AN INTEGRATED APPROACH TO DYNAMIC TASK AND RESOURCE MANAGEMENT IN MULTIPROCESSOR REAL-TIME SYSTEMS

A Dissertation Presented

by

CHIA SHEN

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1992

Department of Computer Science

This work is funded in part by the Office of Naval Research under contracts N00014-85-K-0398 and N00014-92-J-1048, and by the National Science Foundation under equipment grants DCR-8500332 and CCR-8716858.

AN INTEGRATED APPROACH TO DYNAMIC TASK AND RESOURCE MANAGEMENT IN MULTIPROCESSOR REAL-TIME SYSTEMS

A Dissertation Presented

by

CHIA SHEN

Dedicated to

MY PARENTS

and

THE MEMORY OF MY GRANDPARENTS

THE MEMORY OF MARGE M. MOLESKY

ACKNOWLEDGMENTS

Obtaining a Ph.D. is a long task that could not have been accomplished without the many people to whom I am indebted.

I would like to express my deepest gratitude to my advisor, Prof. Krithi Ramamritham, for his essential guidance and encouragement. Through the years of working on this dissertation, and other research problems, I have learned from him what research is and how to be a researcher. His standards of research and professionalism have set the best example for me and have always given me goals to strive for in my own work and professional life. I will always be grateful to him for his patience and open-mindedness in letting me explore my own ideas, and for his prompt support even in trying times.

I also would like to thank Prof. Jack Stankovic for the past years of co-directing the Spring Project of which my dissertation is a part, and for his direction and support on my Ph.D. research. Technical discussions with Jack have always been a challenge — they have greatly sharpened my argumentation skills. I have learned a great deal about real-time systems research from working on the various research problems in the Spring Project.

My gratitude also goes to the other two members, Profs. Jim Kurose and Jim Smith, of my dissertation committee. I am privileged to have Prof. Jim Kurose as my master's thesis advisor, as well as on my Ph.D. dissertation committee. His enthusiasm in both research and teaching has been an inspiration for me. I thank him for his careful reading of the dissertation draft. His invaluable comments have greatly benefited the dissertation, especially in making the introduction chapter more informative. I am thankful for the time Prof. Jim Smith has spent in carefully reading my thesis, especially in correcting the proof of Theorem 4 in Chapter 3.

My officemates, colleagues, and friends have been a great source of help, discussion, and friendship during my graduate student years. I would like to express my special thanks to Goran for always reminding me there is life besides research, to Fuxing for providing me with the Workload Generator and for sharing Chinese culture in the office with me, to Panos for answering all my questions on the logistics in completing a Ph.D. and the UMass thesis style file which saved me weeks of formatting, to Victor for his great sense of humor which made my working hours more bearable, to Carol for all the tennis matches and AmChi lunches, to Doug for all the Advocates, and to Cris for all the MCI and AT&T information. I would like to thank all the secretaries of the Computer Science Department, and the staff members in the RCF, particularly Betty Hardy, Valerie Caro and Sharon Mallory for being patient when answering my endless questions.

My special thanks go to Mr. C. T. Shen, without whose financial support, my education in the U.S.A. would have not been possible, and to Y. P. Liu, who taught me the English language that has been essential in my technical and social communication.

I will always be thankful to my family. I am grateful for my parents for their understanding and support, and for enduring the long years of not seeing me. I thank the Molesky's — Ben, Marge, Dennis, Todd, and Anna, for making me feel at home in a foreign land.

Last, but not least, I want to thank my husband, Lory Molesky. In life one may have a best friend, a close colleague, or a dear spouse, but rarely all three in one. I am very fortunate to have Lory as such a person who knows all my ups and downs, sorrow and joy throughout my graduate student life. His understanding, love and support have made sense out of my life.

ABSTRACT

AN INTEGRATED APPROACH
TO DYNAMIC TASK AND RESOURCE MANAGEMENT
IN MULTIPROCESSOR REAL-TIME SYSTEMS

SEPTEMBER 1992

CHIA SHEN

B.S., STATE UNIVERSITY OF NEW YORK, STONY BROOK

M.S., UNIVERSITY OF MASSACHUSETTS

PH.D. UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Krithi Ramamritham

In a dynamic real-time environment, **predictability** needs to be provided in the face of *unpredictable* dynamic task arrivals and asynchronous concurrent sharing of system resources. Consequently, the underlying computer system for dynamic real-time systems needs to manage **time** explicitly as a *resource* in order to support applications' timing constraints. Such time management requires that the system be *time conscious*, *time conscientious*, and *time conserving.* These three properties encompass the *complexity, correctness* and *performance* issues of algorithms designed for a dynamic real-time system. In this dissertation, we take an integrated approach to attack the problems of algorithm design for dynamic multiprocessor real-time systems that require these three properties in the context of on-line scheduling, dispatching, and resource reclaiming.

Real-time scheduling algorithms require the use of worst case execution times of tasks. However, the worst case execution time is an upper bound, and the actual execution time of a task at run time varies between some minimum value and this upper bound due to the variabilities inherent in both the computer architecture and the software. The problem of *on-line* resource reclaiming in a multiprocessor real-time system has not been addressed previously. The research presented in this dissertation represents an initiative effort in characterizing and solving this dynamic resource reclaiming problem. We analyze the worst case run time anomalies that can occur in a multiprocessor schedule where real-time tasks have both resource and processor constraints. We have developed two resource reclaiming algorithms that are *correct* — guaranteed not to cause run time anomalies, and have bounded complexity. The effectiveness of the algorithms has been demonstrated via simulation and implementation on a multiprocessor kernel.

Predictable integration of multiple functional components is a challenge unique to real-time systems. This challenge is exacerbated by the difficulties brought about by the concurrent and asynchronous nature of *multiprocessor* systems. We demonstrate that, for a dynamic real-time system, it is not sufficient to simply analyze and prove the *static* properties of an on-line algorithm in isolation of the rest of the system components. The sharing and contention of resources, such as memory, shared system buses, and more importantly *time*, require the algorithm designer to take an *integrated* view of the system as a whole, considering the interrelationships of all the system components (be it software, or hardware) that have an effect on the dynamic timing properties of the algorithm at hand. We discuss a predictable integration of scheduling, dispatching, and resource reclamation for a distributed memory real-time multiprocessor system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

With the ever increasing demands of real-time applications, next generation real-time systems will be dynamic, large and complex [87], operating in environments that can be constantly changing. The *correctness* of such real-time applications depends not only on the logical or functional results of the computation, but also on the *timeliness* of the results. In such environments, **predictability** — the ability to reliably determine whether an application's computations can meet their deadlines — needs to be provided in the face of *unpredictable* dynamic task arrivals and asynchronous concurrent sharing of system resources.

Examples of such real-time applications include flexible manufacturing [19], multimedia communication [29, 40, 74], space shuttle and aircraft avionics [13, 32], and multirobotic systems [58]. A common feature of these real-time applications is that timing constraints are associated with many of the application activities. Moreover, many real-time applications demand *multiprocessor* computer systems to meet their requirements of *concurrent* control and computation [7, 17, 19, 44, 58, 59, 78, 85].

Consequently, the underlying computer system for dynamic real-time systems needs to manage **time** explicitly as a *resource* in order to support applications' timing constraints. Such time management requires that the system has the following three properties:

- *Time conscious* — knowing "what time it is", and "how long a computation is going to take".

- *Time conscientious* — ensuring that one computation never impinges on the time constraint of another computation if both have been guaranteed by the system to meet their deadlines.

- *Time conserving* — utilizing time effectively and efficiently.

These three properties encompass the *complexity, correctness* and *performance* issues of algorithms designed for a real-time system. To possess these properties, solutions from the following interrelated areas of real-time system research are needed:

(1) DERIVING THE WORST CASE EXECUTION TIMES AND RESOURCE REQUIREMENTS OF APPLICATION TASKS.

To be *time conscious*, we must know the execution time of application tasks and operating system functions. A real-time task with a deadline needs to be guaranteed to complete execution before its deadline even if it executes for its worst case execution time. Thus the derivation of the worst case execution time and resource requirements of real-time tasks is one of the prerequisites for the predictability of a real-time application.

(2) SCHEDULING TASKS TO MEET THEIR DEADLINES WITH RESPECT TO TASKS' WORST CASE EXECUTION TIMES AND RESOURCE REQUIREMENTS.

Scheduling is necessary to ensure the *time conscientious* property of a real-time system. The scheduling algorithms used by a real-time system must take tasks' worst case execution time and resource requirements into account, such that the ordering of the tasks in the schedule satisfies each task's time constraint and maintains the consistency of the resources.

(3) Dealing with the negative effects on system performance caused by the variance in tasks' execution times due to worst case assumptions.

Real-time scheduling algorithms require the use of worst-case execution times of tasks. However, the worst case execution time is an upper bound, and the actual execution time of a task at run time varies between some minimum value and this upper bound. The variance in task execution time can be caused by both the computer architecture and the software features.

- *architecture features* — Certain architecture features, such as cache, pipelining, as well as shared system bus and shared memory in multiprocessor systems, introduce variability in a task's execution time. For example, a memory reference will exhibit a high variance depending on whether the referenced data is cache resident or not.
- *software data dependencies* — The execution time of many program constructs, such as conditional branches and loop iterations, depend on the volume and values of input data at each invocation of a task. For instance, suppose a program iteration construct has dependencies (directly or indirectly) on input parameters of the program. With an upper bound of one thousand, one may not be sure if the loop condition would be true once, ten times, or one thousand times. In turn, because of this program data dependency, there are variabilities in the amount of time needed to access shared resources, such as shared memory and the system bus, and these variabilities in accessing shared resources may affect the actual run time execution time of other tasks that also use these shared resources.

To be *time conscious, time conscientious*, and *time conserving*, a real-time system must be able to deal with the negative effects on the performance and system correctness caused by such variance in tasks' execution times.

(4) DESIGNING EFFICIENT OPERATING SYSTEM FUNCTIONS THAT HAVE *bounded* AND *low* EXECUTION TIMES.

Typically, general purpose (multiprogrammed) operating systems strive for low *average* case execution times of operating system functions. This is motivated by the desire to achieve high throughput and to minimize the average case response times of application tasks. In contrast, real-time systems require *predictability*. Since any computational resources devoted to the execution of operating system functions will be reflected (directly or indirectly) in the execution time of an application task, the execution time of operating system functions must be bounded in order for the application task's execution time to be bounded. Predictable operating system functions and algorithms are the fundamental building blocks for next generation real-time computer systems. The mutual (but often conflicting) goals of designing *efficient* (average case) operating system functions with *bounded* and *low* worst case execution time is a challenge unique to real-time systems, and is required to support the *time conscientious* and *time conserving* properties of a real-time system. This challenge is exacerbated by the difficulties brought about by the concurrent and asynchronous nature of *multiprocessor* systems. The sharing of resources, such as memory and shared buses, mandates that the algorithm designer to take an *integrated* view of the system as a whole, considering the interrelationships of all the system components (be it software, or hardware) that have an effect on the timing properties of the algorithm at hand. For example, in designing the scheduler, one must ensure that the scheduling activity itself must not *impinge* on the time already allocated to the guaranteed application tasks. This is in contrast with the kind of functions provided by the current commercially available real-time kernels, such as the VRTX kernel [39, 75], — the execution time of the functions are in general bounded only when isolated from the other components of the system.

Many ongoing real-time research efforts have been reported in solving various problems in the areas of (1) [43, 63, 64, 66, 69, 82] and (2) [21, 37, 48, 50, 51, 53, 54, 70, 72, 84, 86, 93] above. On the other hand, little research has been done in solving problems in the areas (3) and (4), and the research described in this dissertation represents a methodical approach to the design and development of algorithms in these research areas in the context of multiprocessor real-time systems. Some interesting questions we would like to ask with respect to research area (3) are:

- Is the processor and resource time allocated to a task in a schedule always *reclaimable* when the real-time system discovers that an application task executed only for a fraction of its worst case execution time? The amount of processor and resource time left by the early completion of a task is said to be *reclaimed* if the time is used for the execution of other real-time activities (instead of being left idle).

- What are the fundamental theoretical issues in solving the resource reclaiming problem in a multiprocessor system?

- What should be the correctness criterion for the design of multiprocessor resource reclaiming algorithms?

- What performance impact, in terms of both overhead cost and performance gains, can a resource reclaiming algorithm have on the system?

With respect to research area (4) the following questions are pertinent:

- What are the requirements for designing concurrent, on-line, predictable real-time algorithms?

- Can we design *predictable* operating system functions in the face of concurrency and resource sharing?

In this dissertation, we provide answers to the above questions dealing with the real-time operating systems and resource reclaiming. Our research entails theoretical analysis, algorithm development, and performance evaluation via both simulation and implementation.

1.2 Contributions of This Dissertation

In addressing the general problems of constructing predictable dynamic multiprocessor real-time systems with respect to the open problems described in research areas (3) and (4) in the previous section, this dissertation makes the following specific contributions:

- DEALING WITH THE NEGATIVE EFFECTS CAUSED BY THE VARIANCE IN TASKS' EXECUTION TIMES DUE TO WORST CASE ASSUMPTIONS.

  The variance in tasks' execution times may result in some tasks completing earlier, compared to their worst case execution times. *Resource reclaiming* refers to the problem of utilizing resources left unused by a task when it executes less than its worst case execution time, or when a task is deleted from the current schedule. When resource reclaiming is not done correctly, *timing anomalies* can arise at run time in a dynamic real-time multiprocessor schedule with resource and processor constraints. These anomalies may jeopardize the deadlines of the tasks that have already been guaranteed. In particular, one cannot simply use any *work-conserving* scheme, i.e., any scheme that will never leave a processor idle if there is a dispatchable task, without verifying that task deadlines will not be missed. The problem of *on-line* resource reclaiming in a multiprocessor real-time system has not been addressed previously. The research presented in this dissertation represents an initiative effort in characterizing and solving this dynamic resource reclaiming problem.

- The analysis of the run time anomalies that can occur in a resource constrained multiprocessor schedule when a *work-conserving* scheme is used to dispatch tasks from the schedule at run time.

  The purpose of this analysis is two fold — (1) to determine the extent to which performance can degrade when timing anomalies occur, and (2) to demonstrate that care must be taken to design *correct* resource reclaiming algorithms, since using simple *greedy* (i.e., work-conserving) schemes may lead to incorrect run time behavior. Existing worst case analysis has been shown only with respect to shared memory multiprocessor schedules, i.e., where task-to-processor assignment is done at run time. Our analysis extends the existing results to multiprocessor schedules in which tasks are statically bound to individual processors.

- The proof of the necessary conditions under which timing anomalies can occur in a resource constraint multiprocessor schedule if a *work-conserving* scheme is used. Moreover, we prove the worst case performance degradation in terms of the maximum number of tasks missing deadlines when *work-conserving* resource reclaiming schemes are used.

- The analysis of the complexity and optimality of multiprocessor resource reclaiming.

- The development of two resource reclaiming algorithms.

While these two algorithms have *bounded* execution time, they vary in complexity. These two algorithms employ strategies that are a form of on-line *local optimization* of a feasible multiprocessor schedule. The correctness of these algorithms is also proved.

  - The performance evaluation via simulation of the two algorithms.

    To understand the performance impact of these algorithms, we have done extensive simulation studies of the resource reclaiming algorithms for a five processor multiprocessor system. We tested a wide range of task parameters, including different worst case execution times and actual execution times of tasks, task laxities, and task resource usage probabilities.

  - Demonstration of the feasibility of on-line resource reclaiming in a multiprocessor real-time system via an implementation on a prototype (Non-Uniform Memory Access) multiprocessor real-time kernel — the Spring Kernel [89].

- DESIGNING OPERATING SYSTEM FUNCTIONS THAT HAVE *bounded* EXECUTION TIMES IN A DISTRIBUTED MEMORY MULTIPROCESSOR SYSTEM.

  Few previous research efforts have addressed the problem of how to construct *time conscientious* schedulers in a concurrent *multiprocessor* real-time system, i.e., schedulers whose executions are guaranteed not to impinge on the executions of application tasks. In this dissertation, we present schemes to solve this problem.

  - The identification of a set of requirements for the design and development of on-line algorithms for real-time systems.

  - The development of schemes to construct predictable concurrent on-line scheduler and dispatcher processes in a multiprocessor real-time system.

– An integration of the scheduler, dispatcher and the resource reclaiming algorithms.

By focusing on these two research areas in building a *time conscious*, *time conscientious*, and *time conserving* real-time system, that have seen little work, this dissertation endeavors to advance the state of the art in building complex dynamic real-time systems.

1.3 Dissertation Outline

The rest of the dissertation is organized as follows.

Chapter 2 presents a characterization of the problems in dynamic task and resource management in the context of scheduling, dispatching and resource reclaiming in a real-time multiprocessor system. In Section 2.2, a set of requirements for the design of on-line real-time task and resource management algorithms is specified. Section 2.3 defines the task and multiprocessor model, as well as the terminology used throughout the dissertation.

The theoretical issues connected with the resource reclaiming problem are studied in **Chapter 3**. After introducing some preliminary concepts and definitions, we present the analysis of the run time anomalies in resource constrained multiprocessor resource reclaiming in Section 3.2. In Section 3.3 we prove the necessary conditions under which timing anomalies can occur in a resource constraint multiprocessor schedule if a *work-conserving* scheme is used. Moreover, we prove the worst case performance degradation in terms of the maximum number of tasks missing deadlines when *work-conserving* resource reclaiming schemes are used. Section 3.4 deals with the complexity and optimality issues. In Section 3.5, a correctness criterion is established for the design of on-line multiprocessor resource reclaiming algorithms.

In **Chapter 4** we present the two resource reclaiming algorithms. The complexity and the correctness of the algorithms are analyzed in Section 4.2. We also discuss the dynamic properties of the algorithms in that section. Section 4.3 describes the applicability of the resource reclaiming algorithms with respect to different task and system characteristics.

In **Chapter 5**, we attack the problem of how to construct concurrent, on-line, bounded-time scheduler and dispatchers, and how to integrate the scheduler, dispatcher and resource reclaiming predictably. In Section 5.1, we motivate our concurrent implementation of the resource reclaiming algorithm by demonstrating quantitatively the advantage of a concurrent implementation versus a centralized implementation. In any concurrent real-time system in which shared access to resources exists, it is necessary to design predictable synchronization mechanisms. We present one such multiprocessor concurrency control synchronization algorithm in Section 5.2. An issue that is crucial to the construction of any on-line real-time function is the maintenance of the *time conscientious* property of the system — that is how to make sure the system function will not impinge on the time already guaranteed to the application tasks. In Section 5.3, we demonstrate one such construction of the scheduler and the dispatcher processes. Finally, in Section 5.4, we integrate the resource reclaiming algorithms with the scheduler and dispatchers.

