $Wp = 0.75$.

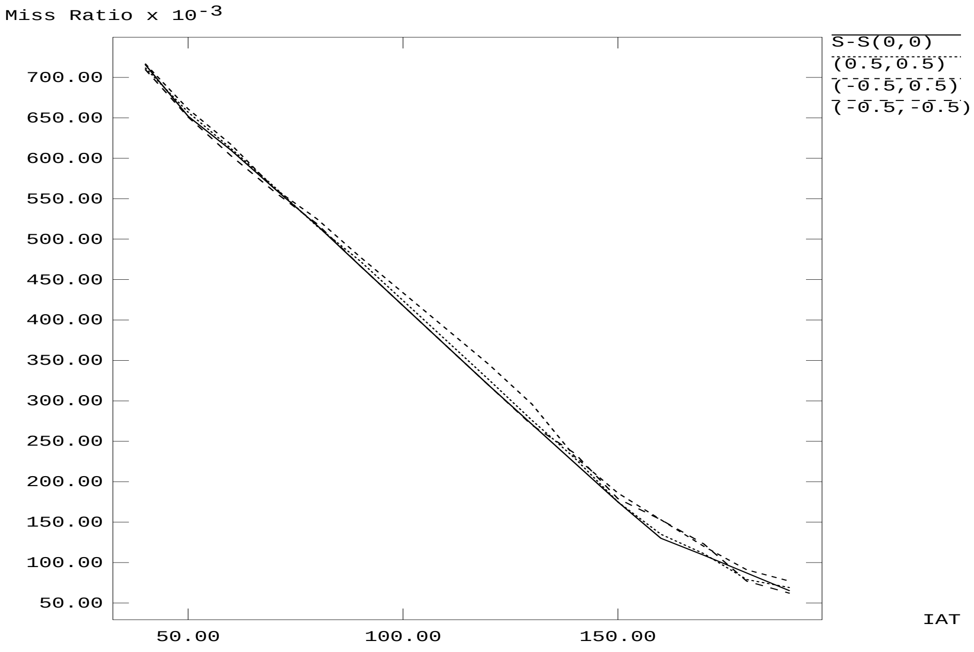


Figure 4.17: The effect of the estimation error on SACRIFICE-S for $Wp = 0.75$.

cases, the algorithm behaves nearly the same as the baseline experiment. We predict that WAIT-L, WAIT-S, and SACRIFICE-L will have the similar behavior on the estimation error, because they all use transaction size to direct conflict resolution decision.

Figure 4.18 plots the results of WAIT-NR. The figure shows that the miss ratio is slightly improved when the execution time is under-estimated. Since the number of non-restartable transactions decreases, when the execution time is under-estimated, the algorithm waits less for non-restartable transactions and commits the validating transaction. Less waiting results in a little bit better performance. By contrast the performance is slightly poor in the case of over-estimation. The first experiment performs in between of the other two experiments, since some transactions have less waiting and some have more. REMOVE-NR and SACRIFICE-NR have similar graphs and hence we eliminate them.

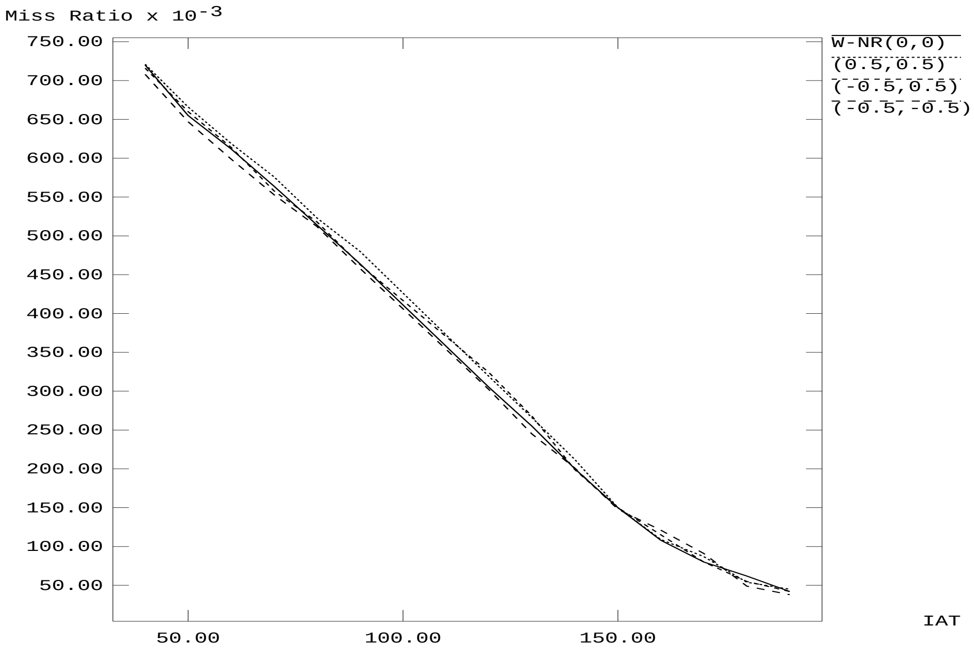


Figure 4.18: The effect of the estimation error on WAIT-NR for $Wp = 0.75$.

Chapter 5

Fault-Tolerant Scheduling

System reliability is becoming an increasing important factor in evaluating the behavior of real-time systems, because the result of a real-time application may be valid only if the application functions correctly even with underlying faults and its timing constraints are satisfied. Fault-tolerance is a technique to enhance the reliability in case faults may occur. A common approach used for fault-tolerance is through redundancy in time or in space. Temporal redundancy is to activate a process at different time, while spatial redundancy is to activate several copies of a process at the same time using different resources. Ideally, we can achieve fault-tolerance by fully temporal redundancy or fully spatial redundancy. However, since real-time systems have timing requirements. Temporal redundancy is not always applicable to the tasks with tight timing constraints. Besides, a real-time system may not have enough resources to provide spatial redundancy for *every* tasks. Combining both redundancy techniques might be able to meet the fault-tolerance goals as well as timing requirements. To distinguish them, we call spatial redundancy replication and temporal redundancy rollback.

Oh and Son [OS94] prove that scheduling a set of non-preemptive hard real-time tasks to tolerate one processor fault is NP-complete. Since such fault-tolerant scheduling problems are computationally intactable, we develop a heuristic fault-tolerant scheduling algorithm, a hybrid method of spatial and temporal redundancy for the real-time systems which require both system reliability and the guarantee of meeting deadlines. Based on the original schedule without consideration of fault-tolerance, the proposed method utilizes the available free slots and includes redundant copies of the tasks into the schedule. Some tasks have rollback copy, if they have enough slack time to rerun the task; others have replicated copy, if their timing constraints are tight. Our approach incrementally increases the fault-tolerance capability without violating the real-time requirements. It can be deployed to the systems with off-line scheduler. In the next chapter, we will devise analytical models to estimate the reliability of the systems using the proposed approach.

The rest of the chapter is organized as follows. In the following section, we define an abstract model for real-time systems (RTS) under which the proposed fault-tolerant scheduling and analytical models can be deployed. In Section 5.2, we present the proposed fault-tolerant scheduling. Section 5.3 discusses a rescheduling algorithm to incrementally put a replicated or rollback copy into the schedule and Section 5.4 describes an approach to migrating the workload of one processor to other processors.

5.1 Abstract System Model and Assumptions

Our system model is a real-time system consisting of a set of processors and a set of periodic tasks. The system uses off-line scheduling to guarantee to meet the timing constraints of all tasks and adopts rollback or replication technique as fault-tolerance strategy. Let the elemental unit (EU) be the smallest non-preemptable execution unit [MA91]. Each EU is characterized by its release time, execution time, and deadline. Each task can be described by an elemental unit graph (EUG), where an EUG is a directed acyclic graph; each node is an EU and each directed edge represents the precedence relation and communication pattern.

Since tasks are periodic, the window size for examining the execution behavior of the cyclic tasks can be the least common multiple (LCM) of the periods of all the tasks. We define such time interval as a *frame*. Let *minframe* of an EU be the time interval between its release time and its deadline. We adopt a fault model which permits transient and permanent faults with fail-stop behavior [SS83], that is, non-Byzantine [LSP82]. We assume that the inter-arrival time between two consecutive transient failures in a system is greater than the longest minframe. It ensures that if a failed EU instance has a redundant copy it can recover without experiencing a second transient fault. If a processor fails due to a permanent fault, we assume that it will be replaced by a new processor within a *frame* period of time. The assumption that the inter-arrival time between two consecutive permanent failures is greater than a *frame* period of time is to make sure that a system has at most one failed processor at any time.

Since each EU is the smallest non-preemptable executable unit, we can do checkpointing for each EU before it exits. In this way, we do not have to restart the whole task if one of its EUs fails. Instead, we can roll back the failed EU or get the result from its replicated copy. Assume that the save time is equal to the load time for a given EU and the save time is included in the execution time of the EU. We also assume that the system has a fault detection mechanism which can detect faults before the failed EUs do checkpointing. To ensure the correctness of the saved checkpoint, an acceptance test is applied to the checkpoint before checkpointing is done [LA90]. The system recovers from a transient failure once the failed EU rolls back or we obtain the result from its replicated copy. We define that an EU is fault-tolerant if it has a replicated copy or a rollback copy to cope with transient failures.

Although our fault model only considers transient and permanent faults, we believe that they are the majority of the faults in a system. The assumptions on fault inter-arrival time are used to simplify our analytical model for estimating reliability. We believe that faults should not happen so often as task arrivals and the assumptions are reasonable and suitable for real systems.

The task model of the underlying system model is adopted from a hard real-time operating system developed by the University of Maryland which has a prototype for distributed systems. Besides, the task model can be extended easily without major modification. A task in the system is represented by a task graph [CR72] where each vertex represents a module and a directed edge (i, j) is in the task graph if and only if module i is followed by module j with non-zero probability. The precedence relationship of modules is determined by the task graphs and a module considered here is the smallest non-preemptable execution unit. Such task model has been adopted by many researchers [Upa90, US86, Gel79, NK83, GRW88] to investigate rollback recovery. The underlying system model may be applicable to real systems.

5.2 The Proposed Adaptive Redundancy Approach

Our scheduling objective is twofold. The primary objective is to provide fault-tolerance capability with the guarantee of meeting deadlines, and the secondary objective is to save resources for aperiodic tasks. Rollback is more favorable than replication, since it can achieve both objectives. However, when rollback cannot satisfy our primary objective, we have to choose replication.

Our approach, called adaptive redundancy (AR), can be briefly described as follows. For each EU instance x, first we try to use temporal redundancy (i.e., rollback) to achieve fault-tolerance. The adjustment of the schedule is temporary, i.e., the modified schedule is used only when the fault occurs; otherwise, the original schedule is used. In other words, the system does not pre-allocate resources to the rollback copy of EU instance x, but it does mark the resources dedicated to the rollback copy such that when the instance x experiences a fault it can rollback to recover. We say that an EU instance has stringent timing constraint if it cannot use temporal redundancy to avoid transient faults. If rollback fails to satisfy our primary objective, in the next step, we attempt to use spatial redundancy (i.e., replication) to avert transient faults. The new schedule with the replicated EU instance is used hereafter, if the modified schedule meets all deadlines.

AR method is dynamic in that temporal redundancy is used only when a fault actually happens. Our approach is adaptive. We do not replicate the whole application; we only replicate the EU instances with stringent timing constraints in order to achieve fault-tolerant execution and high resource utilization.

We use an example to illustrate our approach. Suppose we have a two-processor system running one application. The EUG and the execution time, release time, and deadline of each EU are given in Figure 5.1(a); the original schedule, before applying our approach, is shown in Figure 5.1(b); and the final schedule, after applying our approach, is shown in Figure 5.1(c). Let us first examine EU-1. As described above, we first attempt to put the rollback copy of EU-1 after the primary copy; we want to see if EU-1 can have a rollback to avert a transient fault. We find that EU-1 cannot roll back to skip a transient fault, since the rollback copy will finish at time 20, which exceeds its deadline. We then try out second strategy (replication); therefore, it is replicated. The replicated copy of EU-1, $1'$, runs on processor $\mathcal{P}_2$. Using the same technique, we find that the rollback copy of EU-2, $2'$, can be completed before its deadline, so it can use rollback to achieve our objectives.

The resources will be marked so that the rollback copy can use them if a transient failure occurs. Similar argument applies to EU-3.

We assume that a feasible schedule[1], with the guarantee of meeting all deadlines, is given. We call the original schedule *non-fault-tolerant* schedule (NFT schedule) and that resulting schedule from our approach is called *fault-tolerant* schedule (FT schedule). An NFT schedule contains the scheduling times for the *primary* copies of all EU instances; an FT schedule contains, besides primary copies, the scheduling times for the rollback or replicated copies of EU instances.

The skeleton of our proposed approach is shown in Figure 5.2. Since applications are periodic, an application can have many instances of the application within a *frame*; thereby an EU can have many EU instances in a *frame*. All EU instances need to apply *replication test*, which checks if an EU can use rollback or replication to achieve fault-tolerance. The detail algorithm for *replication test* is given in Figure 5.3. A data structure called *scheduling queue* keeps track of the instances that wait to be scheduled. Part I considers temporal redundancy. The routine *rescheduling* is to adjust the schedule when a (primary or redundant) copy of an instance is put into the scheduling queue. Part II considers spatial redundancy. When neither rollback nor replication can achieve our objective, in step II.4, we discard the replicated copy from the scheduling queue and the EU is not fault-tolerant. The schedulability of an FT schedule depends on the order of applying the test and the complexity of the rescheduling algorithm. We will propose a rescheduling algorithm and show how to apply the test in the next section. However, rescheduling algorithm is not limited to the one we are going to present; other rescheduling algorithms can be used.

5.3 A Proposed Rescheduling Algorithm

Because of our objectives mentioned above, temporal redundancy should be considered first. Let all EU instances go through the part I of *replication test* first and then the part II. We apply the test according to the ascending order of the ID of the EU instance.

The rescheduling algorithm keeps track of all the free slots all the time. When a rollback copy is put into the scheduling queue, The algorithm first tries to schedule it on the same processor where the primary copy is assigned. In order not to affect the scheduling times of the successors and predecessors of an EU instance, we define two terms, the earliest start time (EST) and the latest finish time (LFT), to quantify a scheduling window for the redundant copy of an EU instance. Let EST of an EU instance be the scheduled start time of the primary copy of the EU instance and LFT be the earliest scheduled start time of all the immediate successors of the EU instance. With such scheduling window, the algorithm can search the free slots of a given processor, such as the processor where the primary copy is assigned, to see if there is any slot within the range of the scheduling window. If it can not find such free slot, it will try to schedule both the primary and rollback copies on other processor based on the similar strategy. After all EU instances apply the part I of *replication test*, we have a partial FT schedule.

[1]The schedule includes the allocation of all EU instances and the resource allocation information.

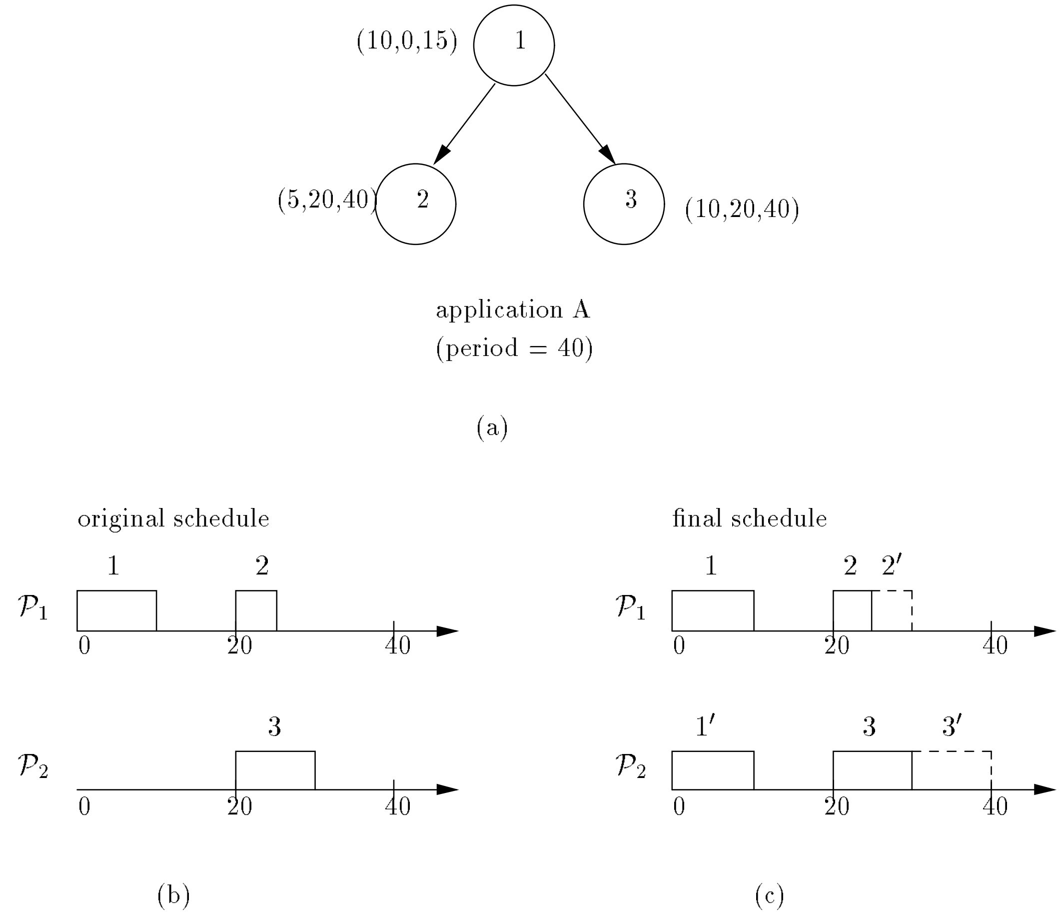


Figure 5.1: An example for illustrating AR method.

Input: NFT schedule

Decide a permutation δ which specifies the order of the rollback and replicated copies of all EU instances in a *frame*;
Apply *replication test* according to δ;

Output: FT schedule

Figure 5.2: The skeleton of AR method.

Part I:

I.1 Add the rollback copy x_i^{roll} of x_i into the scheduling queue;
I.2 Call *rescheduling*;
I.3 If the adjusted schedule meets all deadlines,
    x_i passes *replication test* and exits;
I.4 Delete the rollback copy from the scheduling queue;

Part II:

II.1 Add the replicated copy x_i^{repl} of x_i into the scheduling queue;
II.2 Call *rescheduling*;
II.3 If the adjusted schedule meets all deadlines,
    x_i passes *replication test* and exits;
II.4 (The schedule cannot accommodate the replicated copy.
    If x_i experiences a fault, the system can not recover.)
  Delete the replicated copy from the scheduling queue;
    (so that the system still can meet all deadlines
    and complete the mission if
    x_i does not experience a fault.)

Figure 5.3: *Replication test.*

The following step is to apply the part II for those without rollback copies in the schedule. Again, to avert the cascade effect mentioned above, we use EST and LFT to quantify a scheduling window for replicated copies. Searching the list of all free slots, we can construct a list of candidates of free slots which can accommodate the given replicated copy according to the scheduling window of the EU instance.

The proposed rescheduling algorithm for the replicated copies is an extension of graph matching algorithm [BM76]. First, we construct a bipartite graph G describing the possible assignments for all the replicated copies. Let $T \cup S$ be the set of vertices and E be the set of edges, where T is the set of EU instances that need to go through the part II of the test; S is the set of free slots in the system; edge (x_i, s_j) is in E if the replicated copy of EU instance x_i can be scheduled in the free slot s_j.

The replicated copies with only one edge in the bipartite graph should be assigned first, since they do not have another choice of free slot. Having such initial assignment, we then apply matching algorithm for the rest of replicated copies. We obtain the assignment for the replicated copies of a subset of T, called $\tilde{T}$, and a new set of free slots, $\tilde{S}$, left after the assignment. Iteratively constructing a bipartite graph[2] as described above and applying the matching algorithm until no more new assignment can be added, we finally get an FT schedule. The pseudo code for the proposed method is given in Figure 5.4.

If we implement the data structure, the set, used throughout the algorithm as link

[2] $(T - \tilde{T}) \cup \tilde{S}$ will be the vertex set in the next iteration.

list, the time complexity for constructing a bipartite graph is $O(|T||S|)$ and that for the matching algorithm is $O(|T|^3 + |T||S|)$, where $|V|$ is the number of the elements in the set V. In the worse case, the number of iterations we need to get all the assignments is $|T|$; it happens when each iteration adds one new assignment. Therefore, the complexity for the proposed method is $O(|T|^4 + |T|^2|S|)$.

5.4 A Proposed Migration Algorithm

In case of a permanent fault, the system needs to migrate the EU instances on the failed processor to other processors. In this section, we present a migration algorithm for the proposed approach. We call the processor which experiences a permanent fault *migrated processor* and the schedule after the migration algorithm is applied *migration schedule*. The algorithm is based on the technique described in Section 5.3. EST and LFT defined in Section 5.3 are used to quantify a scheduling window for the EU instances to be migrated. According to the scheduling window of each EU instance to be migrated [3], we build up a list of free slots which can accommodate it.

The proposed migration algorithm is an extension of graph matching algorithm [BM76]. We construct a bipartite graph G describing the possible assignments for all the EU instances to be migrated. Let $T \cup S$ be the set of vertices and E be the set of edges, where T is the set of EU instances to be migrated; S is the set of free slots on the processors other than the migrated processor; edge (x_i, s_j) is in E if the EU instance x_i can be scheduled in the free slot s_j. T is structured as a priority queue where primary copies get higher priority than redundant copies. The primary copies must be migrated; otherwise, the schedule is infeasible. Therefore, they have the priority to be assigned first when the matching algorithm selects a candidate for the assignment.

The primary copies with only one edge in the graph should be assigned first, since they do not have another choice of free slot. Having such initial assignment, we then apply the matching algorithm for the rest of the EU instances to be migrated. In the resulted schedule after the match algorithm is applied, some primary copies might not be migrated successfully. Since the primary copies have to be scheduled, we should sacrifice some redundant copies in order to put such primary copies in. For the primary copies that can not be migrated successfully by the matching algorithm described above, we look for the slots which can accommodate them and are occupied by redundant copies. There are several strategies for selecting the candidate to be sacrificed, such as best sacrifice and first sacrifice. Best sacrifice strategy sacrifices the smallest slot among the possible slots, while first sacrifice selects the first slot it finds. Finally, we get the migration schedule.

The complexity for constructing bipartite graph and the matching algorithm is the same as above, which is $O(|T|^3 + |T||S|)$. Best sacrifice has the complexity of $O(|S|)$, while first sacrifice has constant complexity. Overall, the migration algorithm has the polynomial complexity $O(|T|^3 + |T||S|)$.

[3] The EU instances mentioned in this section includes rollback or replicated copies of EU instances which are originally scheduled on the migrated processor

Input: NFT schedule A_{NFT}.

Let T be the set of all EU instances in A_{NFT};
$A_{FT} = A_{NFT}$;
For each EU instance $x_i \in T$
    Compute EST_i and LFT_i;
For each EU instance $x_i \in T$
    Apply the part I of *replication test*;
    Let the processor j be the processor where the primary copy
    of x_i is assigned;
    If a free slot s on processor j is within
        the range of the scheduling window
        $A_{FT} = A_{FT} \cup \{(x_i^{roll}, s)\}$;
        $T = T - \{x_i\}$;
    else
        For each processor, other than j
            Find two free slots which are within the range of
            the scheduling window or one which can accommodate
            both copies;
            Update A_{FT} accordingly;
Repeat
    Let S be the set of free slots in A_{FT};
    Construct a bipartite graph $G(V, E)$;
    $V = T \cup S$;
    $E = \phi$;
    For each EU instance $x_i \in T$
    For each free slot $s_j \in S$
        if the replicated copy of x_i can be scheduled in s_j
            $E = E \cup \{(x_i, s_j)\}$;
    $\tilde{T} = \phi$;
    For each EU instance $x_i \in T$
        if x_i has only one edge (x_i, s) in E
            $A_{FT} = A_{FT} \cup \{(x_i^{repl}, s)\}$;
            $\tilde{T} = \tilde{T} \cup \{x_i\}$;
    Apply the matching algorithm;
    Let $\tilde{T}$ be the set of EU instances whose replicated
        copies are just assigned;
    $T = T - \tilde{T}$;
Until $\tilde{T} = \phi$ or $T = \phi$;

Output: FT schedule A_{FT}.

Figure 5.4: Pseudo code for the proposed method.

Chapter 6

Reliability Models

In the previous chapter, we propose a fault-tolerant scheduling algorithm which adaptively includes redundant copies into the schedule to achieve the fault-tolerance goals. The static scheduling methods allow us to pre-analyze task execution behavior and to estimate system reliability. In order to evaluate the reliability for systems using the proposed scheduling and to compare the performance of different scheduling approaches, we estimate the reliability through Markov chain model.

In this chapter, we propose an analytic model for a simple fault model that assumes only transient faults. The model is extended to include permanent faults. We validate the analytic models via simulation and the experimental results show that the estimated reliability using our analytic models is within a very small range of error.

6.1 Analytic Model for Transient Faults

In this section, we present a Markov chain model, shown in Figure 6.1, to formalize systems which consider only transient faults. Initially, the system is in the state N (the normal state), if there is no failure. When a transient failure occurs, the system jumps to the state R (the recovery state). It either tries to adjust the schedule to accommodate the rollback of the EU instance that experiences the fault, or takes the result from the replicated copy of the EU instance experiencing the fault. If the system can get the result either from the rollback or replicated copy of the EU instance and can meet all the deadlines, it goes back to the state N; otherwise, it enters the state F (the failure state). In the state F, the system goes back to the state N when a new instance of the failed task is regenerated.