The performance of the resource reclaiming algorithms is evaluated and the simulation results are discussed in **Chapter 6**. Since it is difficult to collect elaborate performance statistics without affecting the true performance of the actual real-time system, we have implemented our resource reclaiming algorithms not only on the Spring Kernel, but also on a software simulator which simulates the multiprocessor Spring Kernel. In this evaluation, the overhead costs of the scheduling and dispatching activities measured on the real system are incorporated.

So far, we have only considered reclaiming the resources when a task completes its execution. However, the system can be notified of a change in a task's worst case execution time not only at task completion time, but also during the execution of a task, for example, when the evaluation of some branching condition is known. In **Chapter 7**, extensions to resource reclaiming are presented to deal with the latter.

Chapter 8 summaries the dissertation and points out future research directions.

CHAPTER 2

Dynamic Task and Resource Management: Problem Characterization

In this chapter, we first establish the requirements for the design of on-line real-time resource management algorithms in Section 2.1. With these requirements in mind, we introduce the problems in the dynamic management of tasks and resources in multiprocessor real-time systems in the context of task and resource *scheduling*, *dispatching*, and *resource reclaiming*. In introducing the problems, we also review related research results. At the end of the chapter, the system and task model, as well as the notation used throughout the dissertation are defined.

2.1 Necessary Properties of On-line Real-Time Task and Resource Management Algorithms

The unique and challenging problems in real-time applications demand new design criteria for algorithms that are to be used on-line in a real-time system. Towards this end, we have identified and established the following four requirements for on-line real-time task and resource management algorithms:

(1) *correctness*: An algorithm is correct only if it can be shown that it is logically correct, and it satisfies the *time conscientious* property, i.e., if it can be shown that the algorithm does not cause time constraint violations. An example of a time constraint violation is an impingement on the time that has already been promised to other real-time activities.

(2) *bounded complexity*: If an algorithm is to be invoked as part of the execution of an application task, then it must have a complexity that can be *bounded*. This way its cost can be incorporated into the worst case execution time of a task. For example, the complexity of a dispatcher should not depend on the number of tasks in the schedule if the number of instances of tasks to be invoked at run time cannot be bounded.

(3) *inexpensive*: To achieve *time conservation* property, we must be able to construct time efficient algorithms and functions. The overhead cost of an algorithm should be very low compared to tasks' execution times since an on-line algorithm may be invoked very frequently at run time.

(4) *effective*: An algorithm should improve the performance of a system. For example, an effective resource reclaiming algorithm should increase the number of task deadlines a real-time system can satisfy, given that the system performance is measured by the number of task deadlines met.

2.2 The Problems and Related Work

Integrated dynamic task and resource management consists of three identifiable but inter-related components — *scheduling, dispatching*, and *resource reclaiming.* These are discussed in this section. To be *time conscientious*, each of the components cannot be studied in isolation, since the scheme used in one component may greatly affect the complexity, timing properties, performance, and strategies of another. Moreover, the properties and inter-relationship of these three components influence both the *software structure* of the resulting real-time system and the *architecture features* necessary to support the system.

2.2.1 Scheduling and Dispatching

In a real-time system, the scheduling of tasks and resources entails three basic steps: *feasibility checking*, *schedule construction*, and *dispatching*. Depending on the type of applications for which the system is designed, the programming model adopted, and the type of scheduling algorithms used, each of the three steps may or may not be present in a system, and the boundaries of these three steps may not always be clear. In this section, we characterize these three steps with respect to the characteristics of real-time systems, and review current scheduling research according to this characterization.

2.2.1.1 Feasibility Checking

The central theme and uniqueness of real-time scheduling (as opposed to scheduling in non-real-time systems) is the consideration of tasks' timing requirements, usually reflected in the deadline attribute of a task, by the scheduling process. *Feasibility checking* of a set of tasks is the process of verifying whether the timing requirements of these tasks can possibly be satisfied, usually under a given set of constraints, such as resource requirements and precedence constraints. The set of tasks is *feasible* if and only if all of their constraints can be met given their timing and resource requirements. *Feasibility checking* is also termed as *schedulability analysis* in some real-time literature [43, 49].

However, different scheduling approaches may or may not have this feasibility checking step. The scheduling approaches that provide feasibility checking can be characterized as *guarantee-oriented* [73], and the ones that do not are *best effort* [29, 41, 56]. In a guarantee-oriented approach, the feasibility of a set of tasks is examined by some *scheduling policy* (i.e., a scheduling algorithm), such as shortest deadline-first, least-laxity-first, and rate monotonic scheduling algorithms, *prior* to the execution of the *entire set* of tasks. In contrast, a best effort approach is not unlike the scheduling found in a traditional (non-real-time) operating system, except that ready tasks may be queued with queueing policies that are deadline-based, e.g., in nondecreasing deadline order.

Feasibility checking can be carried out either *statically* or *dynamically*. In a static approach, the set of tasks is examined for feasibility *off-line*, assuming no other task arrivals will ever occur at run time, i.e., the set of tasks to be executed at run time will never change. On the other hand, the dynamic approach does the feasibility checking *on-line* as tasks arrive (i.e., tasks are dynamically invoked) at run time. In many cases, there exist optimal scheduling algorithms for static task systems, while none exists for dynamic task systems[1]. There is a cost associated with on-line feasibility checking, i.e., it *takes time* to examine whether the worst case requirements of a set of tasks can be met. However, it is generally believed that, for a real-time system, feasibility checking is necessary to ensure the predictability of the application, and this cost is well spent.

The complexity of feasibility checking depends on the task programming model assumed in the application. If tasks are independent with neither precedence constraints, nor resource constraints, then feasibility checking is not very complicated. On the other hand, if tasks have resource constraints, feasibility checking can be much more complicated. Two different approaches have appeared in recent real-time research efforts to deal with resource constraints among tasks:

- One approach is to analyze the resource requirements of each task, and the resource conflicts *among all the tasks*, i.e., among all the tasks that can interfere with each other given a specific scheduling algorithm, in the system. This analysis can provide the worst case blocking time due to resource contention for each task, and this worst case blocking time can be incorporated into a

[1]A dynamic scheduling algorithm is said to be optimal if it always produces a feasible schedule whenever a static scheduling algorithm with complete prior knowledge of the tasks can do so.

task's worst case execution time. Then the scheduling algorithm can assume the tasks are preemptable [70, 80]. This approach requires schemes for the run time management of the resources that correspond to the assumptions made at analysis time. For example, priority ceiling protocols must be implemented to guard resources for systems in which the schedulability of tasks is analyzed using the rate monotonic scheduling algorithm with the priority ceiling protocols.

- The other approach is to analyze and derive a set of worst case resource requirements for each *individual* task only. Then the scheduling algorithm takes into account the resource requirements of the tasks during feasibility checking. Assuming tasks are nonpreemptable, the scheduling algorithm will not schedule two tasks with resource conflicts in parallel [63, 72, 93].

A tremendous amount of research efforts in the past decades has been devoted to the development and analysis of real-time scheduling algorithms by not only computer scientists, but also by researchers from operations research, and by mathematicians. The main concern here is to be able to design approximation/heuristic scheduling algorithms that have provably good performance and have polynomial time complexity, as the majority of the real-time scheduling problems have been proven to be NP-hard or NP-complete in the strong sense [5, 11, 18, 25]. In this section, we will not attempt to review the rich set of available literature on this subject. For some of the more recent work on scheduling algorithms that are suitable for the *guarantee-oriented* approach, a reader may refer to [6, 8, 9, 10, 21, 33, 37, 47, 48, 50, 51, 53, 54, 65, 70, 72, 84, 86, 92, 93, 95]. For queueing theoretic analysis and simulation studies of scheduling algorithms suitable for *best effort* approaches, a reader may refer to [27, 36, 46, 62, 97]. Some good surveys on scheduling algorithms can be found in [14, 16, 24, 42, 93].

2.2.1.2 Schedule Construction

The *schedule construction* step depends on the model used in the feasibility checking step, and defines the operations and the complexity of the dispatching step. *Schedule construction* is the process of ordering the tasks to be executed and storing this ordering in a representation that can be used by the dispatching step. This schedule construction is usually a *direct consequence* of feasibility checking. The schedule construction of different scheduling approaches can be characterized as either *explicit* or *implicit*. By *explicit* schedule construction, we mean that the tasks to be executed are ordered and stored in the form of a *plan* such that the tasks' ordering in the plan is the order in which the tasks should be executed. In contrast, *implicit* schedule construction does not provide such an explicitly laid-out plan. Instead, the tasks to be executed are only ordered by *ranks* with respect to each other so that, at run time, the task in execution is always the highest *ranking* one. An example of the explicit schedule construction is the cyclic scheduling approach [1, 4, 55] and the heuristic nonpreemptive scheduling approach [88, 72, 96], while an example of the implicit schedule construction is the fixed priority scheduling approach, such as rate monotonic scheduling [53].

Whether the schedule construction step is explicit or implicit depends on the scheduling algorithm used in the feasibility checking step, and the characteristics of the tasks. If the scheduling approach is preemptive, i.e., the tasks can be preempted during execution, the schedule construction will be *implicit*. Since a task can be preemptive, and if the exact preemption points in a task program cannot be predicted, it is impossible to generate an explicit plan for the tasks. On the other hand, a nonpreemptive scheduling approach can always generate an *explicit* schedule for the dispatching step. However, in some nonpreemptive scheduling approaches, the explicit schedule is used only as a reference point for ensuring the run time correctness (i.e., the *time conscientious* property) of the set of tasks already guaranteed [38, 83] (this point will be clarified in the characterization of dispatching in the next section).

2.2.1.3 Dispatching

Dispatching is the final step in our characterization of the scheduling process. It is the process leading to the specific decision of exactly *which* task to execute next [2]. The complexity and requirements for the dispatching step depend on (1) the scheduling algorithm used in the *feasibility checking* step; (2) whether *explicit* or *implicit* schedule construction is done; (3) the types of tasks, i.e., whether the tasks are preemptive or nonpreemptive, and whether the tasks are independent or precedence constrained; and (4) the types of systems, i.e., uniprocessor versus multiprocessor systems.

Preemptive Scheduling

Although feasibility checking is made easier by preemptive scheduling, (in terms of theoretical optimality and complexity), the checking generally either ignores the dispatching cost, or assumes it is a negligibly small constant. However, in an actual system, dispatching is more complicated. Dispatching in this case involves preemption, context switching and possibly placing the preempted task back in the schedule according to its *rank* for future resumption. The dispatching process must incorporate timer interrupts, and the context switch time is not negligible (saving the context is pure overhead, typically takes 10 to 100 microseconds [68]).

Nonpreemptive Scheduling

In nonpreemptive scheduling, the complexity of the dispatching process depends on whether the tasks are independent and whether there are resource constraints.

[2]Note that, in this dissertation, we are not concerned with the specific operating system functions necessary to start the *actual execution* of a task, e.g., loading the context of a task. The aspects of these functions that do differ from one system to another are, in general, architecture dependent.

- If the tasks are *independent* (with no precedence constraints), and have no resource constraints, dispatching can be extremely simple.

  - *Implicit* schedule construction:

    The task to be executed is the next in *rank*. Since no explicit schedule is available, searching may be required.

  - *Explicit* schedule construction:

    The task to be executed is the next task in the laid-out schedule, and this task can always be executed *immediately*.

- On the other hand, *precedence constraints* and *resource constraints* may increase the complexity of dispatching.

  - *Implicit* schedule construction:

    If there are precedence constraints among tasks, mechanisms need to be built into both the run time task description (usually called a task descriptor) and the dispatching process to verify the eligibility (i.e., all the predecessors have completed execution) of a task to be dispatched.

  - *Explicit* schedule construction and multiprocessor scheduling:

    If tasks have resource constraints and/or precedence constraints, and the resource conflicts and precedence constraints are *scheduled away*, e.g., as in [93], then the dispatching process must not violate the resource usage integrity and precedence constraints in the given explicit schedule. There are two possible schemes for constructing the dispatcher in this case.

    * Dispatching *exactly* according to the given schedule. In this case, at the completion of a task, the dispatcher cannot simply dispatch a task *immediately* because there might be idle time intervals in the multiprocessor schedule inserted by the scheduler to conform to the

precedence constraints/resource constraints. One way to construct a correct dispatcher is to use a hardware (count down) timer in order to enforce a start time constraint.

* Since, as we have discussed in Section 1.1, the variance in tasks' execution times may result in some tasks completing earlier than expected by the scheduler with respect to their worst case execution times, the dispatcher can try to *reclaim* the time left by such early completion. In this case, correct resource reclaiming algorithms must be designed to be incorporated with the dispatching process.

2.2.1.4 Current Research

Having characterized scheduling using the above three step process, in this section, we review current real-time research according to this characterization. In Table 2.1 we have summarized the research according to our characterization of the three scheduling steps presented in the previous sections. The review is divided into *static* task systems and *dynamic* task systems[3]. In a static real-time task system, *feasibility checking* and *schedule construction* are done *off-line*. Once it is verified that the timing constraints of the set of tasks can be satisfied, at run time only dispatching has to be carried out. On the other hand, in a dynamic task system, *feasibility checking* and *schedule construction* must be done *on-line* since tasks can be invoked at unpredictable times. However, some real-time research includes both *static* and *dynamic* task systems. For these research projects, both the *static* and *dynamic* columns are marked in Table 2.1.

[3]All the research listed in Table 2.1 will be reviewed in this section except the Hawk [35] and MARUTI [52] operating systems since their scheduling-dispatching models are similar to that of one more of the others.

Table 2.1 Characterization of Scheduling Approaches

	Feasibility Checking				Schedule Construction		Dispatching	
	NF	G	static	dynamic	implicit	explicit	pre.	nonpre.
Cyclic		X	X			X		X
RMS		X	X		X		X	
MARS		X	X			X		X
MAFT		X	X		X			X
RMS-MC		X		X	X		X	
TDS	X			X	X		X	
GT		X		X	X		X	
Spring		X	X	X		X		X
Hawk	X			X	X		X	
MARUTI		X	X	X		X		X

NF = No feasibility checking, i.e., Best Effort
G = with feasibility checking, i.e., Guarantee-oriented
TDS = Time-Driven Scheduling RMS = Rate-Monotonic Scheduling
GT. = Georgia Tech. RMS-MC = RMS with Mode Change.
pre. = preemptive, nonpre. = nonpreemptive

Static Task Systems:

Most of static real-time systems assume that real-time tasks are *periodic*, and non-real-time tasks in the system can either be serviced by periodic servers, or only use the idle CPU cycles. By definition, static real-time systems are *guarantee-oriented* systems. There are three basic approaches to static task scheduling: *cyclic executive*, *fixed priority*, and *heuristic.*

The *cyclic executive* approach is nonpreemptive. The tasks are ordered in an *explicit* schedule, and each task is invoked in a fixed order throughout the execution history [55]. The explicit schedule is repeated at a specific rate called the *major* cycle. The frequency of the major cycle is set to be the least common multiple of the set of task periods. The feasibility checking is done during the schedule construction. A major cycle is divided into *minor* cycles, and the execution of each individual task is done within a *minor* cycle. To simplify the implementation, the minor cycles are generally set to the same value. In this case, the software structure and architecture support needed are very simple — A count down timer is loaded with the period of the minor cycle; once the timer reaches zero, an interrupt is generated which triggers the dispatcher to dispatch the next task in the schedule. At each task completion, if the timer interrupt has not been generated yet, the dispatcher can either dispatch a background task or enter a wait state. However, what this implies is that all the task periods must be *harmonically* related, i.e., every task must operate at integer multiples of the timer interrupt frequency. So, during the schedule construction, if the periodic requirements of the tasks are not harmonically related, the periods must be adjusted. This adjustment often leads to some increase in the frequency of execution of one or more tasks, resulting in a loss of available processor load. Examples of cyclic executives can be found in [1, 4, 55].

The second static scheduling approach uses *fixed priority* schedulers. One of the fixed priority scheduling algorithms that has been widely studied in the past years in the real-time research community is the rate monotonic scheduling (abbreviated as RMS in the rest of this section) [3, 48, 53, 70, 81, 90]. The RMS has been proven to be optimal among fixed priority scheduling by Liu and Layland [53]. RMS is a preemptive scheduling approach. Simply stated, the RMS gives the highest priority to the task with the shortest period, and always executes the highest priority task whose period has started. During the execution of a task, preemption occurs if a task with a higher priority becomes ready. Due to its preemptive nature, the schedule construction in RMS is *implicit*, and the dispatching step must incur the costs involved in task preemption, as well as task resumption. However, RMS does not require that the periods of all the tasks to be harmonically related.

Lawler in [47] proved that, for periodic tasks, there exists a feasible schedule if and only if there exists a feasible schedule for the LCM (the least common multiple) of the set of periodic tasks. Therefore, the heuristic scheduling approach in general employs various heuristic techniques for scheduling non-periodic tasks to schedule periodic tasks within the LCM. Heuristic scheduling is especially necessary in systems where the task system contains dependencies and complex requirements, rather than simple independent tasks.

One of the examples of static real-time scheduling research using heuristic scheduling is presented in [71]. In this approach, components of precedence-related periodic tasks are allocated and scheduled across sites in a distributed system and a schedule can be constructed statically. The scheduling algorithm can deal with tasks' precedence constraints, communication costs, replication requirements, as well as periodicities, and is based on the branch-and-bound method. This scheduling algorithm is particularly suitable for determining the allocation and scheduling of complex periodic tasks at preallocation time [89], or in a static system such as MARS for the feasibility checking and schedule construction steps. MARS (MAintainable Real-Time System) [45] is a distributed fault-tolerant real-time system developed at the Technical University of Vienna, Austria. The real-time research in MARS is designed for real-time process control (e.g., the control of a steel rolling mill) in which all the control tasks and their timing and resource requirements are known a priori. The system is designed for peak load conditions, i.e., all possible worst case frequencies of events are considered at design time. All the tasks must complete within a duty cycle[4]. Feasibility checking is done off-line, and *explicit* schedule construction is required. Using a special hardware supported global clock, all CPU schedules and communication bus schedules are synchronized exactly to the message

[4]MARS is designed for process control in which a set of controlling activities with precedence constraints must finish within a single deadline. So a duty cycle is equivalent to the LCM and is the period within which all the tasks must be completed.

sending and receiving times. The run time dispatching of the real-time tasks is precisely according to the stored schedule table, although CPU idle time can be used for non-real-time tasks.

Another example of static real-time scheduling using heuristic is the MAFT (a Multiprocessor Architecture for Fault Tolerant) [38] system. MAFT is a real-time fault-tolerant multiprocessor system, developed by the Allied-Signal Aerospace Company, for high-reliability, critical control applications. The entire set of tasks in the system are fully ordered by fixed priorities. Tasks may have precedence constraints, and all the tasks are periodic. External events may be *polled* at pre-defined time intervals. The scheduling approach is nonpreemptive, and uses a variation of the *list-scheduling* algorithm [16] for *feasibility checking.* After the initial feasibility checking, tasks are only maintained in a priority list. The dispatching process must incur a variable amount of overhead to decide which task to execute next, due to precedence constraints. The fault-tolerance of the system is achieved through replication of tasks and voting on the start and completion of tasks. In the next section, we will return to discuss MAFT with respect to the schedule stablization problem.