A transient failure is said to be *effective* if it hits the primary copy of an EU instance. Let λ_0 be the total transient failure rate of the system and λ be the effective transient failure rate. We define q as primary EU occupation ratio, which is the ratio of the total execution time of all primary copies to the total available processor time. In other words, we can think of q as the probability that a transient failure is effective. Then, $\lambda = q\lambda_0$. Let μ be the repair rate and c be the coverage factor, denoting the conditional probability that the system recovers, given that a fault has occurred. We called $(1-c)\mu$ un-reschedulability rate

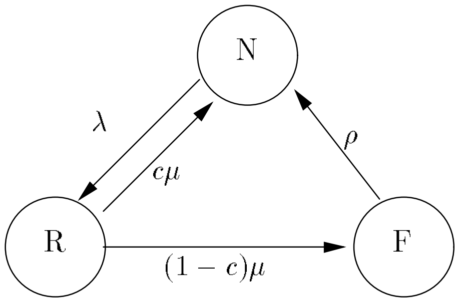


Figure 6.1: Markov chain model for transient faults.

and $c\mu$ reschedulability rate. Let ρ be the regenerating rate. Assume that the interarrival time of two failures is exponential distribution; the repair time is exponentially distributed; and the time between two regenerating tasks also has an exponential distribution. The following sections will describe how to compute system reliability and how to estimate the repair rate μ, coverage factor c, and regenerating rate ρ.

6.1.1 Reliability Analysis

Let $P_s(t)$ be the probability that the system is in state s at time t, for $s =$ N, R, F. Initial state is state N, so that

$$P_N(0) = 1, \qquad P_R(0) = P_F(0) = 0.$$

First, we compute the steady state probabilities for the system [Tri82]. We can get the steady state probabilities:

$$\begin{aligned} P_N &= \frac{\mu\rho}{\mu\rho + \lambda\rho + (1-c)\mu\lambda}, \\ P_R &= \frac{\lambda\rho}{\mu\rho + \lambda\rho + (1-c)\mu\lambda}, \\ P_F &= \frac{(1-c)\mu\lambda}{\mu\rho + \lambda\rho + (1-c)\mu\lambda}. \end{aligned}$$

To compute the mean time to failure (MTTF), we set state F as an absorbing state. The differential equations follow:

$$\begin{aligned} \frac{dP_N}{dt} &= c\mu P_R - \lambda P_N, \\ \frac{dP_R}{dt} &= \lambda P_N - \mu P_R, \\ \frac{dP_F}{dt} &= (1-c)\mu P_R. \end{aligned}$$

Using Laplace transforms, the above equations reduce to:

$$\begin{aligned} S\bar{P_N} - 1 &= c\mu\bar{P_R} - \lambda\bar{P_N}, \\ S\bar{P_R} &= \lambda\bar{P_N} - \mu\bar{P_R}, \\ S\bar{P_F} &= (1-c)\mu\bar{P_R}. \end{aligned}$$

Solving this system of linear equations, we can get:

$$\bar{P_F} = \frac{(1-c)\mu\lambda}{S} * \frac{1}{\alpha_1 - \alpha_2}(\frac{1}{S+\alpha_2} - \frac{1}{S+\alpha_1}),$$

where

$$\alpha_1,\ \alpha_2 = \frac{(\mu+\lambda) \pm \sqrt{\mu^2 + 2(1-2c)\mu\lambda + \alpha^2}}{2}.$$

After an inverse Laplace transform, we can get P_F, the probability that the system is in failure mode at time $t \geq 0$. Let Y be the time to failure of the system. The reliability of the system is:

$$R(t) = 1 - P_F(t).$$

The Laplace transform of the failure density,

$$f_Y(t) = -\frac{dR(t)}{dt} = \frac{dP_F(t)}{dt},$$

is expressed as:

$$\begin{aligned} L_Y(S) &= \bar{f_Y}(S) \\ &= \frac{(1-c)\mu\lambda}{\alpha_1 - \alpha_2}(\frac{1}{S+\alpha_2} - \frac{1}{S+\alpha_1}). \end{aligned}$$

Inverting the transform in the above expression, we obtain:

$$f_Y(t) = \frac{(1-c)\mu\lambda}{\alpha_1 - \alpha_2}(e^{-\alpha_2 t} - e^{-\alpha_1 t}).$$

Hence, the MTTF of the system is given by:

$$\begin{aligned} E[Y] &= \int_0^\infty y f_Y(y) dy \\ &= \frac{1}{1-c}(\frac{1}{\mu} + \frac{1}{\lambda}). \end{aligned} \tag{6.1}$$

6.1.2 Estimation of System Parameters

To compute the system parameters: repair rate μ, coverage factor c, and regenerating rate ρ, we need the following assumptions and notations. Let n be the number of the EU instances in a *frame*, x_1, x_2, ..., x_{n_r} be the EU instances with or without rollback copies, but no replicated copies, in the schedule and $x_{n_r+1}, x_{n_r+2}, \ldots, x_n$ be the EU instances with replicated copy. Let E_i be the random variable representing the execution time of the EU instance x_i, for $i = 1, 2, \ldots, x_n$; E'_i be the random variable representing the time that the system needs to report the unrecovered error condition because x_i fails, for $i = 1, 2, \ldots, n_r$; R_i be the random variable representing the processing time that the system needs to get the result from the replicated copy of x_i, for $i = n_r + 1, n_r + 2, \ldots, n$. Assume that E_i has exponential distribution with the mean e_i, for $i = 1, 2, \ldots, n$; E'_i has exponential distribution with the mean e'_i, for $i = 1, 2, \ldots, n_r$; and R_i has exponential distribution with the mean r_i for $i = n_r + 1, n_r + 2, \ldots, n$. The purpose of making these assumptions is to simplify our approximation procedure. Although such assumptions might not be realistic for RTS, we will see from our simulation results that the Markov chain model performs well on estimating system reliability.

The system parameters that need to be estimated relate to the probability that the system recovers if a given EU instance fails. The system recovers from a transient fault implies that the failed EU has a redundant copy in the schedule such that the EU instance can be recovered by replication or rollback. We define p_i as the probability that x_i has a redundant copy in the schedule. x_i has a redundant copy implies that the redundant copy of x_i can be scheduled by the routine *rescheduling* and all deadlines are guaranteed. Since our system employs static scheduling, we can determine in advance if an EU instance has a redundant copy. For those with redundant copies in the schedule, they can cope with transient faults by rollback or replication. Therefore, p_i equals to 1 for such EU instances and 0 for the instances with only primary copies in the FT schedule.

Let w_i be the probability that x_i experiences the fault given that an effective fault has occurred. We assume that a transient fault can happen at any time. Hence, w_i can be expressed as the ratio of the execution time of x_i to the total execution time of all EU instances. The coverage factor can be expressed as the probability that an EU can be recovered from a transient fault when the system encounters a fault.

$$c = \sum_{i=1}^{N} w_i p_i. \tag{6.2}$$

The repair rate μ can be expressed as the average jumping-out rate. For an EU instance x_i with single copy, its jumping-out rate from the state R is $(p_i \frac{1}{e_i} + (1 - p_i)\frac{1}{e'_i})$; for an EU instance x_i with replicated copy, its jumping-out rate from the state R (always to the state N) is $\frac{1}{r_i}$. Hence, the computation for μ is followed:

$$\mu = \frac{1}{N}\left(\sum_{i=1}^{n_r}\left(p_i \frac{1}{e_i} + (1 - p_i)\frac{1}{e'_i}\right) + \sum_{i=n_r+1}^{N} \frac{1}{r_i}\right). \tag{6.3}$$

To estimate the regenerating rate, we need to know the probability that an EU instance fails and the system enters the state F. Fortunately, we can obtain this probability from p_i. $1 - p_i$ is the probability that the EU instance x_i only has a single copy (primary copy) in the schedule, that is, the probability that x_i enters the state F if x_i has experienced a transient fault. Since tasks are periodic, a new instance of a task is generated at the beginning of its period. Let t_i be the period of the task that contains x_i. we express the regenerating rate as the weighted probability that an EU instance enters the state F multiplied by the frequency, that is,

$$\rho = \sum_{i=1}^{N} \frac{1 - p_i}{\sum_{j=1}^{N}(1 - p_j)} \frac{1}{t_i}. \tag{6.4}$$

We have devised the methods of computing the system parameters (c, μ and ρ), p_i, and system reliability. Once we get the values of the system parameters according to the formula derived above, we can obtain the system reliability. By applying our proposed AR approach, a hybrid method of temporal and spatial redundancy, we can get a very high degree of coverage factor and hence a long MTTF. Besides, the system has higher resource utilization, because it has more free resources can be used for aperiodic tasks.

6.1.3 Simulation

In this section, we validate the proposed model and demonstrate the model is a powerful tool in estimating the reliability. To validate the analytic model in estimating the system reliability, we compare the results from the simulations and the model and we use *difference ratio*, which is defined as the percentage of the difference between the MTTF and the simulated average time to failure (ATTF) over the ATTF, as our performance measurement.

A simulation program was written to capture the behavior of the transient faults and the characteristics of the real-time system model described in Section 5.1. A number of experiments were conducted to examine the performance of the proposed analytic model over various system workloads and failure rates. For each experiment, we generate 500 simulated systems; for each simulated system, 1000 system failures are produced to obtain the ATTF of the system. The final results were evaluated by averaging the difference ratios, between ATTF and MTTF, obtained from the 500 systems.

Simulation Model

The simulation program for the simulated RTS consists of four components: task generation, fault injection, scheduler, and resource management, as shown in Figure 6.2. The task generation is responsible for generating a set of periodic tasks associated with various timing constraints and EUGs for the tasks; the fault injection component generates transient faults and injects to the system according to the probability distribution of the faults. The scheduler takes charge with allocating and scheduling the tasks; it constructs static off-line FT schedule based on the proposed approach. The resource management component maintains the state of the resources and consumes the resources according to the schedule. Based on the FT schedule obtained from the scheduler, the analytic model computes the system parameters defined in Section 6.1.2 and the MTTF for the simulated system.

Based on our model, the MTTF for a system using NFT schedule should be $\frac{1}{\lambda}$, that is $\frac{1}{q\lambda_0}$. With the various values of simulation parameters described below, the average difference ratio for NFT schedules is about 2.5%. It is quite stable under various system

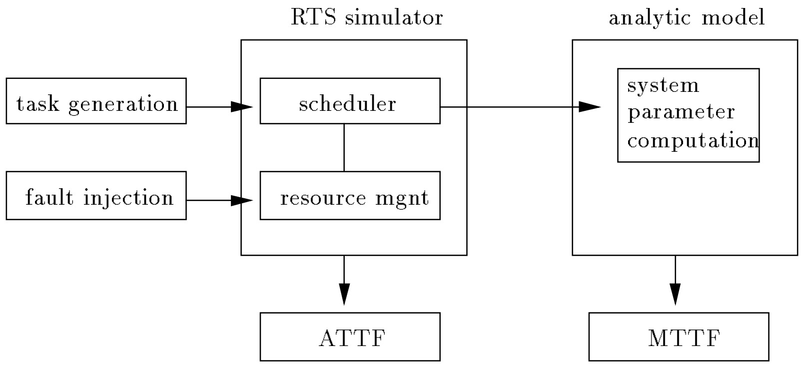


Figure 6.2: The overview structure of the simulation.

workloads and failure rates. The major interest in this section is to discuss the performance of the Markov chain model in estimating the reliability for the systems using off-line FT schedule.

The simulation mainly depends on system workload and failure rate. Our simulation contains the following parameters:

- $Ncpu$ represents the number of processors in the system.
- $Napp$ represents the number of applications in the system.
- $NEUInApp$ represents the number of EUs in an application. It is uniformly distributed over the range of 2 to $Max\_NEUInApp$ for all experiments.
- $MaxOutDegree$ represents the maximum outdegree of an EU in an EUG. It relates to generate precedence relationship. The number of immediate successors of an EU is uniformly distributed over the range of 0 to $MaxOutDegree$. If the number of EUs in an application is fixed, the larger $MaxOutDegree$ is; the higher degree of parallelism the application is. We have tried different values of $MaxOutDegree$. Since they convey the similar behavior, we only show the results with the value of 4.
- $PeriodScale$ represents the scale of a period. The period of an application is a multiple of $PeriodScale$.
- $MinExeTime$ and $MaxExeTime$ determine the range of the execution time of an EU. The execution time of an EU is uniformly distributed between $MinExeTime$ and $MaxExeTime$.
- $GetResultTime$ represents the processing time for obtaining the result from a replicated copy.
- $ErrorReportTime$ represents the processing time for reporting an unrecovered error condition.

- $TransArrivalTime$ denotes the inter-arrival time of two consecutive transient errors (TAT), which determines the transient failure rate.

Because of the interleave execution of the instances of EUs and the random number of EUs in an application, it is not easy to have a clear expression to define workload in this simulation. Every simulation parameter may affect workload. Therefore, our simulation attempts to cover most of the possible angles of changing workload.

Experiments show that the values of $GetResultTime$ and $ErrorReportTime$ do not affect much on the performance of the model. Therefore, we choose fixed values of these two parameters; both are 1 time unit.

We randomly generate EUGs according to the simulation parameters given above. We adopt the allocation and scheduling algorithm proposed by Cheng *et al.* [CHA94] to obtain NFT schedules. The algorithm is the framework of the allocator on MARUTI [SdSA94, MSA92], a hard real-time operating system developed at the University of Maryland. The maximum values of $Ncpu$ and $Napp$ and the value of $Max\_NEUInApp$ are set to 14, 4, and 20, respectively, due to the implementation limitation of the NFT schedule generator implemented by Cheng. We implement the fault-tolerant scheduling approach described in Section 5.3 to get FT schedules. Based on FT schedules and failure rate, we can compute ATTF through the simulation and MTTF from equation 6.1.