Dynamic Task Systems

Research in dynamic real-time scheduling has only emerged in recent years. In a dynamic system, tasks can be invoked at unpredictable times, and thus run time scheduling must be provided by the real-time system. One of the crucial issues unique to a dynamic system is the problem of ensuring the *time conscientious* property in the face of concurrent execution of the scheduling process and the application tasks. One must carefully *quantify* the costs for each of the steps in the scheduling process so that it can be proven that the scheduling activity will not impinge on the time already guaranteed to the application tasks or the other system activities. We will show in Chapter 5 that if care is not taken, this kind of undesirable impingement will happen. Except the Spring Kernel [89], none of the research reviewed in this section correctly addresses this issue.

In [56], a Time-Driven Scheduling model (abbreviated as TDS) is proposed. It is a *best effort* approach in which the system tries to maximize the sum of the *values* of the tasks completed under overload conditions. Since preemptive scheduling is employed in this model, the schedule construction is *implicit*. The feasibility checking step is done at the task dispatching time, instead of task arrival time, and this checking is only applied to the single task to be executed next. The dispatching step may use various policies to decide the next task to be executed, and to decide which task to be discarded in the case of an estimated overload.

There have been a number of extensions to the rate monotonic scheduling model, including one to deal with mode changes [81]. The execution of a real-time application may progress through one mode of execution to another. With the mode change protocol, additions and deletions of tasks, as well as changes in tasks parameters (e.g., increasing the execution frequencies of some tasks) in a task system using the rate monotonic scheduling model can be used in dynamic real-time systems for periodic tasks. In this case, feasibility checking can be done on-line. So whenever a task addition is desired, a feasibility check must be carried out. As in the original RMS approach, the schedule construction is still *implicit* here and dispatching must be able to take care of task preemption.

A multiprocessor scheduler used with real-time robotics applications has been described in [7, 98]. (Since this research is being mainly developed at the Georgia Institute of Technology, we use the abbreviation GT in Table 2.1.) This multiprocessor scheduler is *guarantee-oriented*, employing a variation of the preemptive deadline-first scheduling algorithm for its *feasibility checking*. In addition to deadlines, tasks may have start time constraints. The schedule construction is *implicit* since, once it is verified that a set of tasks can meet their deadline requirements, tasks are only maintained in a list in nondecreasing deadline order. Besides the preemption costs for the preemptive scheduling, the dispatching process (a multiplexor in their terminology) also incurs a cost linear in the number of tasks in a current schedule in order to decide which task to execute next. This is because the task with the smallest deadline does not necessarily have a start time less than or equal to current dispatching time. The costs of the scheduling activity, including both the costs of feasibility checking and dispatching, are not carefully quantified. Also it is not shown that the scheduling will not impinge on the time already guaranteed to the application tasks.

The Spring Kernel [89] is an on-going real-time research project in building distributed multiprocessor real-time systems for large and complex real-time applications. The research presented in this dissertation is part of this on-going research, and concentrates on the on-line scheduling-dispatching-resource-reclaiming aspect of the system. In the Spring Kernel, scheduling is *guarantee-oriented.* Both static and dynamic *feasibility checking* are supported. The static scheduling uses the heuristic scheduling algorithm proposed in [71], and the dynamic scheduling employs the heuristic scheduling algorithm proposed in [72, 95]. For both static and dynamic tasks, the schedule construction is *explicit* and *nonpreemptive* dispatching is assumed. As we demonstrate in the rest of the chapters in this dissertation, the costs of the scheduling activities are carefully quantified and accounted for in the Spring Kernel. Thus, one of the goals of this dissertation is to demonstrate that real-time scheduling research must address the problem of how to construct *time conscientious* schedulers and dispatchers in an integrated fashion.

2.2.2 Resource Reclaiming

In a *guarantee-oriented* real-time system with either *implicit* or *explicit* schedule construction, in order to guarantee that real-time tasks will meet their deadlines once they are scheduled, most real-time scheduling algorithms schedule tasks with respect to their *worst case* execution times. Since this worst case execution time is an upper bound, the actual execution time may vary between some minimum value and this upper bound, depending on the various factors that cause variance in a task's execution time, such as the system state, the volume and value of input data, the amount of resource contention, and the task's semantics.

Resource reclaiming refers to the problem of utilizing resources left unused by a task when it executes less than its worst case execution time, or when a task is deleted from the current schedule. Task deletion occurs either during an operation mode change [81], or when one of the copies of a task completes successfully in a fault-tolerant system and the fault semantics permits deletion of the other copies from the schedule [15]. Resource reclaiming is an important issue especially in dynamic *guarantee-oriented* real-time systems, in which overload with respect to the worst case assumptions of task execution times can occur, and it has not been addressed in practice.

Resource reclaiming is straightforward given a uniprocessor nonpreemptive schedule because there is only one task executing at any moment on the processor. However, resource reclaiming on multiprocessor systems for tasks with resource constraints is much more complicated. This is due to the potential parallelism provided by a multiprocessor system, and the potential resource conflicts among tasks. Resource conflicts mandate the system to either employ *nonpreemptive* scheduling, or to restrict the preemption of a task within critical sections. At run time when the actual execution time of a task differs from its worst case computation time in a given nonpreemptive multiprocessor schedule with resource constraints, timing anomalies may occur. These anomalies may cause some of the already guaranteed tasks to miss their deadlines.

In the rest of this section, we review previous research on two topics: (1) the analysis of the worst case timing anomalies in multiprocessor scheduling, and (2) schedule stablization algorithms for static multiprocessor scheduling.

2.2.2.1 Multiprocessor Scheduling Anomalies

Multiprocessor scheduling anomalies have been studied previously in the context of list scheduling for task systems with precedence constraints. The objective has been to minimize schedule lengths and a *shared-memory* multiprocessor system has been assumed. Graham [30] studied the anomalies that can arise when any of the following four parameter changes occur in a given multiprocessor schedule: (1) *reducing* execution times of some tasks, (2) *weakening* some of the precedence constraints, (3) *increasing* the number of processors, and (4) using a *different priority list*. Since cases (1), (3), and (4) were first discussed, qualitatively, by Richards [76], we call them *Richard's anomalies.*

A change in the last parameter, i.e. a different priority list, causing an increase in the schedule length is easy to understand. This simply corresponds to using a different scheduling algorithm. Intuitively, one would expect the first three parameter changes to decrease the length of the original schedule. However, examples in [30] demonstrate that this is not always the case and the opposite can happen. It was further proven that such an anomaly is not necessarily caused by a poor choice of the schedule for the task set, but is inherent in the *greediness* of the scheduling model. A general worst case bound on the difference between the schedule lengths of two runs of the same set of tasks was proved by Graham in the following theorem:

Graham's Theorem: Given two runs of the same task set, which are related by varying any one of the four parameters stated above, we have the following bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{m-1}{m'}$$

where ω and ω' are the schedule lengths of the two runs respectively, and m and m' are the number of processors used in the two runs. It was shown that this bound was the best possible. Here, if we use the same number of processors for both runs, the above bound becomes:

$$\frac{\omega'}{\omega} \leq 2 - \frac{1}{m}$$

Thus, with the same number of processors, the same precedence constraints and priority list, by reducing the computation times of some of the tasks, the schedule length may increase by as much as $1 - 1/m$ times the schedule length with all tasks running to their worst case computation times.

2.2.2.2 Stability Algorithms for Static Scheduling

If the possibility of timing anomalies exists in a static real-time system, the problem is referred to as the *stablization* of the schedule. For static task systems with precedence constraints, Manacher [57] proposed a stablization algorithm. Given a schedule, precedence constraints, in addition to the existing precedence constraints, are imposed upon tasks to preserve the order in which they can run in parallel; thus preventing the occurrence of the timing anomalies. The additional precedence constraints serve as *stablizing conditions* for the schedule. The algorithm only ensures the *sufficiency* of the additional precedence constraints, i.e., some of the added precedence constraints are superfluous. The question of how to reduce or minimize the set of additional precedence constraints imposed is left open.

In recent years, there has been a resurgence of interest in studying the multiprocessor scheduling anomalies and designing *stablization* algorithms in the real-time community [26, 38, 22]. This resurgence follows from the fact that in any *guarantee-oriented* real-time system, since run time conditions cannot guarantee the invariance of the actual execution times of tasks, worst case assumptions need to be made when scheduling real-time tasks. Thus it is crucial to the correct functioning of the system to ensure that the timing anomalies will not jeopardize the guarantee made to application tasks.

In [38], a set of stablization algorithms are presented for a fault-tolerant multiprocessor real-time system (the MAFT system that we reviewed in Section 2.2.1.4). In MAFT, for fault-tolerance purposes, each task is fully replicated on all processors, and each processor executes the entire task set. Tasks can have precedence constraints, and the operations of task start and completion are contingent on a voting strategy. Therefore, the multiprocessor schedule in such a fully replicated task system is a set of uniprocessor schedules, one per processor, coordinated by the run time voting actions. Timing anomalies can occur due to the variations in the time the voting actions take, and thus cause variations on the order in which tasks become eligible to be started. The algorithms are designed to stablize the static multiprocessor schedule. The mechanisms used in the algorithms are variations of Manacher's solution, i.e., imposing additional precedence constraints among tasks in a given schedule.

2.2.2.3 Beyond Stability

Graham's bound described in Section 2.2.2.1 applies only to task systems without resource constraints. The research on stability algorithms reviewed in the last section is for *static* real-time systems only. In the context of our research on dynamic resource reclaiming, we have taken a step further in both of these areas. In particular:

- Our objective is to ensure the guarantee of tasks in the face of dynamic task arrivals.
- We assume tasks are bound to individual processors (i.e., prohibiting certain processors from executing certain tasks — the analysis of this model is posted as an open question at the end of [31].).
- Tasks may have *arbitrary* resource constraints.
- Tasks can arrive dynamically, therefore rescheduling at run time is required.

In the next chapter, we analyze the timing anomalies in multiprocessor scheduling for real-time tasks with *arbitrary resource and processor constraints*, and prove the existence of Richard's-anomaly-(1) in such a multiprocessor task setting. Although Graham has also developed other worst case bounds on scheduling lengths for multiprocessor scheduling with resource constraints (but without processor constraints) [31], these bounds were only derived for the case of using different scheduling algorithms (i.e., Richard's-anomaly-(4)). Our analysis provides the worst case bounds when tasks' worst case execution time is decreased (i.e., Richard's-anomaly-(1)). The purpose of this study is to demonstrate the necessity of designing *correct* on-line resource reclaiming algorithms that can both (1) ensure the correct dispatching of tasks already guaranteed in a given schedule, and (2) alleviate the negative effects of worst case assumptions in tasks' execution time to improve the performance of the real-time system.

2.3 Notation and Assumptions

In this section we define the multiprocessors, real-time tasks and resources, as well as the scheduling model assumed in this dissertation. We also introduce the notation used.

2.3.1 Multiprocessors

Multiprocessors are asynchronous parallel machines with multiple instruction-streams and multiple data-streams [77]. A *distributed memory* multiprocessor is one in which the physical memory is divided into modules with some placed near each processor (which allows faster access time to that memory); while in a *centralized memory* multiprocessor, access time to a physical memory location is the same for all the processors [34]. For both of these physical memory models, shared memory access (i.e., implicit communication among tasks via shared memory address space) is possible. [5]

In this dissertation, we adopt the *distributed physical memory* multiprocessor model with the capability of *shared memory access*, since this is a more general model — algorithms developed for distributed memory can also be implemented on centralized memory multiprocessors, but the converse is not always true.

2.3.2 Tasks, Resources, and Scheduling

The tasks and resources have the following characteristics:

- Tasks are well-defined schedulable entities.
- A task is independent, i.e., there are no precedence constraints among tasks, and is not preemptable.
- Resources that can be required by a task include variables, data structures, memory segments, files, and communication buffers.
- Resources can either be used in exclusive mode or shared mode [94].

[5]We adopt the terminology used in [34] that the terms distributed/centralized memory refer to the *physical* memory architecture, while the term shared memory refers to the *programming model.*

- Two tasks conflict on a resource if both of them need the same resource in exclusive mode, or one of them needs a resource in exclusive mode while the other needs the same resource in shared mode.
- Two tasks with resource conflict(s) cannot be scheduled in parallel on the multiprocessor.
- Tasks have processor constraints, i.e., each task must be scheduled and executed on the processor(s) where its code and private data are allocated. In this dissertation, we assume each task can be executed only on one processor.
- However, tasks on different processors may share resources; thus all the tasks and their resource needs must be considered together at schedule time.
- Task invocations are dynamic — this is termed dynamic task arrivals.

The processor constraint of a task abstracts the following two types of systems:

- A distributed memory multiprocessor system in which static binding of tasks to processors is done.
- A heterogeneous multiprocessor system in which a task may require some particular processor(s).

In a context of a real-time multiprocessor system with the above tasks and resources, we adopt the *guarantee-oriented* scheduling model in this dissertation.

2.3.3 Notations

- n: the number of tasks $\{T_1, T_2, \ldots\ T_n\}$.
- m: the number of processors $\{P_1, P_2, \ldots\ P_m\}$.
- s: the number of resources $\{r_1, r_2, \ldots\ r_s\}$.

Each task T_i has the following attributes:

c_i : the worst case execution time of T_i. At scheduling time, this value is known to the scheduling algorithm. But at execution time, a task may have an actual computation time $c_i' \leq c_i$.

d_i : the deadline of T_i;

$[R_i^j\ ]$: a vector of resource requirements for $1 \leq j \leq s$, denoting the set of resource requirements of T_i; each element of the vector indicates *exclusive_use*, *shared_use*, or *no_use*. A task aquires all the resources it needs at the beginning of its execution, and the resources are not release until task completion.

p_i^q : a processor id for $1 \leq q \leq m$; this is the *processor constraint* attribute of task T_i.

CHAPTER 3

Resource Reclaiming: Theoretical Issues

Given a feasible schedule, *resource reclaiming* refers to the problem of utilizing resources left unused by a task when it executes in less than its worst case execution time, or when a task is deleted from the current schedule. This chapter establishes the theoretical foundations for the on-line multiprocessor resource reclaiming problem. After some preliminary definitions, we illustrate the timing anomalies that may arise in a multiprocessor schedule with resource constraints when tasks may execute in less than their worst case execution times. To demonstrate the importance of the design of correct resource reclaiming algorithms, we analyze the worst case behavior of two seemingly simple and intuitive resource reclaiming schemes. We also prove the general conditions under which dynamic timing anomalies exist. The complexity and optimality of on-line resource reclaiming in a multiprocessor system are then analyzed. With the results of this theoretical analysis of the resource reclaiming problem, we establish a correctness criterion for the design of resource reclaiming algorithms.

3.1 Preliminaries

In this section, we introduce preliminary concepts and formal definitions that are needed for the analysis carried out in the rest of the chapter.

Definition 1: A *feasible* schedule S is a task schedule in which tasks' worst case execution times and resource constraints are all guaranteed to be met. In this dissertation, we consider nonpreemptive feasible schedules in which a *scheduled start time* (st_i) and *scheduled finish time* (ft_i) are assigned to each task T_i in the schedule such that $\forall i, ft_i \leq d_i$.

Definition 2: Given a feasible schedule S, a *post-run* schedule S' is a layout of the tasks in the same order as they are executed at run time when they execute only up to their actual execution times c'_i, where $\forall i$, $c'_i \leq c_i$. Associated with each task T_i in a post-run schedule S' is a *start time* st'_i and a *finish time* ft'_i. st'_i and ft'_i are the *actual* times at which T_i starts and completes execution, respectively, and they may be different from st_i and ft_i.

Definition 3: Given a feasible schedule S, a *projection list PL* is an ordered list of the tasks in the feasible schedule, arranged in nondecreasing order of st_i. If $st_i = st_j$ for some tasks T_i and T_j, we place the task with the smaller processor id in the PL first. Thus PL imposes a *total ordering* on the guaranteed tasks.

Definition 4: Given a feasible schedule S, a *processor projection list* PPL_q is an ordered list of all the tasks scheduled on processor P_q, also arranged in nondecreasing order of st_i, for $1 \leq i \leq n$ and $1 \leq q \leq m$.

We define a *timing anomaly* as a scenario in which given a set of tasks with a feasible schedule, a decrease in the execution time of some task results in the missing of deadline(s) of one or more tasks at run time in a post-run schedule.

Definition 5: Given a feasible schedule S of tasks, a timing anomaly occurs in a post-run schedule S' if $\exists T_i, c'_i < c_i$, and for some T_j, $ft'_j > ft_j$.

Definition 6: A *work-conserving* dispatching scheme is one that will never leave a processor idle if there is a dispatchable task.

Table 3.1 **Example 3.1:** Task Parameters.

Tasks	pid	c_i	c'_i	d_i	r_1	st_i	ft_i
T_1	2	225	125	225	-	0	225
T_2	2	175	100	400	*shared*	225	400
T_3	1	175	150	175	-	0	175
T_4	1	25	25	200	*exclusive*	175	200
T_5	1	150	75	350	-	200	350
T_6	2	100	100	500	-	400	500
T_7	1	150	125	500	*shared*	350	500

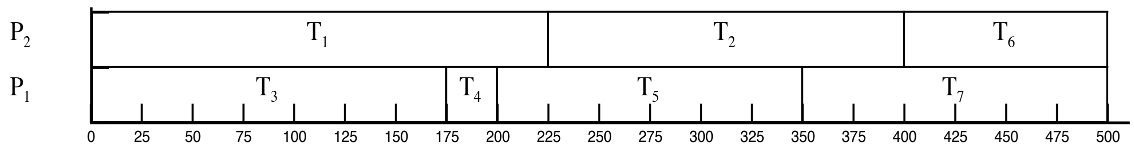


Figure 3.1 **Example 3.1:** A *feasible* schedule S according to tasks' worst case computation times.

Table 3.2 **Example 3.1:** Start Times and Finish Times produced by no resource reclaiming.

Tasks	T_1	T_2	T_3	T_4	T_5	T_6	T_7
st'_i	0	225	0	175	200	400	350
ft'_i	125	325	150	200	275	500	475
c_i	225	175	175	25	150	100	150
c'_i	125	100	150	25	75	100	125

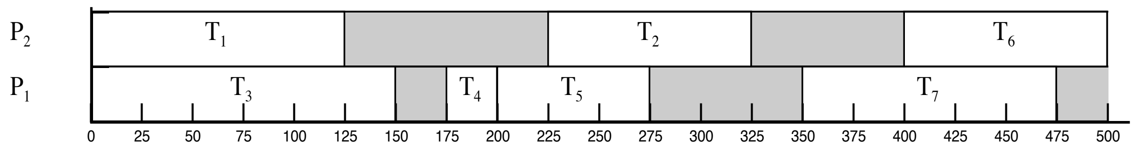


Figure 3.2 **Example 3.1:** A *post-run* schedule S' when tasks execute only up to their actual execution times and no resource reclaiming is done.

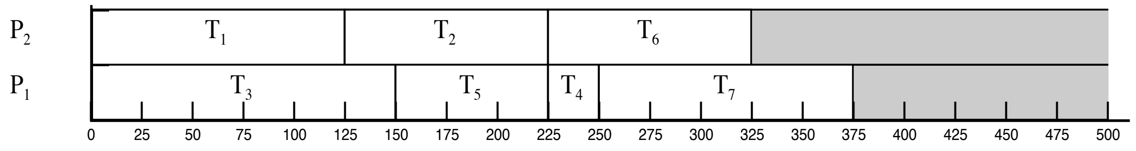


Figure 3.3 A *post-run* schedule S' produced by a *work-conserving* scheme with a timing anomaly.

We illustrate the terminology introduced so far through **Example 3.1** in Figures 3.1 and 3.2. Table 3.1 provides the attributes of a set of seven tasks. Each task requires a processor (indicated by the processor id, pid), and some need an additional resource r_1. The table also includes the worst case and actual execution times (c_i and c'_i), deadlines (d_i), and scheduled start times (st_i) and scheduled finish times (ft_i) for each task. Figure 3.1 shows a two processor *feasible* schedule S corresponding to the table entries. For the feasible schedule given in Figure 3.1, the *projection list* of S is $PL = \{T_3, T_1, T_4, T_5, T_2, T_7, T_6\}$. The *processor projection lists* are: $PPL_1 = \{T_3, T_4, T_5, T_7\}$, and $PPL_2 = \{T_1, T_2, T_6\}$. Table 3.2 and Figure 3.2 show one of the possible *post-run* schedules S' and the corresponding start times st'_i and finish times ft'_i of the tasks. Figure 3.3 demonstrates one of the possible post-run schedules with a *timing anomaly* — T_4 missed its deadline.

3.2 Timing Anomaly Analysis: 2 Case Studies

When resource reclaiming is not done correctly, anomalies can arise at run time in a dynamic real-time multiprocessor schedule with resource and processor constraints. These anomalies may jeopardize the deadlines of the real-time tasks that have already been guaranteed. In particular, one cannot simply use any *work-conserving* scheme, defined as any scheme that will never leave a processor idle if there is a dispatchable task, without verifying that task deadlines will not be missed. In this section, we examine two simple *work-conserving* resource reclaiming schemes and the possible anomalies they can cause at execution time to a resource constrained multiprocessor task schedule. The worst case bounds on these anomalous cases are presented.

3.2.1 Greedy Resource Reclaiming Scheme

The first is the *Greedy Resource Reclaiming Scheme.*

Definition 7: When a task completes execution earlier than its scheduled finish time, the *Greedy Resource Reclaiming Scheme* scans the *Projection List PL* from left to right, and dispatches the first task T_i, if any, such that the resources and the processor T_i needs are all available.

The *Greedy Resource Reclaiming Scheme* reclaims resources by not *intentionally* leaving any processor or resource idle at run time. Figure 3.3, which is the post-run schedule produced by the *Greedy Resource Reclaiming Scheme*, demonstrates the timing anomaly that can occur when the *Greedy Resource Reclaiming Scheme* is used. Although all the tasks' timing and resource constraints are satisfied in S in Figure 3.1, T_4 misses its deadline in S' in Figure 3.3. One is prompted to ask the following two questions:

(1) Why (i.e., under what conditions) does this anomaly occur?

(2) How much performance degradation results from this anomaly?

There are two types of performance degradation that we are concerned with in a real-time system:

(2.1) The first is the maximum *amount of time* by which a task misses its deadline, and

(2.2) the second is the *number of tasks* that miss their deadlines.

We will delay the answer to questions (1) and (2.2) until Section 3.3. In this section and the next, we provide answers to question (2.1). The following theorem quantifies performance degradation type of (2.1) with respect to the *Greedy Resource Reclaiming Scheme.*

Theorem 1: Let m be the number of processors and n the number of tasks. Given a feasible schedule S of length L, when the execution times of some task(s) decrease [1], in the worst case, the length L' of the resulting post-run schedule S' produced by the *Greedy Resource Reclaiming Scheme* is $\frac{m+1}{2} * L$, i.e.,

$$\frac{L'}{L} \leq \frac{m+1}{2}$$

Proof.[2] Recall that c_i is the worst case computation time of task T_i in the feasible schedule, and c_i' is the actual computation time of T_i in the corresponding post-run schedule.

Let $e(t)$ = the set of tasks being executed at time t, and $|e(t)|$ be the cardinality of the set. If $X = \{t|\ |e(t)| = 1\}$, i.e., X is the set of time intervals t in which only one task is being executed in L', then let $\tau(X)$ denote the sum of all the time intervals in X, and let $\tau(\bar{X})$ denote the sum of the rest of the time intervals in L'. Then we have

$$L' = \tau(X) + \tau(\bar{X})$$

Define

$$\mathcal{T} = \bigcup_{T_i \in X} e(t)$$

i.e., $\mathcal{T}$ is the set of tasks executed in $\tau(X)$.

Since no two tasks T_i, T_j in $\mathcal{T}$ can execute in parallel due to their resource

[1] $\exists T_i$, such that $c_i' < c_i$.

[2] The proof of this bound is inspired by a similar proof given in [23]. However their bound was not derived in the context of a decrease in the task execution time at run time.

constraints, (otherwise they would have been dispatched in parallel by the *Greedy Resource Reclaiming Scheme*),

$$L \geq \tau(X)$$

This is because the scheduling algorithm that produced the feasible schedule with length L could not have scheduled any of the tasks in $\mathcal{T}$ in parallel either due to their resource constraints. For example, $\mathcal{T} = \{A_3,\ B_1\}$ for the post-run schedule in Figure 3.5.

Since

$$mL \geq \sum_{i=1}^{n} c_i \quad and \quad c_i \geq c_i',$$

and

$$\sum_{i=1}^{n} c_i \quad \geq \quad \tau(X) + 2\tau(\bar{X})$$

(since there are at least 2 tasks executing in $\tau(\bar{X})$), we have

$$mL \geq \tau(X) + 2\tau(\bar{X})$$

Also since

$$\begin{aligned} L' &= \tau(X) + \tau(\bar{X}), \\ 2L' &= \tau(X) + \tau(X) + 2\tau(\bar{X}), \end{aligned}$$

and so

$$\tau(X) \quad + \quad 2\tau(\bar{X}) = 2L' - \tau(X).$$

Then

$$mL \quad \geq \quad \tau(X) + 2\tau(\bar{X}) \geq 2L' - \tau(X),$$

$$\begin{aligned} mL &\geq 2L' - L, \\ (m+1)L &\geq 2L', \end{aligned}$$

and thus

$$\frac{L'}{L} \leq \frac{m+1}{2}$$

Q.E.D.

The following example demonstrates that the worst case ratio of Theorem 1 is tight.

Example *Greedy*:

Let $n = 4m - 1$, and $r \geq m$. The tasks and their parameters are specified in Table 3.3. The table is formulated with variables for c_i, c'_i, and d_i to illustrate a *class* of schedules which may produce this worst case anomaly. Table 3.4 contains tasks' resource requirements. An 'e' indicates *exclusive* resource usage, and 's' *shared* resource usage. Figures 3.4 and 3.5 illustrate this worst case example with table variables $m = 3$, $Z = 10$, $\epsilon = 2$, and $\delta = 1$. Here L' is 43 and L is 26, i.e. $L' < \frac{m+1}{2} * L$. We now show that if $\epsilon \longrightarrow 0$ and $\delta \longrightarrow 0$, $L' \longrightarrow \frac{m+1}{2} * L$.

- Calculation of the Asymptotic Worst Case Ratio of L'/L:

  Let L be the schedule length of the feasible schedule S, and L' be the schedule length of the post-run schedule L'. We have

$$\begin{aligned} L &= m\epsilon + 2Z \\ L' &= mZ + Z + m(\epsilon - \delta) \\ &= (m+1)Z + m(\epsilon - \delta) \end{aligned}$$

  Then as $\epsilon \rightarrow 0$ and $\delta \rightarrow 0$,

$$L = m\epsilon + 2Z \longrightarrow 2Z$$

Table 3.3 Task Parameters: pid = processor id, c_i = worst case computation time, c_i' = actual computation time, d_i = deadline, for $1 \leq i \leq m$.

Task Type	# of Tasks	pid	c_i	c_i'	d_i
A	m	i	Z	Z	mϵ + Z
B	m	i	Z	Z	mϵ + 2Z
E	m	i	ϵ	$\epsilon - \delta$	mϵ
F	m−1	i	ϵ	ϵ	mϵ

Table 3.4 Task resource requirements.

resources	F_1	F_3	E_1	E_2	E_3	A_1	A_2	A_3	B_1	B_2	B_3
R_1			e	e	e	e					s
R_2	s	e		s	e		e		s		
R_3	s	s		s				e	s	s	

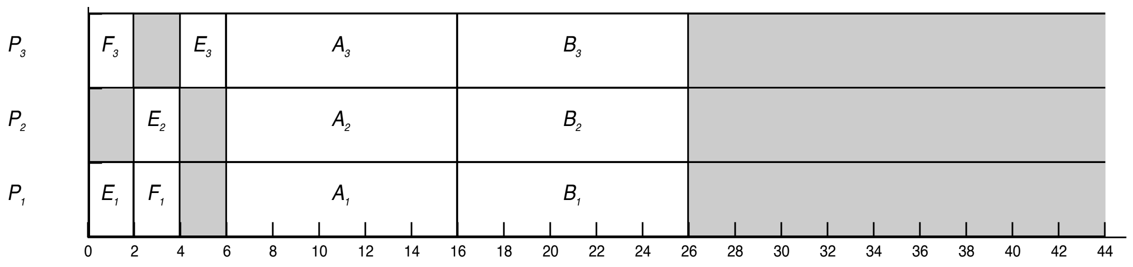


Figure 3.4 A feasible schedule S.

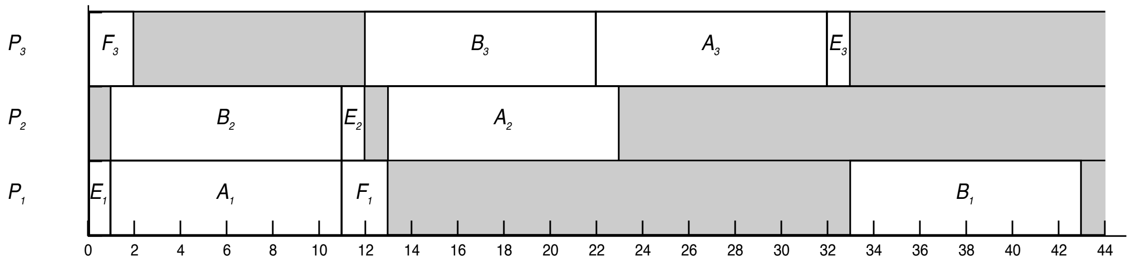


Figure 3.5 The worst-case post-run schedule produced by the *Greedy* scheme.

$$L' = (m+1)Z + m(\epsilon - \delta) \longrightarrow (m+1)Z$$

Thus,

$$\frac{L'}{L} = \frac{(m+1)Z}{2Z} = \frac{m+1}{2}$$

From the above construction of the worst case bounds for the *Greedy Resource Reclaiming Scheme*, the following corollary, which quantifies the performance degradation in terms of the maximum amount of time a task can miss its deadline by, can be derived.

Corollary 1: Given a feasible schedule S of length L, in the case of decreasing execution times of some task(s), a task T_i can miss its deadline by as much as $L' - L$ in the resulting post-run schedule S' produced by the *Greedy Resource Reclaiming Scheme*, where L' is the length of the post-run schedule.

PROOF It follows directly from Example *Greedy.*

Observation 1: The worst case of L'/L, i.e. when L'/L is maximum, occurs when S is produced by an *optimal* scheduling algorithm such that L is smallest. Here by *optimal* we mean an algorithm that can always find a feasible schedule if one exists, and if more than one feasible schedule exists, it will always find the *shortest* one. Since any practical dynamic scheduling algorithm is likely to be heuristic, the resulting L will be larger than the optimal schedule length. Thus, in general, the worst case ratio of L'/L will be smaller than $\frac{m+1}{2}$.

3.2.2 Bounded Greedy Resource Reclaiming Scheme

One may suspect that the *Greedy Resource Reclaiming Scheme* performs so poorly because it is too *greedy.* We now examine a second simple resource reclaiming scheme, the *Bounded Greedy Resource Reclaiming Scheme*, that is seemingly less *greedy.*

Definition 8: The *Bounded Greedy Resource Reclaiming Scheme* only dispatches a task T_i if (1) T_i is the *very first* task in a *processor projection list* PPL_j, and (2) the resources that T_i needs are all available.

The *Bounded Greedy Resource Reclaiming Scheme* limits the greediness by only examining the first task of each PPL_j. However, even though the *greediness* is bounded, run time anomalies can still occur when the *Bounded Greedy Resource Reclaiming Scheme* is used. The worst case ratio bound of L'/L resulting from using the *Bounded Greedy Resource Reclaiming Scheme* is even worse than what results from using the *Greedy Resource Reclaiming Scheme*, as shown in the following theorem.

Theorem 2: Given m processors, if the *Bounded Greedy Resource Reclaiming Scheme* is used, when task execution time decreases at execution time, the length L of the feasible schedule S and the length L' of the post-run schedule S' can have a worst case ratio of $\frac{L'}{L} = m$.

PROOF Any feasible schedule S must have a schedule length $L \geq \frac{\sum_{i=1}^{n} c_i}{m}$, so we have $mL \geq \sum_{i=1}^{n} c_i$. Since there is no time at which all processors and resources are idle in S' and $c_i' \leq c_i$, we must have

$$L' \leq \sum_{i=1}^{n} c_i \leq m * L$$

where m is the number of processors and n is the number of tasks. **Q.E.D.**

The following example demonstrates that the worst case ratio in Theorem 2 is tight for our multiprocessor scheduling model.

Example *Bounded Greedy*:

Let m = the number of processors, n = the number of tasks $= 3m - 1$, and r = the number of resources, $r \geq 1$. The tasks and their parameters are specified in Table 3.5. Table 3.6 contains tasks' resource requirements. Figures 3.6 and 3.7 illustrate this worst case example with $m = 3$, $Z = 20$, $\epsilon = 2$ and $\delta = 1$.

- CALCULATION OF THE ASYMPTOTIC WORST CASE RATIO OF L'/L:

  Let L be the schedule length of the feasible schedule S, and L' be the schedule length of the post-run schedule S'. We have

$$\begin{aligned} L &= m\epsilon + Z \\ L' &= mZ + m(\epsilon - \delta) \end{aligned}$$

Then as $\epsilon \rightarrow 0$ and $\delta \rightarrow 0$,

$$\begin{aligned} L &= m\epsilon + Z \longrightarrow Z \\ L' &= mZ + m(\epsilon - \delta) \longrightarrow mZ \end{aligned}$$

Thus,

$$\frac{L'}{L} = \frac{mZ}{Z} = m$$

The above construction of the worst case bounds for the *Bounded Greedy Resource Reclaiming Scheme* leads us to the following corollary.

Corollary 2: Given a feasible schedule S of length L, in the case of decreasing computation times of some task(s), a task T_i can miss its deadline by as much as $L' - L$ in the resulting post-run schedule S' produced by the *Bounded Greedy Resource Reclaiming Scheme*, where L' is the length of the post-run schedule.

PROOF It follows directly from Example *Bounded Greedy* that $ft_{A_1} = d_{A_1} + (L' - d_{A_1}) = L'$ and $d_{A_1} = L$, where ft_{A_1} and d_{A_1} are the finish time and deadline of task A_1. **Q.E.D.**

Discussion

Why is the worst case bound of the *Greedy Resource Reclaiming Scheme* $\frac{m+1}{2}$ but the *Bounded Greedy Resource Reclaiming Scheme* m? By re-examining the construction techniques demonstrating these bounds, we can intuitively answer this

Table 3.5 Task Parameters: pid = processor id, c_i = worst case computation time, c_i' = actual computation time, d_i = deadline, for $1 \le i \le m$ and $2 \le j \le m$.

Task Type	# of Tasks	pid	c_i	c_i'	d_i
A	m	i	Z	Z	mϵ + Z
E	m	i	ϵ	$\epsilon - \delta$	iϵ
F	m−1	j	ϵ	ϵ	iϵ

Table 3.6 Task resource requirements for the worst case construct of the *Bounded Greedy* algorithm.

resources	E_1	E_2	E_3	F_2	F_3	A_1	A_2	A_3
R_1	e	e	e			s	s	
R_2	s			s	e	s		s

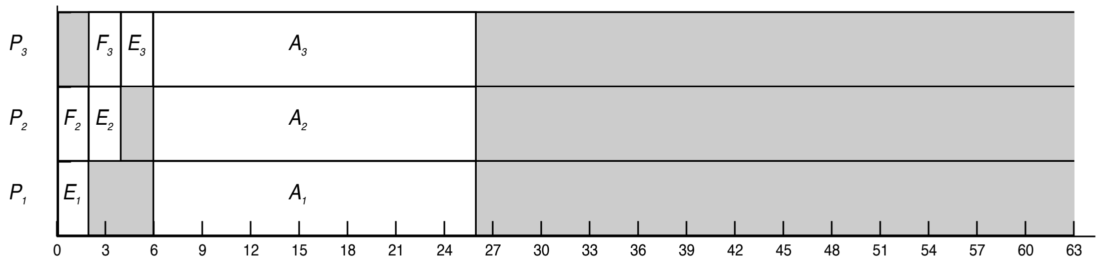


Figure 3.6 A feasible schedule S.