Simulation Results

We will examine the sensitivity of the Markov chain model on failure rate and system workload in this section. We have run many different settings on the experiments of changing the failure rate. Since they convey the similar behavior, we only show several settings on the experiments and the corresponding results.

We vary the parameter $TransArrivalTime$ (TAT) under different workloads. Table 6.1 shows the settings for the experiment and Figure 6.3 gives the corresponding results. It can be found that the difference ratios are very small and quite stable on various TAT. Hence, we conclude that the analytical model is insensitive to TAT.

In order to see the performance on various system workloads, we vary the workload simulation parameters, described in Section 6.1.3, one at a time. The settings are chosen to reflect a diversity of workload, from heavy to light workload. The settings for the experiment of varying the number of processors are shown in Table 6.2 and the results in Figure 6.4. As the workload decreases, that is, the number of processors increases, the chance that the rescheduling algorithm can find a free slot for a replicated or rollback copy increases. Therefore, the chance for a system with full coverage (coverage factor equals 1) increases. The difference ratio for such system is 0. When the workload decreases, the number of full coverage systems increases and hence the average difference ratio decreases.

Simulation results for varying the scale of period are given in Figure 6.5 and the setting in Table 6.3. The difference ratio goes slightly down as the scale of period increases, but the proposed analytic model still can estimate the system reliability accurately.

Table 6.1: Settings for the experiment of varying TAT.

parameter	wkd 1	wkd 2	wkd 3	wkd 4	wkd 5
$Ncpu$	6	7	8	9	10
$Napp$	2	2	3	3	3
$PeriodScale$	26	34	35	36	35
$MinExeTime$ (time units)	8	9	5	8	10
$MaxExeTime$ (time units)	14	15	12	12	14

Table 6.2: Settings for the experiment of varying $Ncpu$.

parameter	case 1	case 2	case 3
$Napp$	2	3	4
$PeriodScale$	32	32	32
$MinExeTime$ (time units)	8	8	8
$MaxExeTime$ (time units)	18	18	18
TAT (time units)	1000	1000	1000

The last experiment varies the range of the execution time. Fixing either the minimum or the maximum execution time while changing the other gives the similar results, so we only show the results with fixed minimum execution time in Figure 6.6. The settings are given in Table 6.4. The workload increases and the slack time of EU instances decreases, when the maximum execution time increases. As mentioned above, the number of full coverage system inclines when the workload increases and hence the difference ratio raises slightly. Again, the Markov chain model performs well on various ranges of the execution time.

By averaging the difference ratios obtained from our simulation, we find that our analytical model performs well with the average difference ratio of 1.2 % over various system workloads and failure rates. Hence, we conclude that the proposed Markov chain model can be used to estimate with high accuracy the reliability of the system using static scheduling.

Table 6.3: Settings for the experiment of varying $PeriodScale$.

parameter	case 1	case 2	case 3
$Ncpu$	6	8	10
$Napp$	2	3	4
$MinExeTime$ (time units)	8	8	8
$MaxExeTime$ (time units)	18	18	18
TAT (time units)	1000	1000	1000

Table 6.4: Settings for the experiment of varying the range of execution time.

parameter	case 1	case 2	case 3
$Ncpu$	5	8	10
$Napp$	2	3	4
$PeriodScale$	32	32	32
$MinExeTime$ (time units)	8	8	8
TAT (time units)	1000	1000	1000

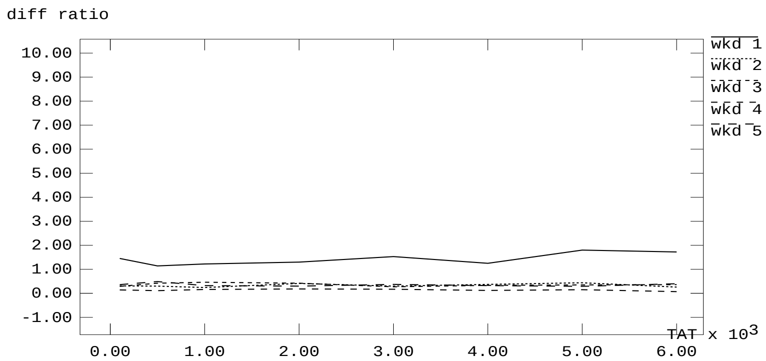


Figure 6.3: Results for varying TAT.

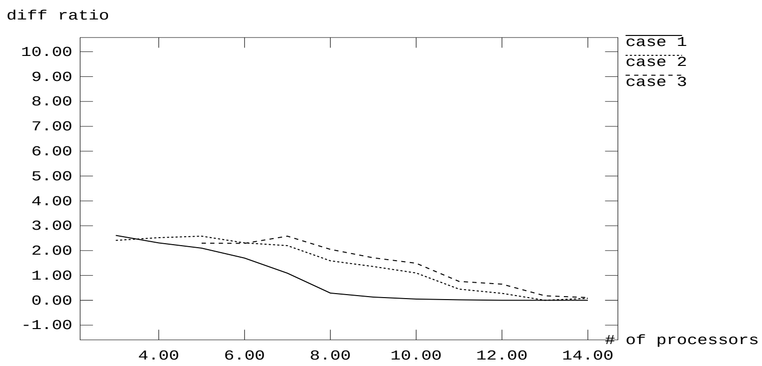


Figure 6.4: Results for varying the number of processors.

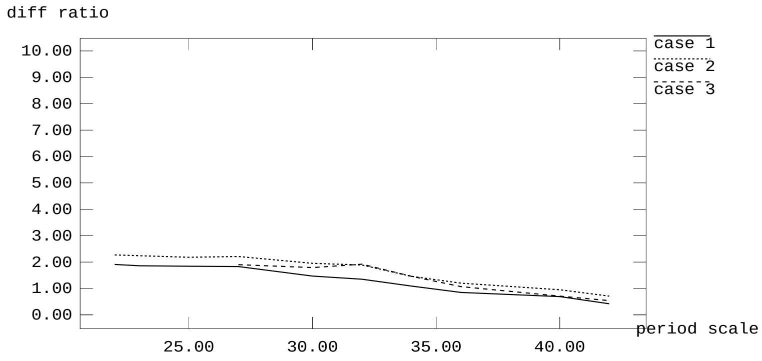


Figure 6.5: Results for varying the scale of period.

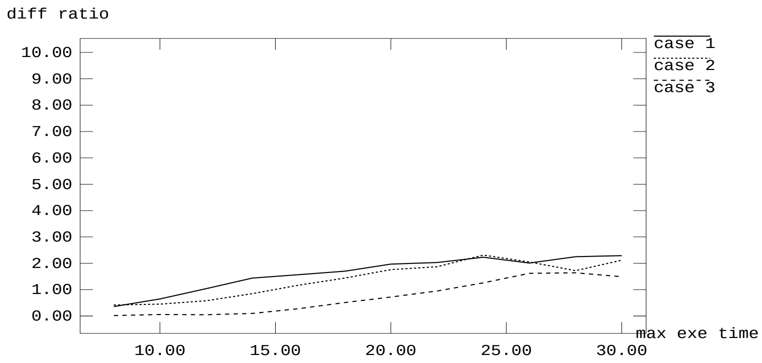


Figure 6.6: Results for varying the range of execution time.

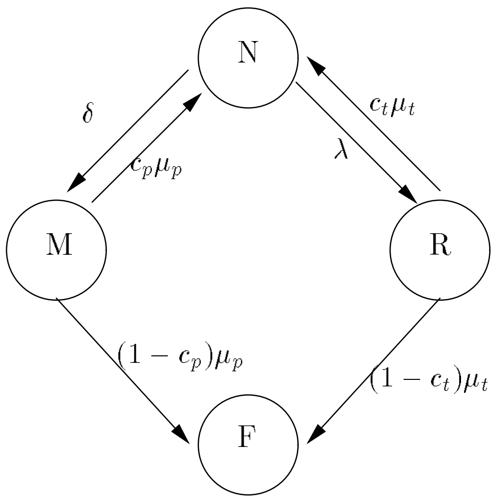


Figure 6.7: Markov chain model for permanent and transient faults.

6.2 Analytic Model for Transient and Permanent Faults

We extend the analytic model presented in Section 6.1 to permanent faults, which is shown in Figure 6.7. Initially, the system is assumed to be in the normal state (the state N). When a processor encounters a permanent fault, the system enters the migration state (the state M) and migrates the EU instances to other processors. If migration process succeeds (i.e., the EU instances on the failed processor with only one copy in the schedule are migrated), the system goes back to the normal state. Otherwise, it enters the failure state (the state F). When a transient fault happened during the execution of an EU, the system enters the recovery state (the state R). If the schedule has a redundant copy of the failed EU instance, either rollback or replication copy, the system recovers from the fault and goes back to the normal state. Otherwise, it enters the failure state.

λ is the transient failure rate; δ is the permanent failure rate. Let μ be the recovery rate and c be the coverage factor, denoting the conditional probability that the system recovers, given that a fault has occurred. The subscript of the coverage factor c and the recovery rate μ denotes the fault type, i.e., c_p represents the coverage factor for permanent faults and c_t represents the coverage factor for transient faults. Same rule applies to the recovery rate μ. We call $c_t\mu_t$ reschedulability rate, $(1-c_t)\mu_t$ un-reschedulability rate, $c_p\mu_p$ migration rate, and $(1-c_p)\mu_p$ un-repair rate. Assume that the inter-arrival time of two consecutive permanent or two consecutive transient failures is exponential distribution; the recovery time is exponentially distributed. The following section describes the system reliability based on our assumptions.

6.2.1 Reliability Analysis

Let $P_s(t)$ be the probability that the system is in the state s at time t, for $s =$ N, R, M, and F. Initial state is the state N, so that

$$P_N(0) = 1, \qquad P_R(0) = P_M(0) = P_F(0) = 0.$$

To compute the MTTF, the differential equations follow:

$$\begin{aligned}
\frac{dP_N}{dt} &= c_t\mu_t P_R + c_p\mu_p P_M - (\lambda+\delta)P_N, \\
\frac{dP_R}{dt} &= \lambda P_N - \mu_t P_R, \\
\frac{dP_M}{dt} &= \delta P_N - \mu_p P_M, \\
\frac{dP_F}{dt} &= (1-c_t)\mu_t P_R + (1-c_p)\mu_p P_M.
\end{aligned}$$

Using Laplace transforms, the above equations reduce to

$$\begin{aligned}
S\bar{P_N} - 1 &= c_t\mu_t\bar{P_R} + c_p\mu_p\bar{P_M} - (\lambda+\delta)\bar{P_N}, \\
S\bar{P_R} &= \lambda\bar{P_N} - \mu_t\bar{P_R}, \\
S\bar{P_M} &= \delta\bar{P_N} - \mu_p\bar{P_M}, \\
S\bar{P_F} &= (1-c_t)\mu_t\bar{P_R} + (1-c_p)\mu_p\bar{P_M}.
\end{aligned}$$

Solving this system of linear equations, we can get

$$\bar{P_F} = \frac{1}{S}\frac{(1-c_t)\lambda\mu_t(S+\mu_p) + (1-c_p)\delta\mu_p(S+\mu_t)}{(S+\lambda+\delta)(S+\mu_t)(S+\mu_p) - c_t\lambda\mu_t(S+\mu_p) - c_p\delta\mu_p(S+\mu_t)}.$$

After an inverse Laplace transform, we can get P_F, the probability that the system is in the failure state at time $t \geq 0$. Since we are interested in the average behavior of the system, instead of instance behavior, we do not necessarily have to know exactly what P_F is. Let Y be the time to failure of the system. The reliability of the system is defined as

$$R(t) = 1 - P_F(t).$$

The Laplace transform of the failure density,

$$f_Y(t) = -\frac{dR(t)}{dt} = \frac{dP_F(t)}{dt},$$

is expressed as

$$\begin{aligned}
L_Y(S) &= \bar{f_Y}(S) \\
&= S\bar{P_F}(S) - P_F(0^-) \\
&= \frac{(1-c_t)\lambda\mu_t(S+\mu_p) + (1-c_p)\delta\mu_p(S+\mu_t)}{(S+\lambda+\delta)(S+\mu_t)(S+\mu_p) - c_t\lambda\mu_t(S+\mu_p) - c_p\delta\mu_p(S+\mu_t)}.
\end{aligned}$$

Hence, the MTTF of the system is given by

$$
\begin{aligned}
E[Y] &= -\frac{dL_Y}{dS}\bigg|_{S=0} \\
&= \frac{\mu_t\mu_p + \lambda\mu_p + \delta\mu_t}{(1-c_t)\lambda\mu_t\mu_p + (1-c_p)\delta\mu_t\mu_p}.
\end{aligned}
\tag{6.5}
$$

6.2.2 Estimation of System Parameters

All the system parameters in the proposed analytic model relate to the conditional probability that the system recovers if an EU instance or a processor fails. In the following, we present the strategy for how to compute these probabilities and then devise the formulae for the system parameters.