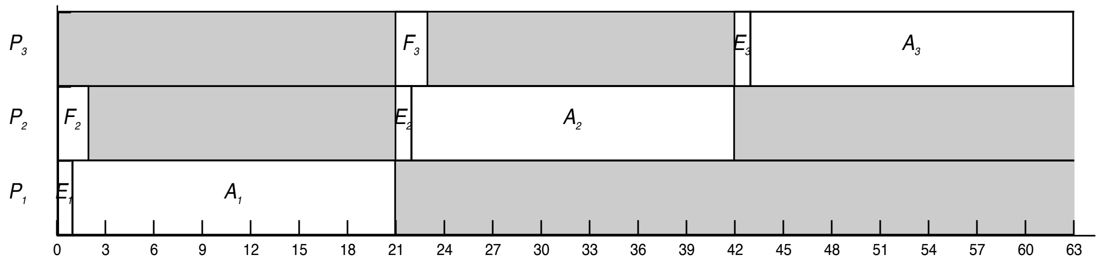


Figure 3.7 The worst-case post-run schedule produced by the *Bounded Greedy* scheme when tasks execute only up to their actual computation times.

question. First observe that the worst case examples are generated by maximizing the parallelism of the feasible schedule, while minimizing the parallelism of the post-run schedules. Given m processors, we know that the worst bound we could possibly construct is m — where the feasible schedule has m tasks in parallel, the post-run schedule will have m tasks in serial. Because the *Bounded Greedy Resource Reclaiming Scheme* can only "look ahead one task" per processor, it can miss all potential *optimal* parallelism among the tasks, thereby making it straightforward to produce a feasible schedule for which it exhibits this worst of the asymptotic worst case bounds (m) (see figure 3.5). However, producing a feasible schedule for which the *Greedy Resource Reclaiming Scheme* produces a post-run schedule length bound of m is impossible. This is because the *Greedy Resource Reclaiming Scheme* will look ahead an "arbitrary" number of tasks. Given that some task B is currently executing, if there is any task B' that has been scheduled in parallel with B in the original feasible schedule that has not been dispatched yet, then the *Greedy Resource Reclaiming Scheme* will find B' **unless** it finds some other task B'' to execute before B' can be dispatched. Although B and B'' can execute in parallel, B'' may have resource conflicts with all other tasks, thus making the parallel execution of m tasks in the feasible schedule into the parallel execution of only two tasks. Thereby, the worst case occurs for the *Greedy Resource Reclaiming Scheme* when it *consistently* finds the *wrong* tasks to execution in parallel, limiting the parallel execution to only two tasks at a time, yielding the $\frac{m+1}{2}$ schedule length bound.

3.3 Timing Anomaly Analysis: General Results

Having demonstrated the run time anomalies via the analysis of the worst case behavior of two greedy schemes, in this section, we prove the conditions necessary for the timing anomalies to occur in a resource constraint multiprocessor schedule if a *work-conserving* scheme is used. Moreover, in contrast to the worst case analysis with respect to the schedule lengths in the last section, we show the worst case performance degradation in terms of the maximum number of tasks missing deadlines as a result of using *work-conserving* resource reclaiming schemes. We first need the following definitions.

For each T_i in a feasible schedule, we can divide the rest of the tasks in the schedule into three disjoint subsets with respect to T_i defined as follows:

Definition 9:

$$T_{<i} = \{T_j : ft_j < st_i\}$$

$$T_{>i} = \{T_j : st_j > ft_i\}$$

$$T_{\simeq i} = \{T_j : T_j \not\in T_{<i} \ \ and \ \ T_j \not\in T_{>i}\}$$

Thus, $T_{<i}$ is the set of tasks that are scheduled to finish *before* T_i starts. $T_{>i}$ is the set of tasks that are scheduled *after* T_i finishes. $T_{\simeq i}$ is the set of tasks whose scheduled execution times overlap with the execution time of T_i. For example, in Figure 3.1, $T_{<5} = \{T_3, T_4\}$, $T_{>5} = \{T_6, T_7\}$, and $T_{\simeq 5} = \{T_1, T_2\}$.

Definition 10: A task T_i *totally passes* task T_j if $st'_i < ft'_j$, but $ft_j < st_i$. Thus *total-passing* occurs when a task T_i starts execution before other task(s) that are scheduled to *finish* execution before T_i was originally scheduled to start.

Assume that tasks never execute longer than their worst case execution times, there is no preemption during the execution of the tasks in S, and no arbitrary idle times are inserted. We have the following lemma which will be used in proving Theorem 3.

Lemma 1: Given a feasible real-time multiprocessor schedule S, if $\exists T_i$, such that $ft'_i > d_i$, in a post-run schedule S', then *total-passing* must have occurred.

PROOF. Since T_i does not finish by its deadline, then $ft'_i > ft_i$. And since $c'_i \leq c_i$, then $st'_i > st_i$. Assume the contradiction, i.e., assume that no *total-passing* occurred. Then the tasks in $T_{<i}$ must have been dispatched before T_i started and the tasks in $T_{>i}$ must have been dispatched after T_i finished execution. By definition of a feasible schedule, the tasks in $T_{\simeq i}$ have no resource conflicts with T_i, therefore, no matter what order these tasks were dispatched with respect to the dispatching time of T_i, they would not have delayed the dispatching of T_i. This contradicts the premise that T_i did not start by its scheduled start time st_i.

Q.E.D.

Theorem 3: Given a feasible schedule, S, a timing anomaly occurs such that some task T_k misses its deadline in S' only if the following conditions are true at some time t.

Condition 1: $\exists T_{block}$ and T_{passer} such that T_{block} is executing at t, and $st'_{passer} = t$.

Condition 2: There is a chain of tasks $\mathcal{TB} = \{T_1, ..., T_{k-1}\}$ that have not started execution at time t, such that (1) $T_{passer} \in T_{>i}$ for some $i, 1 \leq i \leq k-1$, and (2) none of the tasks can execute in parallel due to mutual resource conflicts, and (3) none of them can start execution until $t' \geq ft'_{passer}$. [3]

Condition 3: $\sum_{i=1}^{k} c'_i + (ft'_{passer} - t) > d_k$.

Conditions 1 and 2 state that there must be an occurrence of *total-passing* among a chain of tasks with mutual resource conflicts. Condition 3 gives the length of the chain in terms of execution times that may induce the missing of the deadline of task T_k.

PROOF Given Lemma 1, we only need to prove (2) and (3) in Condition 2, and Condition 3. Assume the contrary. Then $\exists T_i \in \mathcal{TB}$ such that T_i could be

[3]Note here that resource conflicts may include both resource and processor conflicts.

executed in parallel with some $T_j \in \mathcal{TB}$, or T_i could have started before ft'_{passer}. However, if this were true, the lefthand side of Condition 3 could have been reduced to be less than d_k. Then the timing anomaly would not have occurred since $\sum_{i=1}^{k} c'_i + (ft'_{passer} - t) < d_k$.

Q.E.D.

Conditions 1, 2, and 3 are *necessary*, but not *sufficient* because the chain of tasks in $\mathcal{TB}$ is a *set*, i.e., conditions 2 and 3 do not impose any specific ordering among the tasks in $\mathcal{TB}$. Therefore, it is easy to construct examples in which conditions 2 and 3 are true, but no timing anomaly occurs. The next corollary follows from the above theorem.

Corollary 3: Given n tasks remaining unfinished in a feasible schedule S, if a timing anomaly occurs in a post-run schedule S', in the worst case, the number of tasks that miss their deadlines is $n - 2$.

PROOF It follows from Theorem 3 that in the worst case, all the tasks would miss their deadlines, except tasks T_{block} and T_{passer} in the following scenario:

The worst case occurs when $\forall i, 1 \leq i \leq n, ft_i = d_i$, i.e., all the tasks in S have their scheduled finish times equal to their deadlines. Without loss of generality, let $T_{block} = T_1$, $T_{passer} = T_n$, and $\mathcal{TB} = \{T_2, T_3, ..., T_{n-1}\}$. Let $T_{passer} \in T_{>} n - 1$. Assume all the tasks in $\mathcal{TB}$ execute up to their worst case execution time. Since T_{passer} is scheduled to start after all the tasks in the feasible schedule, if it *totally passes* the tasks in $\mathcal{TB}$ in the post-run schedule, all the tasks in $\mathcal{TB}$ would miss their deadlines.

Q.E.D.

Even the necessary conditions proved in Theorem 3 are difficult to detect efficiently. The checking of Condition 2 can be combinatorially explosive since tasks with mutual resource conflicts form a clique, and finding a clique in a graph is well-known to be NP-complete [25, 91]. In this case, one not only has to find the cliques, but must also test Condition 3 for each such clique. We would like to point out that conditions that are both necessary and sufficient are difficult, if not impossible, to construct because these conditions depend on the actual execution times of tasks. We conjecture that, even if sufficient conditions could be derived, it would require the examination of all possible interleavings (combinations) of task orderings.

Given the inherent difficulties in determine the necessary and sufficient conditions, we must develop alternative strategies to prevent timing anomalies during resource reclaiming. How to accomplish this is the subject of the remainder of this chapter and the next chapter.

3.4 Complexity and Optimality

Predictability is of paramount concern in a real-time operating system. The system overhead incurred in scheduling, dispatching, and resource reclaiming should not introduce uncertainty into the system. Since we are working in a dynamic real-time environment, *time conscientious* and *time conserving* properties are of major importance for on-line resource reclaiming algorithms. There are two extreme cases that provide the lower and upper bounds on the cost in terms of time.

EXTREME CASE 1. Dispatching tasks strictly according to their scheduled start times (st). This implies no resource reclaiming and, obviously, the cost of resource reclaiming is zero.

EXTREME CASE 2. Total rescheduling of the rest of the tasks in the schedule whenever a task executes in less than its worst case execution time. Suppose the cost of a particular scheduling algorithm is $f(n)$ for each instance of scheduling involving n tasks. Then, the cost of total rescheduling would be $O(f(n))$, assuming no new task arrivals. *Total rescheduling* can only be used if the cost of this rescheduling is less than the time left unused by a task.

Since every task might complete early (i.e., execute in less than its worst case computation time), every task might incur resource reclaiming overhead. Hence, the resource reclaiming cost must be *low* (i.e., inexpensive) so that it is insignificant compared to the *worst case* execution time of a task. Moreover, the entire dispatching cost, which includes the resource reclaiming cost, should be included in the worst case computation time of a task. Consequently, the overheads of a resource reclaiming algorithm must be *bounded* so that its maximum run time cost does not vary. The straightforward approach to resource reclaiming by rescheduling will not be beneficial if the rescheduling cost exceeds the time reclaimed. Further, most scheduling algorithms have time complexities that depend on the number of tasks to be scheduled, i.e., use of these algorithms for resource reclaiming would result in unbounded overhead costs. Even if one can associate a *cap* on the maximum number of tasks the scheduler would ever consider, this maximum number can be very large compared to the average case, thus resulting in very large variance in the execution time of the scheduler. Thus a resource reclaiming algorithm which employs rescheduling does not meet the requirements of predictability. One of the challenging issues in designing resource reclaiming algorithms is to reclaim resources with a *bounded* complexity and *low* overhead, in particular, a complexity that is not a function of the number of tasks in the schedule.

Before we discuss optimality issues, let us define optimal resource reclaiming.

Definition 11: A resource reclaiming algorithm is optimal if it will always dispatch a task as long as the dispatching cannot result in any timing anomalies.

Although rescheduling can be used as an upper bound on the time cost for resource reclaiming, it does not provide *optimality*. In the following, we prove that the multiprocessor scheduling problem with resource constraints, under the assumption that each task has already been assigned to a particular processor prior to scheduling, is NP-complete in the strong sense[4].

Theorem 4: The problem of deciding whether it is possible to schedule a set of tasks with resource constraints, deadlines and processor constraints is NP-complete in the strong sense.

Proof: It is easy to see that the problem is in NP, since a nondeterministic algorithm need only guess a schedule, and check in polynomial time whether the schedule satisfies the deadlines and resource constraints of the tasks. We shall now give a polynomial transformation of an instance of the 3-PARTITION problem to an instance of our scheduling problem as follows.

We construct an instance of our scheduling problem such that the resource constraints of one subset of the tasks force the remaining subset of tasks to be scheduled in a number of equal sized "holes" left in the schedule. The scheduling of 3m tasks into the m holes of size B directly maps to the 3-partition problem.

An arbitrary 3-PARTITION instance: Let $A = \{A_1, A_2, ...A_{3m}\}$ be a set of $3m$ elements, B a positive integer, and W_i for $i = 1, 2, ...3m$ the weights of the elements in A, such that $\frac{B}{4} \leq W_i \leq \frac{B}{2}$ and $\sum_{i=1}^{3m} W_i = mB$. The problem is to decide whether A can be partitioned into m disjoint subsets such that each of which has a sum of weights of its elements equal to B.

The corresponding instance of our multiprocessor scheduling problem has two processors P_1 and P_2, two resources R_1 and R_2, and two sets of tasks, $A = \{A_1, A_2, ...A_{3m}\}$ and $\mathcal{T} = \{T_1, T_2, ...T_{2m-1}\}$, a total of $5m - 1$ tasks with the

[4]In existing literature [24, 25], the resource constrained multiprocessor scheduling problem has been proved to be NP-complete in the strong sense only for shared memory multiprocessor cases. Here we provide the proof for the special case in which tasks have been statically bound to processors.

following parameters and processor constraints (Let c be the computation time, d the deadline, and r the resource requirement vector):

- All the tasks in the set $\mathcal{T}$ are allocated to processor P_1, and all the tasks in A are allocated to P_2.
- $\forall A_i, i = 1...3m$, $c(A_i) = W_i$, $d(A_i) = mB + m - 1$, $r(A_i) = (R_2)$.
- $\forall T_i, i = 1...2m-1$,

$$d(T_i) = \lceil \frac{i}{2} \rceil B + \lfloor \frac{i}{2} \rfloor$$

$$c(T_i) = \begin{cases} 1 & \text{if } i \text{ is even} \\ B & \text{otherwise} \end{cases}$$

$$r(T_i) = \begin{cases} (R_1, R_2) & \text{if } i \text{ is even} \\ (R_1) & \text{otherwise} \end{cases}$$

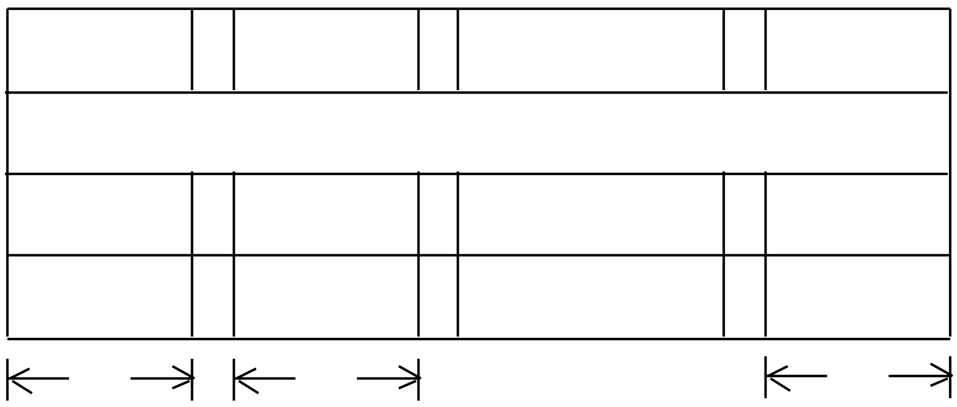


Figure 3.8 The form required of a sequence meeting the constraints of an instance of the resource and processor constrained multiprocessor scheduling problem obtained by transforming an instance of 3-PARTITION.

This transformation can clearly be done in polynomial time. Now notice that all feasible schedules for the scheduling problem instance must run the tasks in $\mathcal{T}$ in the exact order of increasing deadlines as shown in Figure 3.8, since T_1 has zero laxity (the laxity of a task T_i equals to $d_i - c_i$) and, once T_i is placed in a schedule, T_{i+1} has zero laxity. Moreover, with this ordering of tasks in $\mathcal{T}$, the $m-1$ even numbered T_i's divide the time on resource R_2 into m separate slots, each of which is of length B. There can be no idle time in any of these m slots since the sum of the computation times of the tasks in A equals to mB. Now the question is how to partition the tasks in A into m disjoint subsets such that the sum of the computation times of the tasks in each subset equals to B. Thus a feasible schedule exists **iff** the 3-PARTITION problem has a solution. Since 3-PARTITION is NP-complete in the strong sense, we have proved that the problem of deciding whether it is possible to schedule a set of tasks with resource constraints, deadlines and processor constraints is NP-complete in the strong sense as well. **Q.E.D.**

Because the resource constrained multiprocessor scheduling problem is NP-complete in the strong sense, any practical scheduling algorithm used in dynamic real-time systems must be approximate or heuristic. This implies that it is not always the case that the same scheduling algorithm will definitely find a feasible schedule when a task is removed from the original set of tasks when the task finishes execution. Thus, even though extreme case 2 provides us with an upper bound on the time complexity of the resource reclaiming problem, it does not represent the *optimal* solution in terms of being able to find feasible schedules whenever they exist, and to reclaim resources effectively. It does provide an indication of the best a system can do in reordering tasks according to available resources. Clearly, a useful resource reclaiming algorithm should have a complexity less than the total rescheduling extreme, while being just as effective.

We distinguish between two classes of resource reclaiming algorithms. One is resource reclaiming with *total-passing*, and the other is resource reclaiming without *total-passing*. A resource reclaiming algorithm that allows *total-passing* will inevitably incur higher complexity in terms of time than another that does not allow *total-passing*. This is because *total-passing* implies altering the ordering of tasks imposed by the feasible schedule, and thus is similar to rescheduling. As we have shown in Section 3.3, checking the conditions under which timing anomalies may occur could be an extremely complicated and time consuming process. To determine which task in the remaining schedule can utilize an idle period involves searching (since the scheduling problem is in fact a search problem [94]). Any search will have a complexity of at least $O(\log n)$, where n is the number of tasks to be scheduled. Since we are interested in designing resource reclaiming algorithms with *bounded* cost that can be used for dynamic real-time systems, we are motivated to concentrate on resource reclaiming algorithms without *total-passing.*

3.5 A Correctness Criterion

It is clear from Figures 3.5 and 3.7 that both of the Greedy and Bounded Greedy schemes allowed *total-passing.* Timing anomalies occur using greedy resource reclaiming schemes because tasks are not dispatched in the same order as in the given feasible schedule, and this run-time *reordering* is done without verification of the resource conflicts and timing constraints among tasks. For example in Figure 3.3, since the *Greedy Resource Reclaiming Rule* always keeps a processor busy (i.e. work-conserving) whenever possible. When T_1 executes less than its worst case execution time, T_2 is dispatched immediately since both the processor and resource it needs are available. Because of the resource conflicts between T_2 and T_4, T_4 misses its deadline. A correct resource reclaiming algorithm must be able to guarantee that this kind of run time anomaly does not occur in a post-run schedule.

From Lemma 1 and Theorem 3, we know that *total-passing* is a necessary condition for timing anomalies to happen. So one strategy to ensure the correctness of the given feasible schedule during resource reclaiming is to prohibit *total-passing.*

Definition 12: Given a post-run schedule S', a task T_i starts *on-time* if $st'_i \leq st_i$, that is, if the task T_i starts execution by or before its scheduled start time.

Definition 13: A post-run schedule S' is *correct* if $\forall i \; 1 \leq i \leq n$, $ft'_i \leq d_i$.

Lemma 2: If $\forall i \; 1 \leq i \leq n$, T_i starts *on-time* in a post-run schedule S', then S' is *correct*.

PROOF Given nonpreemptive task executions, by definition 12, if T_i starts on time, i.e., $st'_i \leq st_i$, then $ft'_i \leq ft_i \leq d_i$. So the resulting post-run schedule S' is correct.

Q.E.D.

This lemma forms the basis for the correctness of our reclaiming algorithms. Note that the lemma gives us a sufficient condition for task starting times. Our reclaiming algorithms will be designed to start tasks *on-time*. As we shall see, this strategy results in reclaiming algorithms that have bounded reclaiming overhead.

3.6 Summary

In this chapter, we addressed the various theoretical issues in dealing with the problem of on-line resource reclaiming. It was demonstrated that timing anomalies could occur if resource reclaiming was not done correctly, and possible performance problems were quantified via worst case analysis. Previous research on worst case analysis of multiprocessor schedules has not considered systems in which static binding of tasks to processors is assumed. Our analysis extends the previous results in this direction.

A set of necessary conditions for the occurrence of timing anomalies has been developed. As we have discussed, the complexity of examining these conditions is prohibitive, not to mention the fact that they would have to be done on-line in a dynamic real-time system. Moreover, the complexity of on-line resource reclaiming also comes from the difficulty of the scheduling problem itself — we proved that multiprocessor scheduling with resource and processor constraints is NP-complete in the strong sense. This implies that even the scheduling algorithm cannot be used as an optimality measure for the resource reclaiming problem.

At the end of the chapter, considering the theoretical results developed throughout this chapter, we have established a correctness criterion for on-line resource reclaiming algorithms. In the next chapter, we will present two resource reclaiming algorithms.

CHAPTER 4

Resource Reclaiming: The Algorithms

4.1 Introduction

In this section, we present our two multiprocessor resource reclaiming algorithms, the Basic Reclaiming algorithm and the Reclaiming with Early Start algorithm. Before the details of the algorithms are presented, we would like to motivate the ideas behind the algorithms.

Let us reexamine the correct post-run schedule portrayed in Figure 3.2. Actually, this post-run schedule is a result of not doing any run time resource reclaiming. Notice that between time 150 to 175 all the processors are idle. Clearly, every task in the remaining feasible schedule, i.e., tasks T_2, T_4, T_5, T_6, and T_7, could have been started at least 25 time units earlier than their scheduled start times without the risk of a deadline violation. However, with a more careful inspection of Figure 3.2. one can see that we can do even better in utilizing the idle time left in the post-run schedule. For example, T_2 could start as soon as T_3 completes execution, because $T_2 \in T_{\simeq 5}$ (see Definition 9 in Section 3.3). This can be accomplished if we can in some way represent and utilize the information given in Definition 9. Our second resource reclaiming algorithm, Reclaiming with Early Start, does precisely this.

Thus the two resource reclaiming algorithms are based on the idea that a feasible multiprocessor schedule provides task ordering information that is *sufficient* to guarantee the timing and resource requirements of tasks in the schedule. If two tasks T_i and T_j are such that $T_j \in T_{\simeq i}$ (i.e., T_j does not have any resource conflict with T_i as defined in Definition 9) in a schedule, then we can conclude that no matter which one of them is dispatched first at run time, they will never jeopardize each other's deadlines. On the other hand, if $T_j \in T_{<i}$ or $T_j \in T_{>i}$, we cannot make the same conclusion without re-examining timing and resource constraints or without total re-scheduling.

While the information encoded in the PL and the set of $\{PPL_1, ..., PPL_m\}$ is necessary for the construction of correct resource reclaiming algorithms, timing anomalies may result due to a naive use of these lists (this was demonstrated in Chapter 3). Since the PL and the set of $\{PPL_1, ..., PPL_m\}$ are *linear* lists, they cannot represent information pertaining to concurrency and resource conflicts in a multiprocessor schedule with resource constraints.

Thus, our resource reclaiming algorithms utilize the scheduled start time st_i and scheduled finish time ft_i to infer the information used in Definition 9 in Chapter 3 at run time, i.e., to identify tasks in $T_{\simeq f}$ where T_f is such that $st_f \leq st_i \; \forall i$, and to reclaim resources using these tasks. By using such a *local* optimization scheme, we do not have to explicitly examine the availability of each of the resources needed by a task in order to dispatch a task when reclaiming occurs. This keeps the complexity of the algorithms independent of the number of tasks in the schedule and the number of resources in the system — a desirable property for any algorithm that has to be used in dynamic real-time systems at run time. In the following, we assume the existence of:

(1) a feasible schedule for n tasks $\{T_1, T_2, ..., T_n\}$, which have been guaranteed with respect to their timing and resource constraints (e.g., using the algorithm presented in [72]),

(2) the corresponding *projection list* PL and m *processor projection lists* PPL_1 ... PPL_m,

(3) scheduled start time st_i, and scheduled finish time ft_i for each task entry T_i in the feasible schedule.

We also assume that we can associate a *constant* cost to determine the identity of and access the first task in the PL and the first task in each of PPL_q (These assumptions are very practical and easily achievable.).

4.2 Two Algorithms for Multiprocessor Resource Reclaiming

In Chapter 3, we defined the concept of *total passing*, and proved that *total passing* may cause timing anomalies. In the following we define *passing* and *partial passing.*

Definition 14: A task T_i *passes* task T_j if $st'_i < st'_j$, but $st_j < st_i$. Thus *passing* occurs when a task T_i starts execution before those task(s) that are scheduled to *start* execution before T_i.

Definition 15: A task T_i *partially passes* task T_j if $st'_i < st'_j$, but $st_j < st_i$ and $ft_j > st_i$. Thus T_i *partial-passes* T_j when it starts execution before T_j, that is scheduled to *start* execution before T_i, but to *finish* after T_i starts.

Observe that *passing* is the weakest of the three kinds of passings we have defined so far, i.e., both *partial passing* and *total passing* imply *passing.* Thus an algorithm that does not allow any passing will be the most restrictive. The two resource reclaiming algorithms are presented in pseudo code in Figure 4.1, 4.2, 4.3, and 4.4. Basic Reclaiming is an algorithm without *passing*, while Early Start is an algorithm with *partial-passing.* Throughout the descriptions of the algorithms, we use processor index q for the processor on which a task just completed execution, and r for any of the rest of the processors. Following are the variable definitions used in the algorithms:

- m — the number of processors.

- *reclaimed_δ* — the amount of time that has been *reclaimed*. The value of *reclaimed_δ* is cumulative, updated at each task completion, and monotonically increasing for a particular feasible schedule. *reclaimed_δ* is set to zero initially.

- T_{q_i} — the newly completed task in PPL_q for some processor q.

- T_{qf} — the first task in PPL_q after T_{q_i} is removed from PPL_q.

- T_f — the first task in the current PL.

- T_{r_f} — the first task in the current PPL_r.

- nst_i indicates a new start time for task T_i. This new start time is calculated for tasks with respect to the value of *reclaimed_δ*.

Figure 4.1 gives the outline of the major steps of the resource reclaiming algorithms. The algorithm is parameterized by the type of resource reclaiming to perform, either Basic, or Early Start. Resource reclaiming is carried out upon the completion of each task. A resource reclaiming algorithm must make two calculations. First, it must calculate the amount of *reclaimable time*, and second, it must decide *which task(s)* to apply the reclaimable time to, i.e., which task(s) can be dispatched next.

Calculating the amount of *reclaimable time* is performed by Step1 in Figure 4.2. Details of this step are the same for both the Basic Reclaiming and the Reclaiming with Early Start algorithms. Given a feasible schedule, *reclaimed_δ* is the amount of time that has been reclaimed up to the current time — its value is the cumulative amount of time during which *all* the processors and resources would have been idle if the dispatching were strictly according to the scheduled start times of tasks. Step1 updates the value of *reclaimed_δ* at each task completion. Therefore, every task

```
Whenever a task $T_{q_i}$ completes execution on a processor $q$, do
{
    $original\_reclaimed\_\delta = reclaimed\_\delta$;
    Step1($T_{q_i}$, $reclaimed\_\delta$, $PL$, $PPL_q$);
    switch ($algorithm\_choice$)
            case BASIC_RECLAIMING:
                    Step2.BASIC($q$, $reclaimed\_\delta$, $original\_reclaimed\_\delta$,
                                    $PL$, $PPL_1$, ... , $PPL_m$);
            case EARLY_START:
                    Step2.EARLYSTART ($r$, $reclaimed\_\delta$,
                                                $PL$, $PPL_1$, ... , $PPL_m$);
}
```

Figure 4.1 The resource reclaiming algorithm.

in the remaining feasible schedule can start execution at least *reclaimed_δ* amount of time earlier than their scheduled start time. After the calculation of the current value of *reclaimed_δ*, the determination of which tasks can use *reclaimed_δ* is made by the second step. In the second step, the next task in PPL_r, $\forall r$ such that P_r is idle, is examined to decide whether it can be immediately dispatched, i.e., whether it can use the *reclaimable* time. Figure 4.3 and 4.4 present the second step for each of the Basic Reclaiming and the Reclaiming with Early Start algorithms, respectively.

4.2.1 Calculation of Reclaimable Time

Step1 in Figure 4.2 calculates the amount of *reclaimable* time upon each task completion. This calculation is carried out only if the finishing task executes in less

```
Step1 ($T_{q_i}$, $reclaimed\_\delta$, $PL$, $PPL_q$);
    1.  REMOVE($T_{q_i}$, $PL$, $PPL_q$);
    2.  $T_f \leftarrow$ the first task in the current $PL$;
    3.  if ($current\_time < (ft_{q_i} - reclaimed\_\delta)$)
    4.     then
    5.     {
    6.           $reclaimable\_\delta = st_f - (current\_time)$;
    7.           if $reclaimable\_\delta > 0$
    8.              then $reclaimed\_\delta \leftarrow reclaimable\_\delta$;
    9.           end if
    10.    }
    11. end if
end Step1
```

Figure 4.2 Step 1: Calculation of *reclaimed_δ*.

than its worst case execution time (since otherwise the value of *reclaimed_δ* stays the same.).

The procedure for this calculation in Figure 4.2 is as follows:

- A task scheduled on processor q is not removed from the PL and PPL_q until it finishes execution. This restriction is required to ensure a consistent view of the amount of time reclaimable. Line 1 accomplishes this task removal.

- As mentioned above, we only need to update *reclaimed_δ* if the finishing task executes in less than its worst case execution time. The condition in Line 3 does this checking. Since both resource reclaiming algorithms will always start every task's execution at least by its scheduled start time minus the current value of *reclaimed_δ*, if T_{qi} finished before time $(ft_{q_i} - reclaimed\_\delta)$, it must have executed in less than its worst case execution time.

- Potential idle time on all processors and resources is identified by computing the function $reclaimed\_\delta = st_f - current\_time$ (lines 6 to 8 in **Step1**); where st_f is the scheduled start time of the current first task in the PL.

- Since the PL imposes a total ordering on the guaranteed tasks, st_f must be the minimum scheduled start time among all tasks in the schedule, including the task(s) still in execution. Any *positive* value of $reclaimed\_\delta$ indicates the length of the idle period resulting from tasks finishing early. Since a task is removed from the schedule only upon its completion (line 1 in Figure 4.2), $reclaimable\_\delta$ could have a *negative* value (if the first task in the PL is still in execution) and, in this case, $reclaimed\_\delta$ retains its original value.

For example, let us examine Figure 4.6. At time 125 when task T_1 completes execution, the current first task in the PL is T_3 which is still in execution, and so $reclaimable\_\delta = 0 - st_3 = 0 - 125 = -125$ since $st_3 = 0$ (refer to Table 3.1 and Figure 3.1 for the scheduled start times and scheduled finish times of the tasks.). On the other hand, at time 150 when T_3 finishes execution, T_4 becomes the first task in the PL, and so $reclaimable\_\delta = st_4 - 150 = 175 - 150 = 25$.

4.2.2 A Resource Reclaiming Algorithm without Passing

Figure 4.3 is the pseudo code for the Basic Reclaiming algorithm. This algorithm does not allow *passing*, and only starts the execution of a task earlier than its scheduled start time by as much as $reclaimed\_\delta$, i.e., the amount of time by which the entire remaining feasible schedule can be shifted forward.

The Basic Reclaiming algorithm in Figure 4.3 works as follows:

- We immediately start the execution of the first task T_{r_f} on processor P_r only if the task is the current first task in the PL (i.e., it is the next task in the total order of tasks), or if it has the same st (scheduled start time) as the current first task (line 4 and 5 in Figure 4.3).

```
**Step2.BASIC** ($q$, $reclaimed\_\delta$, $original\_reclaimed\_\delta$, $PL$, $PPL_q$);
1.  **if** $reclaimed\_\delta > original\_reclaimed\_\delta$
2.      **then**
3.          **for** all idle processors $r$ **do**
4.              **if** $(T_{r_f} == T_f)$ **or** $(st_{r_f} == st_f)$
5.                  **then** startexecution($T_{r_f}$);
6.                  **else**
7.                  {
8.                      $nst_{r_f} = st_{r_f} - reclaimed\_\delta$;
9.                      pend($T_{r_f}$,$nst_{r_f}$);
10.                 }
11.             **end if**
12.     **else**
13.     {
14.         **if** $(st_{qf} - reclaimed\_\delta) == current\_time$
15.             **then** startexecution($T_{r_f}$);
16.             **else**
17.             {
18.                 $nst_{q_f} = st_{q_f} - reclaimed\_\delta$;
19.                 pend($T_{q_f}$,$nst_{q_f}$);
20.             }
21.     }
22. **end if**
**end Step2.BASIC**
```

Figure 4.3 Step 2 of the Basic Reclaiming Algorithm.

- Otherwise we compute a function nst_{r_f} for T_{r_f} to decide the *new start time* (vs. the scheduled start time given in the schedule) for it, taking into consideration the reclaimed time accumulated up to now, i.e., the value of $reclaimed\_\delta$. This function is $nst_{r_f} = st_{r_f} - reclaimed\_\delta$, where st_{r_f} is the original scheduled start time of task T_{r_f}.

- The pend() function takes as its arguments a task id and a time value. This function wakes up the dispatcher process to start the execution of the task with the given task id at the specified future time.

- With this pend() function, processor r will idle until (1) either the calculated nst_{r_f} has arrived, or (2) some other task finishes early and *reclaimed_δ* is incremented. In the former case, task T_{r_f} will be dispatched at time nst_{r_f}. In the latter case, **Step2.BASIC** will be invoked again (see Figure 4.1).

- Because Basic Reclaiming does not allow any *passing*, if there is no increase in the value of *reclaimed_δ* when a task completes, then the decision made regarding the tasks' new start times for those tasks whose processors are still pending to start them from the last invocation of the resource reclaiming algorithm is still valid. Only the next task in the same PPL as the finished task needs to be examined. Lines 14 to 19 in Step2.BASIC accomplish this for task T_{q_f} on processor q.

4.2.3 A Resource Reclaiming Algorithm with Partial Passing

Notice that the Basic Reclaiming algorithm will start a task early by an amount of time equal to *reclaimed_δ* which is the length of time that all the processors can reclaim. The Reclaiming with Early Start algorithm eases this requirement. It allows a task T_{r_f}, the first task in PPL_r, to start as long as no *total-passing* occurs. More precisely, Reclaiming with Early Start works as follows:

> T_{r_f} can start if for $1 \leq v \leq m$ and $v \neq r$, T_{v_f} is either in $T_{\simeq r_f}$ or in $T_{>r_f}$. Let us define that a task T_{r_f} is being *early started* if $st'_{r_f} < st_{r_f} - reclaimed\_\delta$. In Reclaiming with Early Start, we first compute (lines 6 to 11) a Boolean function $can\_start\_early = st_{r_f} < ft_{v_f}$, $\forall v$ such that $v \neq r$ and $1 \leq v \leq m$, where st_{r_f} is the scheduled start time of the first task on processor r and ft_{v_f}

```
**Step2.EARLYSTART** (r, $reclaimed\_\delta$, $PL$, $PPL_1$, ... , $PPL_m$);
1.  $can\_start\_early \leftarrow$ true;
2.  **for** each idle processor $r$ **do** {
3.      **if** $st_{r_f} \neq st_f$
4.        **then**
5.        {
6.              $v \leftarrow 0$;
7.              **while** ($can\_start\_early$ **and** $v < m$) **do**
8.              {
9.                    $v \leftarrow v + 1$;
10.                   **if** ($v \neq r$) **and** ($st_{r_f} > ft_{v_f}$)
11.                     **then** $can\_start\_early \leftarrow$ false;
12.                   **end if**
13.             }
14.       }
15.     **endif**
16.     **if** $can\_start\_early$
17.       **then** startexecution($T_{r_f}$);
18.       **else**
19.       {
20.             $nst_{r_f} = st_{r_f} - reclaimed\_\delta$;
21.             pend($T_{r_f}$,$nst_{r_f}$);
22.       }
23.     **end if**
24. }
**end Step2.EARLYSTART**
```

Figure 4.4 Step 2 of the Early Start Algorithm.

is the scheduled finish time of the first task on processor v. This function identifies parallelism between the first task on processor r and the first tasks on all other processors by checking to see whether the first tasks on all other processors are in $T_{\simeq r_f}$. That is, for any two tasks T_{r_f} and T_{v_f}, if $st_{r_f} < ft_{v_f}$, then $T_{r_f} \in T_{\simeq v_f}$, or $T_{r_f} \in T_{<v_f}$. The task T_{r_f} will be dispatched if *can_start_early* is true. Otherwise, nst_{r_f} is computed and pend() is invoked for task T_{r_f} as in **Step2.BASIC**.

As mentioned earlier in this chapter, one must be careful about violating resource constraints when using the the PL and PPL's as a basis for resource reclaiming. Intuitively, the Reclaiming with Early Start algorithm does not violate resource constraints, since a task T_{r_f} will not be started early if this early start would result in T_{r_f} executing concurrently with another task T_{vf} that had not been originally scheduled to execute concurrently with T_{r_f}. Thus, we are using more knowledge to perform local optimization than in the Basic Reclaiming algorithm. One can expect that the Reclaiming with Early Start algorithm will perform better than the Basic Reclaiming algorithm because it makes better use of the reclaimed time.

4.3 Properties of the Resource Reclaiming Algorithms

The two resource reclaiming algorithms presented above guarantee that run time anomalies as shown in Chapter 3 will not occur. In this section we shall analyze the complexity of the two algorithms, and prove their correctness. We also illustrate some interesting properties and behavior of the two resource reclaiming algorithms through an example.

4.3.1 Complexity Analysis of the Algorithms

In this section, we show that the complexities of the two resource reclaiming algorithms are bounded and only depend on the number of processors in the system, rather than the number of tasks or resources. In the following analysis, we use C_i to represent the cost of functions, or operations that are constants, $\mathcal{T}()$ to denote actual run time cost, and $O()$ the worst case computational complexity.