Let P_j^p be the probability that the system has a feasible schedule given that processor j has encountered a permanent fault, where a feasible schedule is defined as a schedule which can schedule and complete all EUs before their deadlines expire. Since our system uses static schedule, we can prepare one migration schedule for each processor in advance and hereafter P_j^p can be determined. We call processor j is *migratable* if the migration schedule for processor j is feasible.

Let P_i^t denote the probability that EU instance x_i has a redundant copy, either rollback or replicated copy, in the schedule given that a transient fault has happened during the execution of x_i. As mentioned in Section 5.1, a failed processor will be replaced by a processor functioning correctly within one *frame* period of time. To compute the conditional probability P_i^t, we should consider if there is a permanent fault happening during the last *frame*. Let t be the current time; t_p be the time that the last permanent fault occurs; and L be the length of a *frame*. If the last permanent fault happens during the last *frame*, i.e., $(t - t_p) \leq L$, the system uses the migration schedule for the failed processor. In the other case that $(t - t_p) > L$, the failed processor has been replaced and the system uses the original FT schedule to schedule EU instances. The time that the last permanent fault occurs determines which schedule, FT schedule or migration schedule, is used and therefore it determines the probability that an EU instance has a redundant copy.

If a permanent fault occurs on a non-migratable processor, i.e., it is an un-recoverable permanent fault, the system fails, enters the failure state, and never goes back to the normal state. No transient fault will occur in such failed system after an un-recoverable permanent fault is detected. Hence, the permanent faults considered here should be recoverable, that is, they happen on migratable processors. The probability P_i^t is constituted by the following two probabilities: the probability that the system recovers given that x_i experiences a transient fault and no permanent fault happened during the last *frame* and the probability that the system recovers given that x_i experiences a transient fault and a permanent fault happened during the last *frame*.

We need the following notations to decompose the conditional probability P_i^t. Let A_i represent the event that the EU instance x_i recovers from a transient fault, H_i be the event that x_i experiences a transient fault, U_j be the event that a recoverable permanent fault happened on processor j during the last *frame*, U be the event that a recoverable permanent fault happened in the system during the last *frame*, and V be the event that no permanent fault happened in the last *frame*. $P(U)$ denotes the probability that a recoverable permanent fault happened in the last *frame*, similarly, we can define the probability $P(V)$ and the conditional probabilities $P(A_i|H_i\&V)$, $P(A_i|H_i\&U)$, and $P(A_i|H_i\&U_j)$. As mentioned above that P_i^t is built up by $P(A_i|H_i\&U)$ and $P(A_i|H_i\&V)$, by the definition of conditional probability, P_i^t can be expressed as follows:

$$\begin{aligned} P_i^t &= P(A_i|H_i) \\ &= P(A_i|H_i\&U)\frac{P(U)}{P(U)+P(V)} + P(A_i|H_i\&V)\frac{P(V)}{P(U)+P(V)}. \end{aligned}$$

Let m be the number of processors in the system, m_m be the number of the migratable processors, and S_m be the set of the migratable processors. Since we assume that the inter-arrival time between two permanent faults is exponentially distributed and that each processor has equal chance to encounter a permanent fault, $P(U)$ and $P(V)$ can be computed as follows:

$$\begin{aligned} P(U) &= (1-e^{-\delta L})\frac{m_m}{m}, \\ P(V) &= e^{-\delta L}. \end{aligned}$$

$P(A_i|H_i\&U)$ can be obtained by summing up the probabilities that each migratable processor contributes. Namely,

$$P(A_i|H_i\&U) = \frac{1}{m_m}\sum_{j\in S_m} P(A_i|H_i\&U_j).$$

Like P_j^p, $P(A_i|H_i\&V)$ can be obtained from the FT schedule and $P(A_i|H_i\&U_j)$ can be get according to the migration schedule.

To compute the coverage factor for transient fault c_t, we assume that a transient fault can happen at any time. Therefore, a long EU instance has a greater chance to experience a transient fault than a short one. The coverage factor c_t is computed as the sum of the weighted probability that an EU instance recovers from a transient fault. Let w_i be the probability that EU instance x_i experiences the fault given that a transient fault has occurred. According to our assumption, it can be computed as the ratio of the execution time of x_i to the total execution time of all EU instances in a frame. c_t be expressed as

$$c_t = \sum_{i=1}^{n} w_i P_i^t. \tag{6.6}$$

The coverage factor for permanent fault c_p is simpler. A permanent fault has equal probability to happen on any processor. c_p averages the probability that the system recovers given that processor j has encountered a permanent fault, for $j = 1, 2, \ldots, m$, that is,

$$c_p = \sum_{j=1}^{m} P_j^p. \tag{6.7}$$

To compute recovery rates, μ_t and μ_p, we need the following assumptions and notations. Let n be the number of EU instances in a *frame*, x_1, x_2, ..., x_{n_r} be the EU instances with or without rollback copies, but no replicated copies, in the schedule and $x_{n_r+1}, x_{n_r+2}, \ldots, x_n$ be the EU instances which have replicated copies in the schedule. Let E_i be the random variable representing the execution time of the EU instance x_i, for $i = 1, 2, \ldots, n$, $O_{error-report}$ be the random variable representing the overhead that the system needs to report an unrecoverable error condition, $O_{get-result}$ be the random variable representing the processing time that the system needs to get the result from a replicated copy, and $O_{migration}$ be the random variable representing the overhead that the system needs to migrate the EU instances on a failed processor to other processors. Assume that E_i has an exponential distribution with the mean e_i, for $i = 1, 2, \ldots, n$; $O_{error-report}$, $O_{get-result}$, and $O_{migration}$ are exponentially distributed with the mean $o_{error-report}$, $o_{get-result}$, and $o_{migration}$, respectively. The purpose of making these assumptions is to simplify our approximation procedure. Although such assumptions might not be realistic to RTS, we will see from our simulation results that the analytic model performs well on estimating system reliability.

The system enters the recovery state due to a transient fault. If the fault occurs during the execution of an EU instance without redundant copy, the system goes to the failure state with the rate of $\frac{1}{o_{error-report}}$. Similarly, if an EU instance x_i with a rollback copy encounters a fault, the system goes back to the normal state with the rate of $\frac{1}{e_i}$. If the failed EU instance has a replicated copy, the system enters the normal state with the rate of $\frac{1}{o_{get-result}}$. The recovery rate μ_t can be expressed as the averaged jumping-out rate. We already have the transition rates from the recovery state to other states for each EU instance. μ_t can be calculated by averaging the jumping-out rates for all EU instances. The formula for μ_t is

$$\begin{aligned} \mu_t &= \frac{1}{n}(\sum_{i=1}^{n_r}(P_i^t\frac{1}{e_i} + (1 - P_i^t)\frac{1}{o_{error-report}}) + \\ &\quad \sum_{i=n_r+1}^{n}(P_i^t\frac{1}{o_{get-result}} + (1 - P_i^t)\frac{1}{o_{error-report}}). \end{aligned} \tag{6.8}$$

The recovery rate for permanent fault c_p can be computed similarly. If the failed processor is migratable, the system goes back to the normal state with the rate of $\frac{1}{o_{migration}}$. Otherwise, it enters the failure state with the rate of $\frac{1}{o_{error-report}}$. Hence, it can be computed by the following expression:

$$\mu_p = \frac{1}{m}(\sum_{j=1}^{m}(P_j^p\frac{1}{o_{migration}} + (1 - P_j^p)\frac{1}{o_{error-report}})). \tag{6.9}$$

We have presented the method of computing the system parameters used for the analytic model (c_t, c_p, μ_t, and μ_p) and the reliability. Once we get the values of the system parameters, we can obtain the reliability. By applying our proposed fault-tolerant scheduling approach, a hybrid method of temporal and spatial redundancy, we can get a very high degree of coverage factor and hence a long MTTF. Besides, the system has higher resource utilization, because it has more free resources can be used for aperiodic tasks.

6.2.3 Simulation

The simulation design mainly is an extension of the previous simulation described in Section 6.1.3. Besides the simulation parameters described in that section, two extra simulation parameters are needed for the simulation:

- $PermArrivalTime$ denotes the inter-arrival time of two consecutive permanent faults (PAT), which determines the permanent failure rate, δ.
- $MigrationTime$ represents the overhead for migrating EU instances of the failed processor.

The task generation component of the simulation program basically is the same as the previous simulation. The fault injection component generates transient and permanent faults independently and simultaneously once the simulated RTS starts. Besides FT schedule, the scheduler needs to construct migration schedule for each processor by applying the migration algorithm presented in Section 5.4. For each simulated RTS, 1000 system failures are generated to compute ATTF for a given system.

Analytic model simulation computes system parameters based on the FT schedule and the migration schedules obtained from the scheduler. MTTF for a given system can be determined by the values of the system parameters.

Simulation Results

This section discusses the experimental results for the validation of the proposed model. For the sets of the experiments on various workloads, we change one simulation parameter relating to workload at a time to observe the performance of the proposed model over a wide range of workloads.

Table 6.5 gives the settings for the experiments that varies the number of processors and Figure 6.8 presents the corresponding results. The figure shows that the difference ratio drops as the number of the processors increases. The number of the systems with full coverage (c_t and c_p are 1) increases when the workload decreases, since the number of free slots in the schedule increases and the probability that an EU instance has a redundant copy increases. The estimated reliability of full coveraged system is exactly the same as the simulated reliability which is infinity; such case has zero difference ratio. Therefore, when the workload decreases, the number of the full coveraged systems increases and the difference ratio decreases.

The settings for the experiments that varies the range of execution time are shown in Table 6.6 and the corresponding results in Figure 6.9. The same reasoning described above applies to this set of experiments, since the workload decreases when the maximum execution time becomes small.

The settings and the results for the experiments on varying the scale of period are shown in Table 6.7 and Figure 6.10, respectively. Because of the interleaving of EU instances in the schedule, small period does not necessarily imply heavy workload. For each case of

Table 6.5: Settings for the experiment of varying $Ncpu$ for extended model.

parameter	case 1	case 2	case 3
$Napp$	2	3	4
$PeriodScale$	32	32	32
$MinExeTime$ (time units)	8	8	8
$MaxExeTime$ (time units)	18	18	18
PAT (time units)	3000	3000	3000
TAT (time units)	1000	1000	1000

Table 6.6: Settings for the experiment of varying the range of execution time for extended model.

parameter	case 1	case 2	case 3
$Ncpu$	5	8	10
$Napp$	2	3	4
$PeriodScale$	32	34	36
$MinExeTime$ (time units)	8	8	8
PAT (time units)	2500	2500	2500
TAT (time units)	1000	1000	1000

different scale of period, approximately 10% to 20% of generated systems has full coverage. The results show that the analytic model performs quite stable on various scales of period. In general, the model performs very well at various workloads.

We conduct the experiments that change the inter-arrival time between two transient failures (TAT) or two permanent failures (PAT). The settings are presented in Tables 6.8 and 6.9 and the results are shown in Figure 6.12 and 6.11, respectively. The results show that the analytic model performs stable at various failure rates. Hence, we conclude that the proposed analytic model can estimate the reliability accurately at various system workload and failure rates.

Table 6.7: Settings for the experiment of varying $PeriodScale$ for extended model.

parameter	case 1	case 2	case 3
$Ncpu$	6	8	10
$Napp$	2	3	4
$MinExeTime$ (time units)	8	8	8
$MaxExeTime$ (time units)	18	18	18
PAT (time units)	2500	2500	2500
TAT (time units)	1000	1000	1000

Table 6.8: Settings for the experiment of varying TAT for extended model.

parameter	wkd 1	wkd 2	wkd 3	wkd 4	wkd 5
$Ncpu$	6	7	8	9	10
$Napp$	2	2	3	3	3
$PeriodScale$	26	34	35	36	35
$MinExeTime$ (time units)	8	9	5	8	10
$MaxExeTime$ (time units)	14	15	12	12	14
PAT (time units)	3000	3000	3000	3000	3000

Table 6.9: Settings for the experiment of varying PAT for extended model.

parameter	wkd 1	wkd 2	wkd 3	wkd 4	wkd 5
$Ncpu$	6	7	8	9	10
$Napp$	2	2	3	3	3
$PeriodScale$	26	34	35	36	35
$MinExeTime$ (time units)	8	9	5	8	10
$MaxExeTime$ (time units)	14	15	12	12	14
TAT (time units)	2000	2000	2000	2000	2000

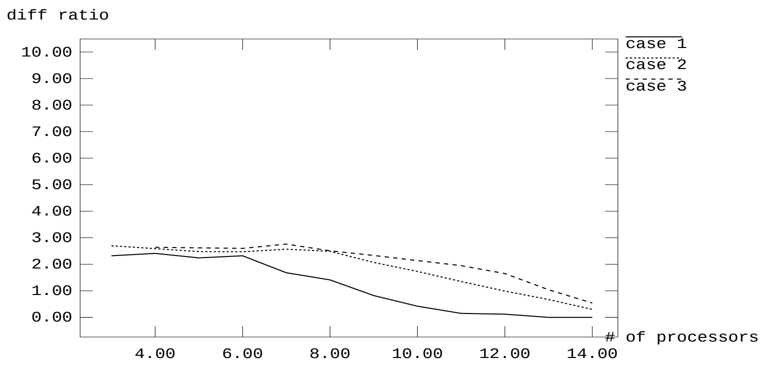


Figure 6.8: Results on evaluation of extened model for varying the number of processors.