Step1:

The computational complexity of the calculation of *reclaimed_δ* in Figure 4.2 is $O(C_1)$, where C_1 is some constant representing the sum of the cost of the conditional and assignment operations in lines 3 through 9.

Step2.BASIC:

The actual run time cost of Step2.BASIC at each invocation depends on the value of *reclaimed_δ* from the most recent update in Step1. If *reclaimed_δ* has not been incremented, only lines 12 through 21 are executed, and thus the cost is $\mathcal{T}(C_2)$, where C_2 is the sum of the costs for the operations of line 1 and lines 12 through 21. On the other hand, if *reclaimed_δ* has been incremented, lines 4 through 11 must be executed for all currently pending processors. Let C_3 be the sum of the costs for the operations of lines 4 through 11. In this case, the cost is $\mathcal{T}(mC_3)$.

Thus the worst case complexity of the Basic Reclaiming algorithm is: $O(C_1) + max(\mathcal{T}(C_2), \mathcal{T}(mC_3)) = O(m)$, where m is the number of processors.

<u>Step2.EARLYSTART:</u>

Step2.EARLYSTART in Figure 4.4 is always executed for all currently pending processors as indicated by the **for** loop. The function that identifies the parallelism among the first tasks on all processors (lines 7 through 13) has a complexity of $O(m)$. Thus the Early Start algorithm has a worst case complexity of $O(C_1) + O(m^2) = O(m^2)$. Note that the *best case* run time cost of Step2.EARLYSTART is $\mathcal{T}(m)$. This best case occurs in two situations: (1) when there are no currently pending processors except the processor that just completed a task execution, and (2) when only one task can start execution and the rest of the pending processors *fail* the *can_start_early* test the very first time the **while** loop on line 7 in Figure 4.4 is entered.

Since the complexities of the two resource reclaiming algorithms only depend on the number of processors, which can be assumed to be constant at run time, the costs of the algorithms are *bounded*.

4.3.2 Correctness

In the following, we shall prove that the two resource reclaiming algorithms presented in this section are *correct*, that is, they will not cause the type of timing anomalies shown in Chapter 3.

Observation 2: A *time translation* of x time units of a feasible schedule is an operation that subtracts x time units from all the scheduled start times and scheduled finish times of the tasks in the feasible schedule. *A feasible schedule remains feasible under time translation.*

Theorem 5: Given a feasible multiprocessor schedule S with resource and processor constraints, the Basic Reclaiming Algorithm will produce a correct post-run schedule.

PROOF By Lemma 2, we only need to prove that all tasks start *on-time* in the post-run schedule produced by the Basic Reclaiming Algorithm.

By Definition 12, if tasks are dispatched according to their *st* in the feasible schedule, they all start *on-time*. Note that the value of *reclaimed_δ* in **Step1** reflects the reclaimed time units on all resources and processors. Therefore, for $reclaimed\_\delta > 0$, we can perform a time translation of *reclaimed_δ* time units on the portion of the feasible schedule remaining to be dispatched. Since the feasible schedule remains feasible under time translation, and since **Step2.BASIC** dispatches every task at $st'_i = st_i - reclaimed\_\delta$, it follows that the tasks in the post-run schedule produced by the Basic Reclaiming Algorithm must have been started *on-time*.

Q.E.D.

Theorem 6: Given a feasible multiprocessor schedule S with resource and processor constraints, the post-run schedule produced by the Reclaiming with Early Start Algorithm is correct.

PROOF We shall prove that *total-passing* does not occur when Reclaiming with Early Start is used. Then by Lemma 1, we know that all tasks start *on time*.

We prove this by contradiction. Consider a task T_j to be dispatched in **Step2.EARLYSTART**. Suppose $\exists\ T_i$ such that T_j were dispatched at some time $st'_j < st_i$ while $st_j > ft_i$. This implies that T_j *totally passed* T_i. But this is impossible; because if $st_j > ft_i$, *can_start_early* would have become false in line 11 of **Step2.EARLYSTART**, and hence T_j would not have been dispatched.

Q.E.D.

4.3.3 Discussion Through an Example

Assume we have the same feasible schedule in Figure 3.1 for the set of tasks defined in Table 3.1. The post-run schedule produced by the Basic Reclaiming Algorithm is shown in Figure 4.6 and the post-run schedule produced by the Reclaiming with Early Start Algorithm is shown in Figure 4.8. We show the values of *reclaimed_δ* at the time of each task completion in Figures 4.5 and 4.7 for the two algorithms respectively[1]. Figure 3.2 is the post-run schedule when no resource reclaiming is done. Thus from Figures 3.2, 4.6, and 4.8, one can see the effects of resource reclaiming.

Note that once the new value of *reclaimed_δ* is determined in **Step1**, *every* task T_i in the rest of the schedule can in fact be started *reclaimed_δ* time units earlier than its st_i, e.g., at time 150 when T_3 completes execution, T_4 can start execution (see Figures 4.6 and 4.8). This is equivalent to a time translation of *reclaimed_δ* units of time on the remaining feasible schedule, i.e., the st_i and ft_i of every task T_i in the remaining feasible schedule can be translated to $st_i - reclaimed\_\delta$ and $ft_i - reclaimed\_\delta$. However, we do not *explicitly* carry out this time translation in the remaining feasible schedule because we will incur a time complexity of $O(n)$ to modify the st_i and ft_i of each task, thus violating our *boundedness* premise.

[1]Note that although there is no task completion at time 300 in Figure 4.8, we include the value of *reclaimed_δ* in Table 4.7 for comparison purposes.

time	0	125	150	175	250	300	425	450
reclaimed_δ	0	0	25	25	25	50	50	50

Figure 4.5 The values of *reclaim_δ* at each task completion when the Basic Reclaiming Algorithm is used.

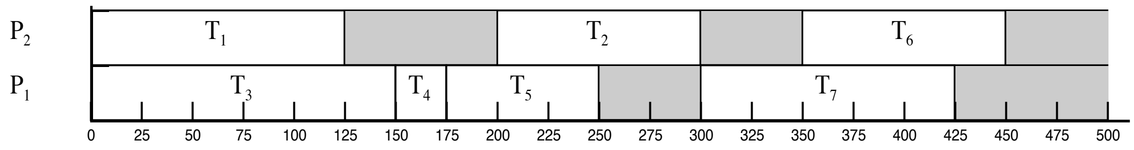


Figure 4.6 The *post-run* schedule S' produced by the Basic Reclaiming Algorithm.

time	0	125	150	175	250	275	300	375
reclaimed_δ	0	0	25	25	25	25	25	125

Figure 4.7 The values of *reclaimed_δ* at each task completion when *early start* is allowed.

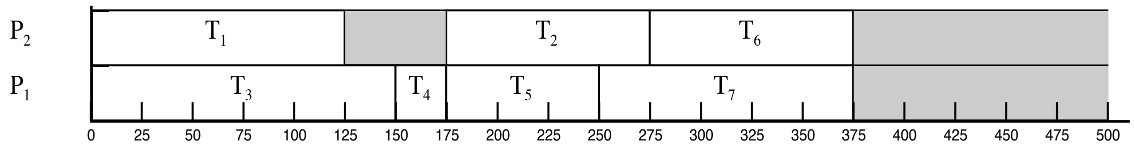


Figure 4.8 The *post-run* schedule S' produced by the Reclaiming with Early Start Algorithm.

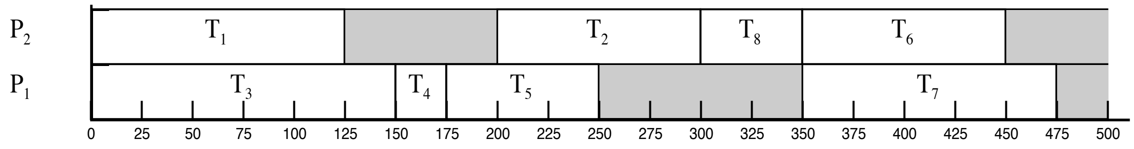


Figure 4.9 The *post-run* schedule S' produced by the Basic Reclaiming with the addition of T_8.

From the description of the algorithms, it seems obvious that Reclaiming with Early Start should be more effective than Basic Reclaiming. However, there are two interesting aspects of the Reclaiming with Early Start Algorithm that are not very intuitive.

- First, Reclaiming with Early Start does not necessarily accumulate a larger value of *reclaimed_δ* in the *short term.* For example, compare the values of *reclaimed_δ* at time 300 in Figures 4.5 and 4.7. The value of *reclaimed_δ* from using Basic Reclaiming is larger than from using Reclaiming with Early Start at time 300, even though at time 375, the opposite is true. This is because *reclaimed_δ* reflects the time reclaimed on *all* processors and resources. In general Reclaiming with Early Start keeps the processor and resource utilization higher than Basic Reclaiming does. So when using Reclaiming with Early Start, *reclaimable_δ* might be found to be positive *less* frequently in **Step1**. But in the long run, such as by time 375, Reclaiming with Early Start can have a large value of *reclaimed_δ*.

- Second, since we are dealing with dynamic real-time systems, tasks can arrive at any time. Whether a task can be feasibly scheduled depends very much on the particular time the task arrives at the system, i.e., the current system state including the number of tasks and their worst case requirements, and which tasks are already in execution. Therefore, even though Reclaiming with Early Start can *eventually* have a larger value of *reclaimed_δ*, it does not outperform the Basic Reclaiming algorithm with respect to guaranteeing dynamic task arrivals at *every* task arrival instance. This is because starting the execution of a task as early as possible is not necessarily always the best choice in a system with nonpreemptive scheduling and dynamic arrivals. For example, assume we have the same feasible schedule as in Figure 3.1 and, for the ease of explanation, let us assume scheduling occurs instantaneously. If a task T_8 arrives at time 300 with $c_8 = c'_8 = 50$, $d_8 = 375$, and $R_8^1 = exclusive$ (i.e., having a resource conflict with T_7), a system using the Basic Reclaiming algorithm will be able to feasibly schedule T_8 as shown in Figure 4.9, while a system using the Reclaiming with Early Start will not be able to schedule T_8 (since T_6 and T_7 are already in execution). Thus we need to examine the effectiveness of Reclaiming with Early Start and Basic Reclaiming with respect to dynamic task arrivals through experimental studies. This is done in Chapter 6.

4.4 Applicability of the Resource Reclaiming Algorithms

Here we discuss the applicability of the two resource reclaiming algorithms to task and multiprocessor systems with various characteristics.

4.4.1 Centralized Memory vs. Distributed Memory Multiprocessor Models

There are two types of multiprocessor scheduling models. In one type, a global memory is assumed, enabling the execution of a task on any of the processors, with access time to a physical memory location the same for all the processors. In the other type, physical memory is divided into modules with some placed near each processor (which allows faster access time to that memory), and a task is allocated to one of the processors, and thus can only be executed on a particular processor at run time. The former can only model *identical* multiprocessor systems, while the latter can model both identical and heterogeneous multiprocessors. In either type of multiprocessor system, tasks executing on different processors can share the use of resources, such as shared data structures. Thus the scheduling algorithm used in either model must consider not only the timing constraints of tasks, but also the resource constraints. Both of our resource reclaiming algorithms preserve the processor assignment a multiprocessor scheduler makes in constructing a feasible schedule; therefore they are applicable for both types of multiprocessor scheduling models.

4.4.2 Precedence Constraints among Tasks

In this Chapter, we have assumed that tasks are mutually independent. There are many applications in which tasks are related by *precedence constraints.* Precedence constraints specify the partial ordering among tasks such that a task can start execution only when all of its predecessors have completed execution. In a nonpreemptive scheduling scheme where the schedule construction is explicit and the precedence constraints are *scheduled away* (as we discussed in Chapter 2), the correctness criterion defined in Chapter 3 for resource constrained multiprocessor resource reclaiming also applies to precedence constrained task schedules. Since neither of the resource reclaiming algorithms presented in this chapter allows *total passing*, they are both directly applicable for task systems with precedence constraints. If tasks have precedence constraints in a feasible schedule, the resource reclaiming algorithms will never violate these precedence constraints.

4.4.3 Tasks with Explicit Start Time Constraints

Some systems may have tasks that cannot be started until after some specific time, called a *ready time.* For example, periodic tasks cannot be started until the beginning of their periods. In such systems, a task with a ready time may have been placed in the feasible schedule, but it cannot be moved forward to pass its ready time in the schedule. In this case, our resource reclaiming algorithms needs to be modified to take into consideration a task's ready time. In Step 2 of each of the algorithms, we need to consider the ready time of a task when we try to start a task. Specifically:

- In lines 4 and 14 in Figure 4.3, and line 16 in Figure 4.4, the following condition should be added:
  - **if** $current\_time \geq ready\_time(T_{r_f})$.
- At line 8 and line 18 in Figure 4.3, and line 20 in Figure 4.4 we need to modify the calculation of the new start time of a task to the following:
  - $nst_{r_f} = max(st_{r_f} - reclaimed\_\delta, ready\_time(T_{r_f}))$.

4.4.4 Other Types of Tasks

In addition to dynamic hard real-time tasks, a system may have (1) monotone tasks [84], (2) dual-copy fault-tolerant tasks [15], and (3) non-real-time tasks. Real-time systems with these types of tasks can all benefit from resource reclaiming. Instead of using the reclaimed time $reclaimed\_\delta$ for the tasks that have already been *guaranteed* in the feasible schedule, a system can use the time to (1) execute the optional part of a monotone task, (2) increase the time assigned to the primary copy of a dual-copy fault-tolerant task in a feasible schedule, or (3) preemptively execute non-real-time background tasks.

4.5 Summary

In this Chapter, we presented two resource reclaiming algorithms, Basic Reclaiming and Reclaiming with Early Start. The algorithms utilize the information given in a feasible schedule to (1) identify reclaimable time intervals, and (2) reason about the allowable parallelism and potential resource conflicts among the tasks. We proved the correctness of both algorithms with respect to the avoidance of timing anomalies. The complexities of the algorithms are shown to be *bounded* and a function of the number of processors in a multiprocessor system. The applicability of the algorithms to tasks and multiprocessor systems with various characteristics was also discussed.

As the goal of this dissertation is to construct integrated solutions for dynamic real-time systems, we will next present a concurrent, on-line, bounded-time solution to scheduling and dispatching in a multiprocessor real-time system. We demonstrate the importance of being *time conscientious* — being able to operate without unintentional impingement on the time already guaranteed to other real-time activities in the system. As part of the evaluation of the resource reclaiming algorithms presented in this chapter, a predictable integration of the resource reclaiming algorithms with the scheduling and dispatching processes is also presented.

CHAPTER 5

Scheduling and Dispatching with Resource Reclaiming: A Concurrent, On-line, Bounded-time Implementation

In many real-time applications, the system is required to execute tasks in response to external events and signals. Dynamic feasibility checking and scheduling of real-time tasks are required for such real-time systems. The provision of the *time conscientious* property is more difficult in these systems because the guaranteed application activities must be able to continue as the system schedules incoming task arrivals. In this chapter, we describe the issues and our solutions for the construction of concurrently executing on-line scheduler, dispatchers, and resource reclaiming.

After introducing the important issues and considerations in constructing concurrent, on-line, bounded time scheduling and dispatching processes, in Section 5.2, we motivate our concurrent implementation of dispatching and resource reclaiming by demonstrating quantitatively the advantage of a concurrent implementation vs. a centralized implementation. A prerequisite to support concurrent processes on a multiprocessor with shared data structures is the existence of appropriate synchronization mechanisms. We present a multiprocessor synchronization algorithm with bounded waiting in Section 5.2. Then the potential timing problems that may occur at run time are discussed with respect to (1) concurrent execution of the scheduler and the dispatcher processes, and (2) multiple invocations of the scheduler during the course of the execution of a feasible schedule. For both problems, Section 5.3 presents solutions that possess the property of being *time conscientious.* In Section

5.4, we present the issues in the integration of the resource reclaiming algorithms with the concurrent scheduler and dispatchers. Section 5.5 summarizes this chapter.

5.1 Introduction

In dynamic real-time systems, tasks can arrive at the system at unpredictable times. In a *guarantee-oriented* system, we need to schedule tasks as they arrive. On a distributed memory multiprocessor system where the real-time tasks have resource constraints, the tasks must be guaranteed in an integrated fashion in order to account for their resource conflicts. This can be accomplished by scheduling all the tasks in an integrated fashion — irrespective of which processor on which they are supposed to execute. For such a dynamic multiprocessor real-time system, there is more than one choice for the implementation of the scheduler and the dispatcher processes:

(1) Place both the scheduler and a centralized dispatcher on one and the same processor.

(2) Place the scheduler and a centralized dispatcher on two different processors.

(3) Place the scheduler and a set of concurrent dispatchers on individual processors.

For each of the above choices, we have a choice of either executing the scheduler on the same processor(s) with application tasks, or designating a system processor to handle all the system operations, such as scheduling, dispatching, and communication. To ensure that the system possesses the *time-conscientious* and *time-conserving* properties, the two major concerns here are: *predictability*, and *efficiency* (i.e., low and bounded costs). If the scheduler is on the same processor with application tasks, the scheduler needs to be either *preemptable* or *periodic*, because guaranteed tasks must be dispatched by their assigned scheduled start time in a feasible schedule in order not to miss their deadlines. To be preemptable, the following problems must be addressed:

- *The execution time of the scheduler must be accounted for:* The scheduler can only be executed during idle periods in a feasible schedule. The preemption cost of the scheduler needs to be added onto the cost of the dispatcher since the dispatcher potentially needs to preempt the scheduler every time before it dispatches a task. This cost must be bounded. If the scheduler is not preemptable, the worst case execution time of the scheduler needs be included in the cost of the dispatcher, so that if the scheduler is invoked at the time the dispatcher needs to dispatch a task, the dispatcher can wait until the completion of the scheduler. As we have mentioned in Chapter 3, real-time scheduling algorithms for *guarantee-oriented* systems usually have run time costs that depend on the number of tasks, and bounding the worst case cost of a scheduler results in large variance of the dispatcher's execution time. This is not acceptable for our *time-conservation* requirement.

- *The time for the* feasibility-checking *of each task arrival needs to be considered:* If the scheduler is preemptable, it is difficult to predict how many times the scheduler will be preempted before the feasibility-checking is completed. Thus a preemptable scheduler may impose a large variance on the feasibility-checking response time.

The same problems must also be addressed if the scheduler is a periodic process. In this case, the scheduler executes for a maximum fixed amount of time, i.e., for the execution time allocated to it, every period. In this case, we must derive the worst case execution time of the scheduler with respect to some maximum number of tasks that will ever be considered at each periodic invocation of the scheduler. Assuming that a feasibility check can be completed within a single execution, the response time of the *feasibility-checking* of each task arrival will vary from as little as the execution time of the scheduler to as much as twice the scheduler's period.