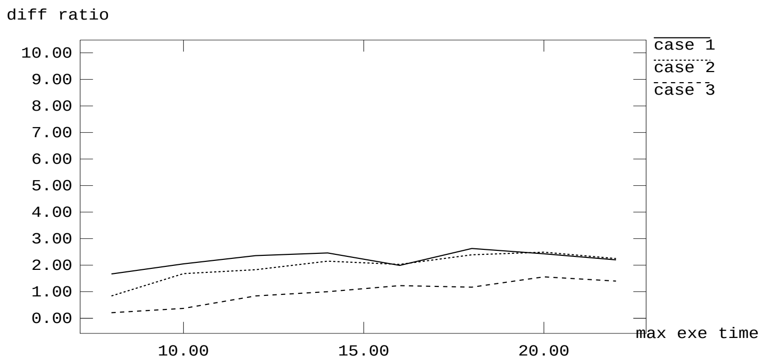


Figure 6.9: Results on evaluation of extened model for varying the range of execution time.

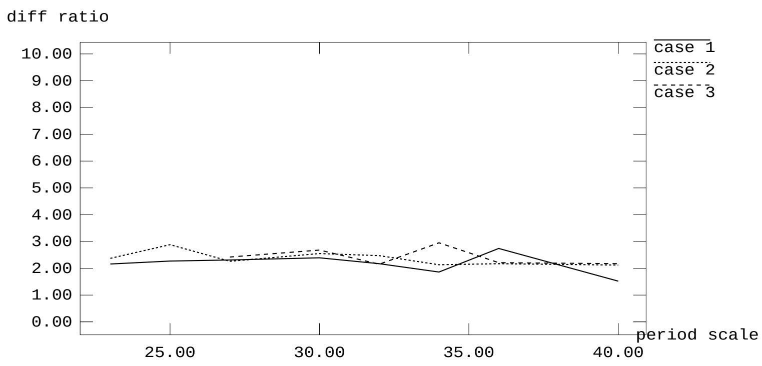


Figure 6.10: Results on evaluation of extened model for varying the scale of period.

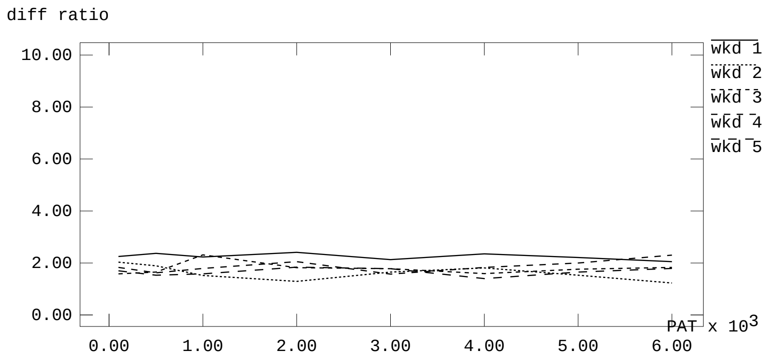


Figure 6.11: Results on evaluation of extened model for varying PAT.

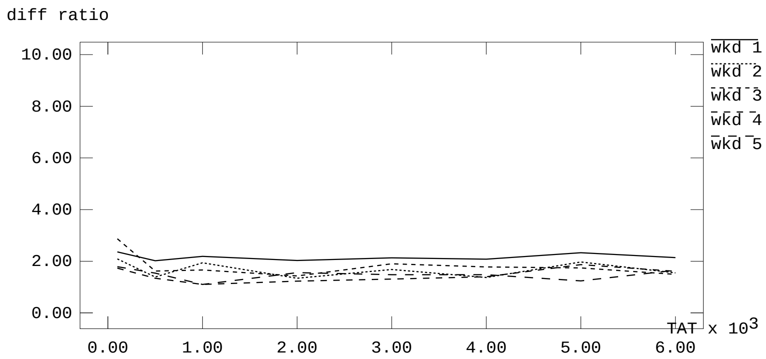


Figure 6.12: Results on evaluation of extened model for varying TAT.

Chapter 7

Performance Evaluation

In this chapter, we present two conventional fault-tolerant scheduling approaches: replication and rollback schemes and compare the performance of the proposed adaptive redundancy (AR) scheduling approach with the basic fault-tolerant scheduling schemes. The simulation results show that our proposed fault-tolerant scheduling, AR method, outperforms the replication and rollback schemes under various system workloads and failure rates.

7.1 Basic Fault-Tolerant Scheduling

It has been proven that finding a feasible schedule for a multiprocessor system is an NP-hard problem [Bur90, GJ75] and that adding a task into a schedule is also an NP-hard problem [GJ79]. Thus, given a permutation of all the EU instances in the schedule, the rescheduling algorithm for scheduling a redundant copy of an EU instance is still an NP-hard problem. Exhaustive search on all the possible permutations wastes tremendous time and resources. Considering the tradeoff between finding an optimal fault-tolerant schedule and the computational time for the rescheduling algorithm, we adopt polynomial-time rescheduling algorithms and propose branch and bound algorithms to eliminate unnecessary search for both schemes to be presented.

7.1.1 Replication Scheme

We propose a branch and bound algorithm to find a suboptimal solution for replicating EU instances. The search space is all the possible permutations of the EU instances in a *frame*. We represent it as a search tree with $n!$ leaf nodes. The root of the search tree, the node in level 0, is the original NFT schedule; an intermediate node is a partial FT schedule; a son of a node is an immediate extension of the partial FT schedule corresponding to the node. A leaf node, specifying a complete sequence of all the EU instances, is a full FT schedule. The goal of the replication algorithm is to find a best FT schedule among all the possible FT schedules obtained from the leaf nodes.

We use the heuristics to constraint the search space. The objective function of the branch and bound algorithm is constituted by the heuristics. we mentioned, in Section 6.1.2, that larger coverage factor has longer MTTF. According to the equation 6.2 for computing the coverage factor, a long EU instance has more contribution on the coverage factor than a short one. Therefore, the objective function for a full FT schedule is defined as the total execution time of the instances with replicated copies in the schedule. Hence we can say that the goal of the algorithm is to search for an FT schedule which has the maximum value of the objective function among those possible FT schedules.

In an intermediate node, we need an estimation function for the corresponding partial FT schedule in order to decide if the node is worth expending. The estimated objection function for an intermediate node should represent the best value of the objective function that its descendants can get. It is defined as the total execution time of the instances whose replicated copies are in the partial FT schedule corresponding to the intermediate node *plus* the execution times of all instances which are not in the permutation sequence of the node. Let $< x_{i_1}, x_{i_2}, \ldots, x_{i_k} >$ be the sequence associated with a node in level k. The estimated objective function of the node can be computed as follows:

$$\begin{aligned} \hat{\mathcal{F}}(< x_{i_1}, x_{i_2}, \ldots, x_{i_k} >) &= \sum_{l=1}^{k} \theta(x_{i_l}) * exe\_time(x_{i_l}) + \\ & \qquad \sum_{\forall l \notin \{i_1, i_2, \ldots, i_k\}} exe\_time(x_l), \end{aligned} \tag{7.1}$$

where $exe\_time(x_i)$ gives the execution time of EU instance x_i; the function θ used for replication scheme is defined as:

$$\theta^{repl}(x_i) = \begin{cases} 1 & \text{if the replicated copy of } x_i \text{ is in the partial schedule,} \\ 0 & \text{otherwise.} \end{cases}$$

The estimated objective function also defines the objective function for the leaf nodes, where the second term of the equation 7.1 is omitted. It is clear that the value of the estimated objective function for a node is greater than or equal to the value of the objective function for any one of its descendants.

During the search, we keep track of currently the best (largest) objective function value $BestVal$ and the corresponding FT schedule. At each intermediate node, we compute the value of the estimated objective function defined above. If the value of the estimated objective function is greater than the best value $BestVal$, we expand (branch) the node; otherwise, we prune the node. When the search reaches a leaf node, the objective function value for the leaf node is calculated and is compared with the current best value. $BestVal$ will be updated, if the new value is larger.

Rescheduling Algorithm for Replication Scheme

During the expansion of an intermediate node, the rescheduling algorithm is invoked to see if the replicated copy can be put into the schedule. The proposed algorithm is described as follows.

EST and LFT as defined in Section 5.3 construct a scheduling window for each EU instance. Let x_z be an instance whose scheduled start time to the finish time is within the range of the scheduling window of the replicated copy to be scheduled. The rescheduling algorithm keeps track of all free slots. We first try to allocate a free slot for the replicated copy. If no such free slot exists, we then attempt to schedule the replicated copy by shifting the scheduled start time of x_z. Note that such shifting should not affect the scheduling times of its successors, since x_z is still scheduled within its scheduling window; otherwise, the computational complexity of the proposed rescheduling algorithm is not polynomial.

7.1.2 Rollback Scheme

The branch and bound algorithm for rollback scheme resembles the algorithm proposed for the replication scheme. The search space is the same, all the possible permutations of EU instances in the schedule, and is represented as a search tree. The definitions for the objective function and the estimated objective function are similar to those used for the replication scheme. The objective function is defined as the total execution time of the EU instances with rollback copies in the schedule; the estimated objective function for an intermediate node is defined as the total execution time of the instances whose rollback copies are in the partial FT schedule corresponding to the node plus the total execution time of the instances that are not in the permutation sequence of the node. The function θ in the equation 7.1 for computing the estimated objective function is defined as follows.

$$\theta^{roll}(x_i) = \begin{cases} 1 & \text{if the rollback copy of } x_i \text{ is in the partial schedule,} \\ 0 & \text{otherwise.} \end{cases}$$

Rescheduling Algorithm for Rollback Scheme

In order not to affect the scheduling times of the successors and predecessors of an EU instance, we use EST and LFT defined in Section 5.3 to construct a scheduling window for each EU instance. The rescheduling algorithm may move the scheduled start time of an instance, only if it finishes before its LFT.

Since the rollback copy has to be on the same processor where the primary copy runs, the choice for a free slot for the rollback copy is very limited. If we can not find such slot in the same processor, we will try to find one large slot for accommodating both primary and rollback copies or two slots for each of them in one processor. As mentioned in previous section, this may involve shifting the scheduled start time of an EU instance whose scheduling window overlaps with that of the rollback copy to be scheduled.

7.2 Simulation

The objective of the performance study is to investigate the usefulness of the proposed AR method. We have shown that the analytic models can estimate the system reliability accurately for the systems using static scheduling. Hence, we can use the theoretic value, MTTF, as the reliability index for the simulated system, instead of ATTF which needs a lot of computational time to get.

To reflect the improvement or degradation of the performance for the proposed approach relative to the basic schemes, we adopt *improvement gain* defined below as our performance index. Let $MTTF_{AR}$ and $MTTF_{basic}$ be the mean time to failure of a system using AR and the basic fault-tolerant scheduling approach, respectively; $G(MTTF_{AR}, MTTF_{basic})$ be the improvement gain of the AR method relative to the basic scheme, which is defined as

$$G(MTTF_{AR}, MTTF_{basic}) = \begin{cases} 0 & \text{if } MTTF_{AR} = \infty \\ & \text{and } MTTF_{basic} = \infty, \\ 1 & \text{if } MTTF_{AR} = \infty \\ & \text{and } MTTF_{basic} \neq \infty, \\ -1 & \text{if } MTTF_{AR} \neq \infty \\ & \text{and } MTTF_{basic} = \infty, \\ \frac{MTTF_{AR} - MTTF_{basic}}{\max(MTTF_{AR}, MTTF_{basic})} & \text{otherwise.} \end{cases}$$

For a system with full coverage, its MTTF is infinity (∞). We need to have a bounded improvement gain; otherwise, it is hard to justify a system with infinity MTTF and one with limited MTTF. Therefore, we use the maximum value of the MTTF from both approaches as the denominator. A positive value of the improvement gain implies that a system using our approach is more reliable than that using the basic approach.

7.2.1 Simulation Design

Figure 7.1 depicts the high level structure of the simulation for the performance study. Basically, this is an extension of the previous simulation. The component, system parameter computation, is the same in these simulated systems with different fault-tolerant scheduling approach, since they use the same method to estimate the system parameters. Their only difference lies on the scheduler. The leftmost system simulates the system using our proposed method; the center one simulates the system using the replication scheme; and the rightmost one simulates the system using the rollback scheme. The detail rescheduling algorithms for the corresponding schemes are described in Sections 5.3, 7.1.1, and 7.1.2. However, these three systems deploy the same migration algorithm developed in Section 5.4, because we intend to observe the performance among different fault-tolerant scheduling approaches, instead of migration algorithms.

In this simulation study, we follow the settings of the simulation parameters as we evaluate the performance of our extended analytic model for transient and permanent faults, because we think the set of settings we employ can generate a wide range of workloads and systems. For one set of the settings on the simulation parameters, the same 500 sets of task sets are generated for each simulated system with different fault-tolerant approach. The simulated system invokes the corresponding scheduler to obtain the FT schedule and migration schedules, computes the system parameters for both analytic models, and estimates the MTTF for both models. The results of one experiment are calculated by averaging the

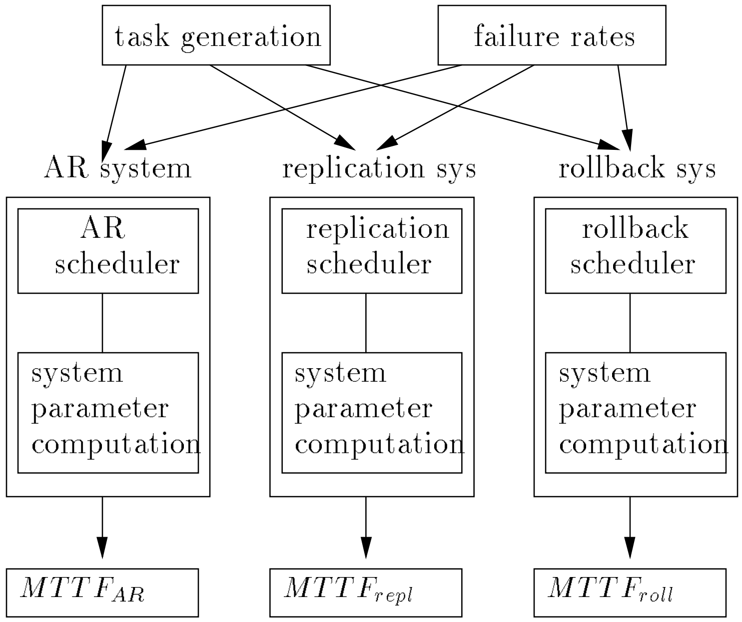


Figure 7.1: The simulation structure for performance study.

improvement gains obtained from all sets of the settings for that experiment listed in the corresponding tables.

7.2.2 Simulation Results

We conduct experiments for evaluating the performance of the proposed scheme compared with the basic fault-tolerant scheduling approaches under various system workloads and failure rates. In the following figures, "REPL-1" represents the improvement gain of the AR method relative to the replication scheme for the analytic model which models transient faults only; "REPL-2" represents the improvement gain of the AR method relative to the replication scheme for the analytic model which models transient and permanent faults; "ROLL-1" represents the improvement gain of the AR method relative to the rollback scheme for the analytic model which models transient faults only; "ROLL-2" represents the improvement gain of the AR method relative to the rollback scheme for the analytic model which models transient and permanent faults.

Figure 7.2 presents the results for the experiment for varying the permanent failure rate; the settings are shown in Table 6.9. The replication scheme generally performs better than the rollback scheme. It is primarily because a replicated copy has more choice of free slots than a rollback copy. A rollback copy has to be in the same processor where the primary copy runs, while a replicated copy does not have to. Therefore the replication scheme has more redundant copies in the schedule than the rollback scheme and the improvement gain relative to the replication scheme is lower comparing with the rollback scheme. The change of permanent failure rate does not affect the performance. The AR method outperforms, because it has positive values of improvement gain over various permanent failure rates.

Figure 7.3 dislays the results for the experiment that changes the transient failure rate of the system. The improvement gains over various failure rates are quite stable, because the resulted FT schedules remain the same under different failure rates. The performance of these three approaches is insensitive to different failure rates. The AR method performs better than the other two fault-tolerant scheduling schemes.

The results for the experiment varying the number of processors are shown in Figure 7.4. As the number of processors increases, the system workload decreases and the number of free slots increases. The replication scheme has more replicated copy under a light loaded system, because the system has more free space. Therefore, its performance becomes competitive in light loaded systems. When the workload is heavy, the NFT schedule has less space for redundant copies, no matter what fault-tolerant scheduling we choose. However, even with limited resources left, the AR method still has better performance, because it has two options to choice for redundant copy and utilizes resources more efficiently.

As for the rollback scheme, its performance degrades comparatively as the workload decreases. The primary reason for this phenomenon is because our proposed AR method makes more systems with full coverage as the workload decreases. Hence, the improvement gain relative to the rollback scheme raises.

The next experiment attempts to measure the impact on the change in the scale of period. The experimental results are shown in Figure 7.5. Large scale of period does not imply the decrease of workload, because of the interleaving of EU execution. However, the scheduling window raises as the scale of period gets large. For larger scheduling window, AR algorithm can shift the scheduled times of EUs so that it has the flexibility of put more rollback or replicated copies into schedules. Its increased improvement gains are caused by combining the merits of the other two algorithms.

Figure 7.6 shows the results for the experiment that changes the range of the execution time to generate various workloads. As the maximum execution time increases, the average execution time increases and the workload increases. With the same timing constraints, the same release time and deadline, a long EU has less chance having a rollback copy than a short one, because long EU has smaller slack time. The improvement gain relative to the replication scheme declines as the average execution time raises. This is mainly because EUs in such heavy loaded system most likely have replicated copies, instead of rollback copies, and hence the benefit of having rollback copy in our method is less visible.

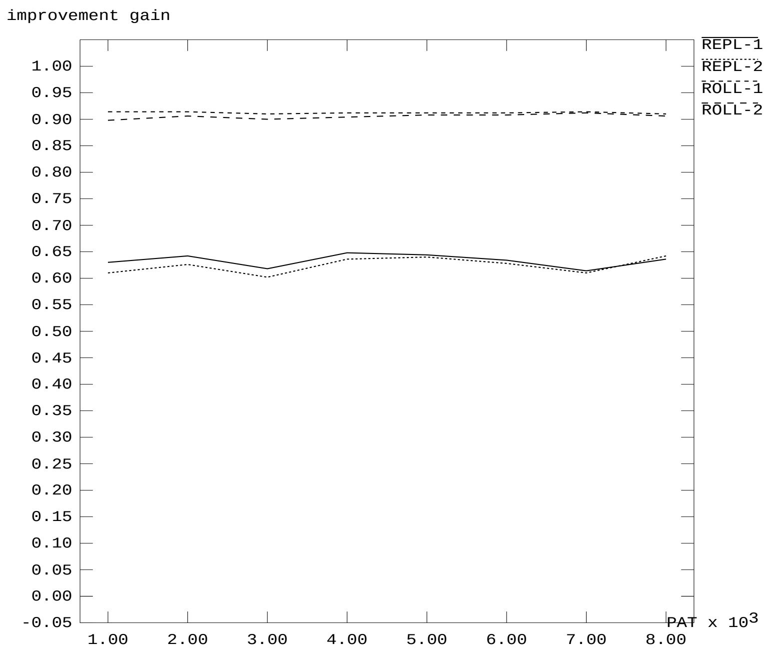


Figure 7.2: Results on evaluation of fault-tolerant scheduling approaches for varying PAT.

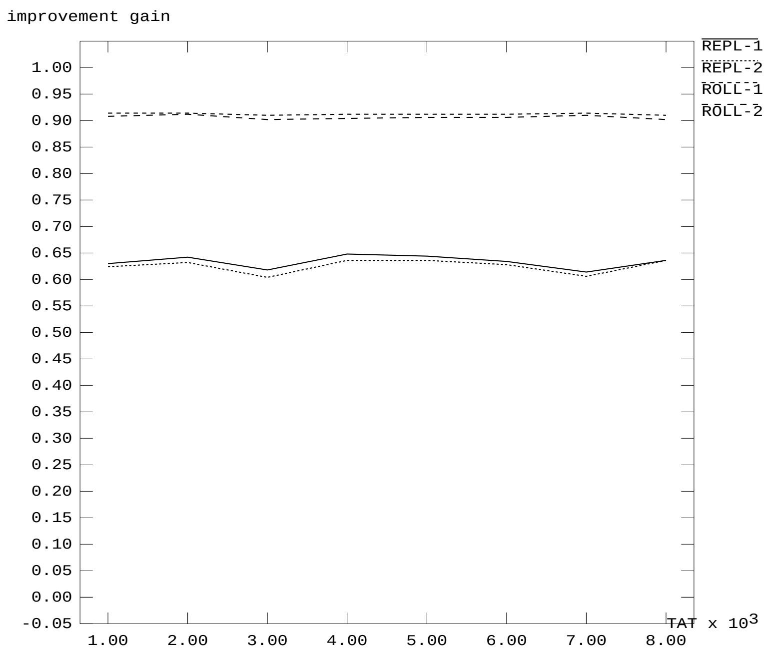


Figure 7.3: Results on evaluation of fault-tolerant scheduling approaches for varying TAT.

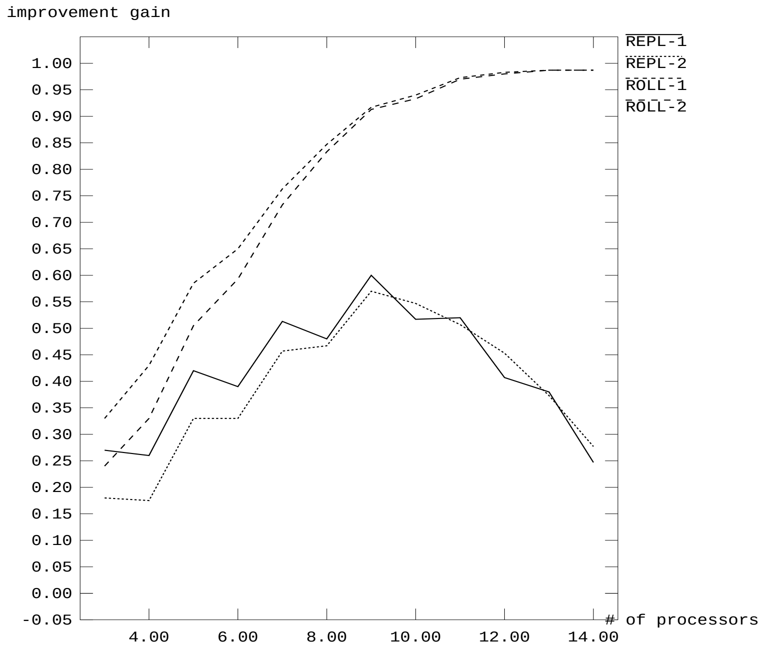


Figure 7.4: Results on evaluation of fault-tolerant scheduling approaches for varying the number of processors.

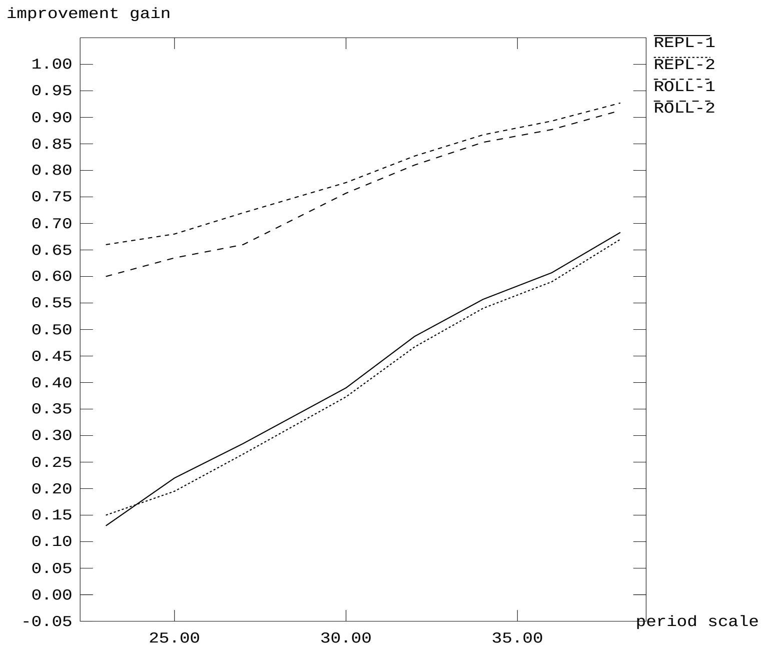


Figure 7.5: Results on evaluation of fault-tolerant scheduling approaches for varying the scale of period.

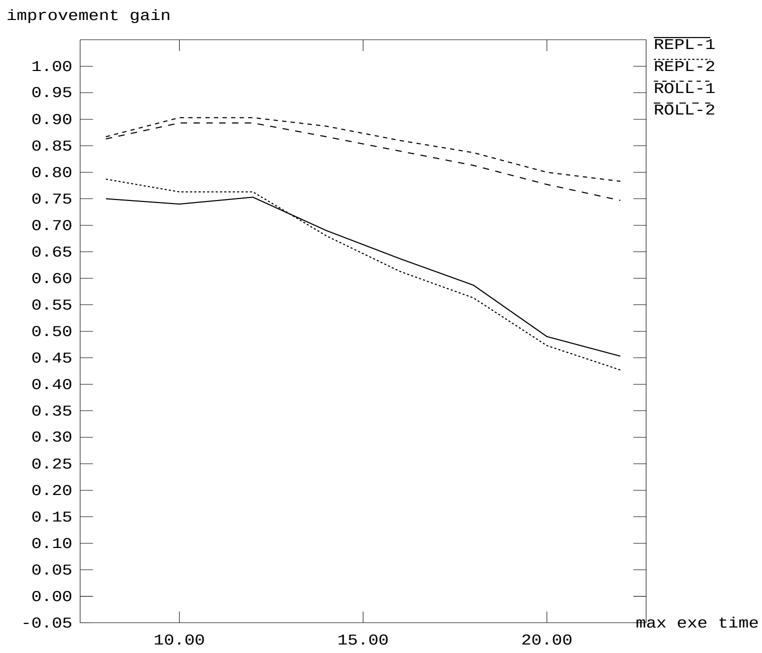


Figure 7.6: Results on evaluation of fault-tolerant scheduling approaches for varying the range of execution time.

Chapter 8

Conclusion

In this dissertation, we have addressed the real-time scheduling problem with a view of providing support for satisfying resource and fault-tolerance constraints in a comprehensive and efficient way. We propose a resource synchronization protocol for use in multiprocessor hard real-time systems that allows jobs to simultaneously lock more than one global resource. The synchronization protocol may be applied to both static and dynamic priority scheduling, and prevents deadlock and transitive blocking. We also devise sufficient utilization bounds to guarantee schedulability. Our experimental performance studies show that the proposed protocols that allow nested global critical sections have better performance than the protocols which do not allow a job to simultaneously lock multiple global semaphores. The improvement is due to the fact that our protocols provide flexible granularity of synchronization and hence allow a greater degree of parallelism.

In Chapter 4, we discuss the issues on incorporating execution time information into optimistic concurrency control (OCC) algorithms and demonstrate that proper use of the knowledge of execution time can improve data conflict resolution decision. Several OCC algorithms using the knowledge of execution time are developed and evaluated. We develop a simulation model to analyze the performance of the proposed algorithms. With the knowledge of execution time, we can predict in advance if a transaction can not make its deadline (i.e. non-restartable) and discard it earlier before its deadline expires. By allocating the resources, saved from the early discarded non-restartable transactions, to restartable transactions, the proposed protocols can commit more transactions than the baseline algorithm and minimize the miss ratio.

We also present a scheduling algorithm to improve system reliability and to meet the timing constraints. The proposed fault-tolerant scheduling method uses both rollback and replication techniques while scheduling secondary copies of EUs. The experimental results reveal that the hybrid technique combines the benefits of rollback only and replication only schemes and has more secondary copies in schedules than the other two schemes. One side benefit of the algorithm is that the resources reserved by rollback copies may be reclaimed by aperiodic tasks, if no fault occurs.

We devise reliability models as a tool for real-time system designers to compare different fault-tolerant scheduling schemes and develop an abstract system model to simulate RTS employing different scheduling schemes. The models can also be used to evaluate alternative rescheduling and migration algorithms used in the proposed approach. We use such tool to demonstrate that the proposed approach provides more reliable systems than rollback only and replication only schemes.

8.1 Future Research

There are several interesting extensions of the research.

Workload characteristics: The results obtained in this dissertation are based on synthetic characteristics of workloads on an abstract system model. Further research is necessary for the issues on the priori information about workload in the existing RTS and the representation of workload characteristics.

Overheads on resource synchronization: Most of synchronization protocols require many context switches, but the overhead is ignored in most research. Further investigation is needed to find ways to limit or estimate such overhead.

Concurrency control: We have seen the significance of the knowledge of the execution time on data conflict resolution of OCC algorithms. The impact of such knowledge on other classes of concurrency control needs to be explored. For example, locking protocols use locks to control the access of shared resources and deploy conflict avoidance rules to enforce data consistency and prevent deadlock. It would be interesting to see if execution time information can help locking protocols make better conflict avoidance decision. On the other hand, our proposed algorithms do not use such information to classify validating transactions. For instance, an OCC algorithm, considering the restartability of validating transactions, might not sacrifice validating transaction which is non-restartable. Extended research can focus on the classification of validating transaction and propose algorithms that make distinct conflict resolution decisions on different classes of validating transactions.

Fault-tolerant scheduling: The fault model used for the proposed fault-tolerant scheduling algorithm does not assume correlated transient failures. Experimental studies are necessary to quantify and better understand the impact of such failures. In addition, our research does not consider the criticality of applications in the same system. With limited resources left for fault-tolerance purpose, it is practical to develop a fault-tolerant scheduling algorithm which increases system reliability by maximizing the total criticality of tasks having secondary copies in schedules. The proposed scheduling algorithm does not consider communication cost. There is a need for research into distributed version of fault-tolerant scheduling algorithms. It is necessary to devise a communication allocation algorithm which minimizes communication overhead imposed by secondary copies.

Bibliography

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th on VLDB Conference*, pages 1–12, 1988.

[AGM89] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th on VLDB Conference*, pages 385–396, 1989.

[AGM92] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, Sep. 1992.

[Aud91] N. C. Audsley. Resource control for hard real-time system: A review. Department of Computer Science, University of York, UK, 1991.

[Bak90] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *Proceedings of the Real Time Systems Symposium*, pages 191–200, 1990.

[Bat80] K. E. Batcher. Design of massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, Sep. 1980.

[BB87] V. Balasubramanian and P. Banerjee. A fault-tolerant massively parallel processing architecture. *Journal of Parallel Distributed Computing*, 4(4):363–383, Aug. 1987.

[BCH91] J. Bruck, R. Cypher, and C. T. Ho. On the construction of fault-tolerant cube-connected cycles networks. In *Proceedings 1991 international Conference on Parallel Processing*, volume 1, pages 692–693, 1991.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976.

[BMHD89] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *Proceedings of Data Engineering*, pages 470–480, 1989.

[Bur90] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, 1990.

[CC90] C. H. Chen and V. Cherkassky. Task reallocation for fault tolerance in multiprocessor systems. In *Proceedings of the IEEE 1990 National Aerospace and Electronics Conference*, pages 495–500, 1990.

[CHA94] S. Cheng, S. I. Hwang, and A. K. Agrawala. Mission-oriented replication of periodic tasks in real-time distributed systems. submitted to IEEE Parallel and Distributed Technology, 1994.

[CJL89] M. J. Carey, R. Jauhari, and M. Livny. Priority in dbms resource scheduling. In *Proceedings of the 15th on VLDB Conference*, pages 397–410, 1989.

[CL90a] M. I. Chen and K. J. Lin. Dynamic Priority Ceilings: A concurrency control protocol for real-time systems. *Journal of Real Time Systems*, 2:325–246, 1990.

[CL90b] M. I. Chen and K. J. Lin. Schedulability conditions of real-time periodic jobs using shared resources. Technical Report UIUCDCS-R-91-1658, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[CL91] M. I. Chen and K. J. Lin. A Priority Ceiling Protocol for multiple-instance resources. In *Proceedings of the Real Time Systems Symposium*, pages 141–148, 1991.

[CR72] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[CT93] C. M. Chen and S. K. Tripathi. An optimistic concurrency control algorithm in real-time database systems. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing*, pages 275–280, 1993.

[CT94] C. M. Chen and S. K. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CSTR-3253, UMICAS-TR-94-42, Dept. of Computer Science, University of Maryland at College Park, 1994.

[CT95a] C. M. Chen and S. K. Tripathi. An analytic model for the reliability of real-time systems. In *IASTED International Conference on Applied Modelling, Simulation and Optimization*, 1995.

[CT95b] C. M. Chen and S. K. Tripathi. Fault-tolerance scheduling in real-time systems. In *ISCA International Conference on Computer Applications in Industry and Engineering*, 1995.

[CTB94] C. M. Chen, S. K. Tripathi, and A. Blackmore. A resource synchronization protocol for multiprocessor real-time systems. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume 3, pages 159–162, 1994.

[CTC94] C. M. Chen, S. K. Tripathi, and S. Cheng. A fault-tolerance model for real-time systems. In *The 1994 IEEE Workshop on Fault-Tolerant and Distributed Systems*, 1994.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lome, and I. L. Traiger. The notions of consistency and predicate locks in a data base system. *Communications of the ACM*, 19(11), Nov. 1976.

[Gel79] E. Gelenbe. On the optimum checkpoint interval. *Journal of the Association for Computing Machinery*, 26(2):259–270, Apr. 1979.

[GJ75] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, Dec. 1975.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, 1979.

[GRW88] R. Geist, R. Reynolds, and J. Westall. Selection of a checkpoint interval in a critical-task environment. *IEEE Transactions on Reliability*, 37(4):395–400, Oct. 1988.

[Har84] T. Harder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[HCL90] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the Real Time Systems Symposium*, pages 94–103, 1990.

[HSTR89] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the Real Time Systems Symposium*, pages 144–153, 1989.

[KJC89] S. Y. Kung, S. N. Jean, and C. W. Chang. Fault-tolerant array processors using single-track switches. *IEEE Transactions on Computers*, 38(4):501–514, Apr. 1989.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[KS89] C. M. Krishna and A. D. Singh. Modeling correlated transient failures in fault-tolerant systems. In *Proceedings IEEE Fault-Tolerant Computing Symposium*, pages 374–381, 1989.

[LA90] P. A. Lee and T. Anderson. *Fault Tolerance, Principles and Practice*. Springer-Verlag, New York, NY, 1990.

[LL73] C.L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[LS93] K. J. Lin and S. H. Son. Real-time database systems: Schedulability and serializability. 1993.

[LSP82] L. Lamport, R. Shostak, and M Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MA70] F. P. Mathur and A. Avizienis. Reliability analysis of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair. In *1970 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, volume 36, 1970.

[MA91] D. Mossé and A. K. Agrawala. Resilient computation graphs for distributed real-time environments. Technical Report CS-TR-2613, Dept. of Computer Science, University of Maryland at College Park, 1991.

[Mar67] J. Martin. *Design of Real-Time Computer Ssytems.* Prentice-Hall, Englewood Cliffs, NJ, 1967.

[MN82] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[Mos93] D. Mossé. *A Framework for the Development and Deployment of Fault Tolerant Applications in Real-Time Systems.* PhD thesis, University of Maryland, 1993.

[MSA92] D. Mossé, Manas Saksena, and Ashok Agrawala. The Design of the MARUTI System. In *Proceedings Complex Systems Engineering Synthesis and Assessment Technology Workshop.* Naval Surface Warfare Center, July 1992.

[Nak93] H. Nakazato. *Issues on Synchronizing and Scheduling Tasks in Real-Time Database Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Jan. 1993.

[Neu56] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, 1956.

[NK83] Nicola and Kylstra. A model of checkpointing and recovery with a specified number of transactions between checkpoints. In *Proceedings of Performance 83*, pages 83–99, 1983.

[OS94] Y. Oh and S. H. Son. Scheduling hard real-time tasks with tolerance of multi-processor failures. In *Microprocessing and Microprogramming*, pages 193–206, 1994.

[Pie65] W. H. Pierce. *Failure Tolerant Computer Design.* Academic Press, New York, NY, 1965.

[RBK90] V. P. Roychowdhury, J. Bruck, and T. Kailath. Efficient algorithms for reconfiguration in vlsi/wsi arrays. *IEEE Transactions on Computers*, 39(4):480–489, Apr. 1990.

[Rob82] J. Robinson. *Design of Concurrency Controls for Transaction Processing Systems.* PhD thesis, Carnegie Mellon University, 1982.

[Ros92] A. L. Rosenberg. The diogenes approach to testable fault-tolerant vlsi processor arrays. *IEEE Transactions on Computers*, 32(10):902–910, Sep. 1992.

[RSL88] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Real Time Systems Symposium*, pages 259–269, 1988.

[Sch90] H. D. Schwetman. CSIM reference manual (revision 14). Technical Report ACA-ST-257-87 Rev 14, Microelectronics and Computer Technology Corporation, 1990.

[SdSA94] M. Saksena, J. da Silva, and A. Agrawala. Design and Implementation of Maruti-II. In Sang Son, editor, *Principles of Real-Time Systems*. Prentice Hall, 1994. Also available as CS-TR-2845, University of Maryland.

[Son91] S. H. Son. Scheduling real-time transactions. In *Proceedings of the Real Time Systems Symposium*, pages 25–32, 1991.

[SRL87] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Dept. of Computer Science, Carnegei-Mellon University, 1987.

[SRSC91] L. Sha, R. Rajkumar, S. Son, and C. H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–799, July 1991.

[SS83] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, Aug. 1983.

[Tri82] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Ulu92] O. Ulusoy. *Concurrency Control in Real-Time Database Systems*. PhD thesis, University of Illinois at Urbana-Champaign, July 1992.

[Upa90] S. J. Upadhyaya. Rollback recovery in real-time systems with dynamic constraints. In *The 14th Annual International Computer Software and Applications Conference COMPSAC 90*, pages 524–529, 1990.

[US86] S. J. Upadhyaya and K. K. Saluja. A watchdog processor based general rollback technique with multiple retries. *IEEE Transactions on Software Engineering*, 12(1):87–95, Jan. 1986.

[VLH91] J. P. C. Verhoosel, E. J. Luit, and D. K. Hammer. A static scheduling algorithm for distributed hard real-time systems. *Journal of Real Time Systems*, 3(3):227–246, Sep. 1991.

[WKF85] C. J. Walter, R. M. Kiechhafer, and A. M. Finn. MAFT: a multicomputer architecture for fault-tolerance in real-time control systems. In *Proceedings of the Real Time Systems Symposium*, pages 133–140, 1985.

[XP90] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, Mar. 1990.

[ZR87] W. Zhao and K. Ramamritham. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *Journal of Systems and Software*, 7:195–205, 1987.