In order to maximize the potential parallelism provided by multiprocessor systems, our approach supports the concurrent execution of application tasks and the scheduling algorithm. This is accomplished by using one processor on a multiprocessor node as the system processor to offload task scheduling and other operating system overhead, therefore eliminating many possibilities of interference with application tasks, while using the remaining processors to execute guaranteed application tasks. The scheduler on the system processor is responsible for dynamically producing a feasible schedule for the multiprocessor as tasks arrive. Feasibility checking employs the heuristic scheduling algorithm proposed in [72], which has a complexity of $O(n)$. There is a *dispatcher process* on each application processor. Keeping the issues we have presented in this section in mind, in the rest of the chapter, we present a systematic construction of this concurrent on-line implementation.

5.2 Computation Time Comparison of Centralized vs. Concurrent Implementation

Given a multiprocessor feasible schedule, two approaches can be taken to implement the dispatching process with resource reclaiming on a multiprocessor system — *centralized* and *concurrent.* In a centralized scheme, the process can be implemented on a single processor. In a concurrent scheme, each processor performs its own dispatching and reclaiming, therefore, all the processors in the multiprocessor system concurrently dispatch application tasks and reclaim unused time as tasks complete execution. The parallelism provided by a multiprocessor can be more effectively exploited with a concurrent implementation.

Table 5.1 compares the computation time of the centralized and the concurrent implementation under worst case assumptions. The worst case occurs when all m processors complete their respective current task executions to perform resource

Table 5.1 Computation Time Comparison for Centralized vs. Concurrent Implementation

	centralized	**concurrent**
Step1	$m * C_{Step1}$	$C_{Step1} + (m-1)(\epsilon + \tau)$
Step2.BASIC	$m * C_{Basic}$	$C_{Basic} + 5\tau$
Step2.EARLYSTART	$m * C_{Early}$	$C_{Early} + (m+3)\tau$

reclamation and dispatching at the same time, and all shared variables are accessed simultaneously by all processors. The effect of this worst case is to serialize certain portions of the Basic and Early Start algorithms. As illustrated in Table 5.1, substantially more code is serialized in the centralized than in the concurrent implementation. We discuss the computation of the results in Table 5.1 further below.

The pseudo code of **Step2.BASIC** in Figure 4.3, and **Step2.EARLYSTART** in Figure 4.4 are presented for a *centralized* dispatcher. For the concurrent implementation, the **for** loops in both algorithms can be eliminated, since all processors execute concurrently. In addition, the following two modifications must be made for the resource reclaiming algorithms:

- **Step2.BASIC**: If a task completion results in an increment of the value of *reclaimed_δ*, this new value is broadcast to all processors so that all *pending* processors may update the *new start times* they are pending on (see Section 4.2.2 for the definition of the function *pend.*). To ensure the consistency of the value of *reclaimed_δ* that is seen by all the dispatchers in face of simultaneous broadcast, the actual *broadcast* is implemented as a trigger of interrupts on all the processors without any accompanying data. The broadcast recipients perform a read of *reclaimed_δ* after receiving the broadcast notification.

- **Step2.EARLYSTART**: Whenever a task completion occurs, all idle processors are notified with a *broadcast*. Since Reclaiming with Early Start allows

```
Whenever a task $T_{q_i}$ completes execution on a processor $q$, do
{
    $original\_reclaimed\_\delta = reclaimed\_\delta$;
    Step1($T_{q_i}$, $reclaimed\_\delta$, $PL$, $PPL_q$);
    **switch** ($algorithm\_choice$)
            **case** BASIC_RECLAIMING:
                Step2.BASIC($q$, $reclaimed\_\delta$, $original\_reclaimed\_\delta$,
                            $PL$, $PPL_1$, ... , $PPL_m$);
            **case** EARLY_START:
                broadcast($task\_completion$);
                Step2.EARLYSTART ($r$, $reclaimed\_\delta$,
                                    $PL$, $PPL_1$, ... , $PPL_m$);
}
```

Figure 5.1 The resource reclaiming algorithm for concurrent implementation.

partial passing, each task completion may result in the *early start* of one or more task.

We show the pseudo code for the concurrent implementation of the resource reclaiming algorithms with these modifications in Figures 5.1, 5.2, and 5.3.

In the concurrent implementation, one must ensure consistency when concurrent conflicting access to shared data is possible. In our approach, we use critical sections protected by semaphores to enforce consistency. Recall that in our approach to on-line real-time algorithms, we must ensure a bounded worst case execution time. In Section 5.3, the details of the construction of semaphores which possess the

```
**Step2.BASIC** ($q$, $reclaimed\_\delta$, $original\_reclaimed\_\delta$, $PL$, $PPL_q$);
1.  if $reclaimed\_\delta > original\_reclaimed\_\delta$
2.      then broadcast($reclaimed\_\delta$);
3.  if ($T_{q_f} == T_f$)
    or ($st_{q_f} == st_f$)
    or ($st_{qf} - reclaimed\_\delta$) == $current\_time$
4.      then startexecution($T_{q_f}$);
5.      else
6.      {
7.          $nst_{q_f} = st_{q_f} - reclaimed\_\delta$;
8.          pend($T_{q_f}$,$nst_{q_f}$);
9.      }
10. end if
end Step2.BASIC
```

Figure 5.2 Step 2 of the Basic Reclaiming Algorithm for Concurrent Implementation.

property of *bounded waiting* are presented. The use of semaphores with this property allows us to perform worst case analysis of concurrent reclaiming algorithms.

The formulas in Table 5.1 are derived from the pseudo-code of the Basic and the Early Start algorithms. Code common to these algorithms (Step1) is in Section 4.2.1 Figure 4.2. Code specific to the Basic algorithm **Step2.BASIC** is in Figures 4.3 and Figures 5.2. Code specific to the Early Start algorithm **Step2.EARLYSTART** is in Figure 4.4. The notation and the derivation of the formulas in Table 5.1 are explained in the following.

Let

- C_{Step1} be the computation time of **Step1**,
- C_{Basic} the computation time of **Step2.BASIC** without the **for** loop, and
- C_{Early} the computation time of **Step2.EARLYSTART** without the **for** loop.

```
**Step2.EARLYSTART** (r, $reclaimed\_\delta$, $PL$, $PPL_1$, ... , $PPL_m$);
1.  $can\_start\_early \leftarrow$ true;
2.  **if** $st_{r_f} \neq st_f$
3.    **then**
4.    {
5.        $v \leftarrow 0$;
6.        **while** ($can\_start\_early$ **and** $v < m$) **do**
7.        {
8.            $v \leftarrow v + 1$;
9.            **if** ($v \neq r$) **and** ($st_{r_f} > ft_{v_f}$)
10.             **then** $can\_start\_early \leftarrow$ false;
11.           **end if**
12.       }
13.   }
14. **endif**
15. **if** $can\_start\_early$
16.   **then** startexecution($T_{r_f}$);
17.   **else**
18.   {
19.       $nst_{r_f} = st_{r_f} - reclaimed\_\delta$;
20.       pend($T_{r_f}$,$nst_{r_f}$);
21.   }
22. **end if**
**end Step2.EARLYSTART**
```

Figure 5.3 Step 2 of the Early Start Algorithm for the Concurrent Implementation.

In a concurrent implementation, lines 7–8 in **Step1** must be within a critical section to maintain the consistency of the value of *reclaimed_δ*. Let us define ϵ to be the computation time plus the lock-request and lock-release time of lines 7–8 in **Step1**[1]. Because it is a shared variable, the access to *reclaimed_δ* itself incurs additional cost

[1]The lock-request and lock-release time using the predictable multiprocessor synchronization mechanisms developed in [60] is $0.05ms$ in the worst case when the communication bus shared by four processors is fully saturated.

Table 5.2 Execution times (μs) of **Step2.BASIC** on a VMEbus based Motorola 68020 multiprocessor.

number of processors	1	2	3	4	5
V_{ns}	1				
C_{Basic}	20.5				
V_s		2.75	3.92	5.33	6.71
centralized		41	61.5	82	102.5
concurrent		29.25	35.1	42.15	49.05

in a concurrent implementation[2]. Let $\tau = V_s - V_{ns}$, where V_s is the worst case cost of accessing one shared variable, and V_{ns} the cost of accessing one non-shared variable. Thus τ represents the difference in terms of cost of accessing a variable between the concurrent and centralized implementation schemes. Thus the worst case cost of executing **Step1** is $C_{Step1} + (m-1)(\epsilon + \tau)$. This cost is worst case since it accounts for the cost of m processors sequentially accessing the critical section.

In a concurrent implementation, the first task in PL will be accessed by all the processors. Moreover, in a concurrent implementation of **Step2.EARLYSTART**, the first tasks in all the PPL's will be accessed by all the processors. However, no critical sections are necessary in the concurrent executions of **Step2.BASIC** and **Step2.EARLYSTART** since all the shared variable accesses in them are all read-accesses. So without locking, we assume *pessimistic* resource reclaiming, i.e., we may *miss* an update without affecting the correctness. Thus, using similar arguments as for the derivation of the worst case cost of **Step1** in Table 5.1, the expressions $C_{Basic} + 5\tau$ and $C_{Early} + (m+3)\tau$ model the 5 and $m+3$ shared variables accessed in **Step2.BASIC** and **Step2.EARLYSTART**, respectively.

[2]Time to access a variable located in remote memory over the shared bus can also be bounded (e.g., by imposing a round robin arbitration on the shared bus access).

Table 5.2 illustrates the advantage of the concurrent scheme as implemented of **Step2.BASIC** on a VMEbus based Motorola 68020 multiprocessor. V_{ns} is defined as the access time of on-board memory. C_{Basic} is $20.5\mu s$ excluding the *startexecution* operation[3]. The values of V_s were obtained from the worst case timings when two to five processors contending for the same remote memory location. The worst case execution times of **Step2.BASIC** on two to five processors for the centralized and concurrent implementation were derived from the formulas given in Table 5.1. It is evident from Table 5.2 that, for a reasonable number of processors in a shared bus multiprocessor system, the concurrent scheme is more efficient.

5.3 Multiprocessor Synchronization with Bounded Waiting

In a multiprocessor system in which tasks share information via shared resources, synchronization mechanisms must be available to coordinate processes [34]. On a multiprocessor, both the shared memory and the shared bus fall into the category of shared resources. In a real-time system, this synchronization must possess the bounded timing property in order to achieve overall predictability.

Although the mutual exclusion problem has been extensively studied in the non-real-time context, and many hardware and software solutions exist, real-time systems offer new challenges in dealing with the mutual exclusion issue. As we have pointed out in Chapter 1, among the most difficult operating system primitives to construct with the aim of achieving predictability are those which involve concurrent access to shared resources. For example, concurrent interaction between a scheduler and multiple dispatchers on a multiprocessor may require mutually exclusive access to shared data pertinent to a task schedule. This shared access to the scheduling data

[3]*startexecution* is the function that performs the actual loading of a task's context to start up the execution of a task, this should have the same execution time cost for both centralized and concurrent schemes.

will in turn require the access to the shared bus on a multiprocessor if the scheduler and the dispatchers reside on different processors. Operations for enforcing mutual exclusion operations such as P() and V(), if constructed in a bounded fashion, can provide the framework for other, higher level, bounded operating systems primitives. This boundedness forms a basis for the predictability of the entire system.

Conventional shared memory multiprocessors often support mutual exclusion in the form of atomic *read-modify-write* (RMW) instructions. Systems such as the Motorola MVME136-a, Sequent Symmetry, and the Ultracomputer [28, 61, 79] fall into this category. This support of an atomic RMW instruction is also often referred to as support for test-and-set. On a uniprocessor, test-and-set like hardware support enables the construction of more concise and more efficient solutions of synchronization primitives, while on a multiprocessor, atomic, test-and-set like operations make correct synchronization possible [67]. However, in the conventional use of these hardware implementations on a multiprocessor, one processor may encounter *starvation* when contending for a semaphore. This *unbounded waiting* does not meet the requirements of predictability for real-time systems.

Another problem with conventional shared memory multiprocessors' semaphore implementations is that of resource wastage [20]. The ubiquitous busy wait loop generates both bus traffic and consumes CPU resources. The bus traffic generated by the busy-wait can be mitigated by a scheme which busy-waits on a cache memory address [79]. The Sequent's approach allows each processor only *one* attempt (per semaphore change) to acquire the semaphore. If this fails, the processor will spin on the cache memory location. In [2], it was noted that this scheme can cause a cascading of cache invalidations, thus causing additional bus traffic. This resource wastage is detrimental for both non real-time and real-time systems. For real-time systems in which we desire *time consciousness*, *time conscientiousness*, and *time conservation*, what is needed is a semaphore implementation that has both the properties of *bounded waiting*, and efficient usage of system resources (such as the shared bus).

In order to construct an implementation which achieves these properties, solutions that integrate both hardware and software facilities are needed. First, bounded access to the shared bus must be ensured. This can be accomplished by requiring shared bus access with the *round robin* arbitration mechanism. Round-robin bus access is a necessary, but not sufficient condition for the construction of mutual exclusion protocols with bounded waiting. Mutual exclusion protocols using this mode are not guaranteed to be starvation free [60]. Obviously, system services with the possibility of starvation violate the requirements for predictability. We formally define bounded waiting as follows:

Definition 16: *Bounded waiting* is achieved if there is a constant k such that if a process is in its busy-wait loop, then that process will enter its critical region before any other process has entered its critical region more than k times [12].

We next present a *bounded waiting* solution with k=1.

5.3.1 A Semaphore Implementation with Bounded Waiting and Efficient Resource Usage

In this subsection, we present a resource efficient semaphore implementation that achieves bounded waiting. Our implementation is based on an algorithm by Burns ([12]). Burns' algorithm assumes bounded bus access, as would be provided with a round robin bus protocol. Burns' original solution provided bounded waiting through the use of a *global* wait queue, containing one bit per processor. Once a processor fails on the TAS (test-and-set) in its P() operation, it asserts an appropriate flag in a waiting array. When a release of the semaphore occurs in the V() operation, the next processor (in cyclic order) with its flag set in the waiting array is allowed to acquire the semaphore. This implementation achieves bounded waiting by imposing a cyclic ordering of waiting processors. Assume a maximum of m processors contending for the semaphore. Since the waiting array is scanned in cyclic order, (e.g., from 0, 1, ... , $m-1$ back to 0), if processor P_i is waiting (e.g., has entered P()), it will enter its critical section within at most $m-1$ turns.

We have extended Burns' algorithm to be resource efficient as follows. Traffic over the shared bus is reduced by, whenever possible, spinning on a secondary *local* semaphore instead of the *global* (central) semaphore as in Burns' original solution. The Extended Burns implementation of P() and V() is illustrated in Figure 5.4. In order to spin locally, a local semaphore (Lsem) is introduced. In the P() operation, once the TAS of the global semaphore (Gsem) fails, spinning will occur on local variables only (the wait queue, $Try[]$, and Lsem). Thus, the wait queue is *distributed*, with one bit per processor. In Figure 5.4, the integer i is a unique processor number between 0 and $m-1$. TAS(Gsem), an indivisible TAS (test-and-set) operation, returns *true* when the set is accomplished on the semaphore Gsem (the lock is acquired). The clear(Gsem) operation resets the Gsem. The broadcast_clear(Lsem) operation clears all local semaphores in a single atomic instruction (and this operation is necessary to overcome possible race conditions caused by processor speed difference). This primitive is used for efficiency — a loop which cleared all Lsems would suffice for correctness. To initialize the algorithm, the global semaphore Gsem, the local semaphore Lsem, and the wait queue $Try[]$ should all be cleared to zero.

It was demonstrated by Burns in [12] that, since a process in its P() operation cannot be skipped by any process which enters its critical section at a later point, a cyclic order of selecting processors is guaranteed. The extended algorithm maintains this feature. However, when dealing with both a primary and a secondary semaphore, one must be careful not to introduce the possibility of livelock into the

```
1:  P()
2:  {
3:  check:
4:        Lsem = true;
5:        if (TAS(Gsem)) return();
6:        Try[i] = true;
7:        while (Try[i] and Lsem) ;
8:        if (Try[i]) goto check;
9:  }
```

```
1:  V()
2:  {
3:        int j;
4:
5:        Try[i] = false;
6:        j = (i+1) mod m;
7:        while (!Try[j] and j != i)
8:             j = (j+1) mod m;
9:        if (j == i) {
10:            clear(Gsem);
11:            broadcast_clear(Lsem);
12:       }
13:       else Try[j] = false;
14: }
```

Figure 5.4 Extended Burns Solution with Bounded Waiting.

concurrent algorithm. In the following, we prove that the extended algorithm is livelock free.

Theorem 7: The Extended Burns' Solution is livelock free.

PROOF. As in Burns' original algorithm, it is clear that if P_j is waiting when P_i releases the semaphore, P_j will obtain it. A subtlety arises when analyzing a potential race condition, occurring in P() when one bit in $Try[]$ becomes true only after V() scans the entire waiting array. We must ensure that the semaphore is granted to some requesting process (therefore ensuring progress is made). This race condition can occur in two cases (We use P:i or V:i to mean line i in the P() or V() code of Figure 5.4.):

- CASE 1: Gsem is cleared (V:10) before the TAS (P:5) is executed. Then P_i acquires the semaphore since the TAS on (P:5) returns *true*.
- CASE 2: Gsem is not cleared before the TAS (P:5) is executed.

For example, P:5 is executed before V:10. Lsem is invariant until V:11 is executed, at which point the test of Lsem fails in P:7, so a branch is executed to P:3. At this point we know Gsem was cleared by V:10, thus the conditions for case 1 are true.

In either case, the semaphore is acquired by some processor.

Q.E.D.

5.4 Predictable Construction of Concurrent Scheduling and Dispatching

Having demonstrated the advantage of using concurrent implementation in Section 5.2, and the bounded semaphore implementation presented in Section 5.3, we proceed to construct the concurrent implementation of the scheduler and dispatchers. As stated in Section 5.1, in our multiprocessor implementation, a scheduler on the system processor is responsible for dynamically producing a feasible schedule for the multiprocessor as tasks arrive. There is also a *dispatcher process* on each application processor. In our concurrent implementation, effectively, whenever a task completes, this dispatcher process executes **Step1** and **Step2** of the reclaiming algorithm. Thus, reclaiming occurs *concurrently* on the application processors. This scheme is illustrated in Figure 5.5. The System Task Table is the current feasible schedule, organized as the PL and PPL lists.

If the deadline of a task can be met with respect to the task's worst case execution time, resource requirements, as well as the needs of other tasks that have already been feasibly scheduled by the system (i.e., those tasks that arrived before), then we say this newly arrived task is *guaranteed* by the system. One of the challenging issues of this approach is how to maintain the guarantees the system has made so far in face of dynamic task arrivals, i.e., how to maintain the *promises* the system has made to the tasks that have already been feasibly scheduled. The difficulties stem from the fact that (1) newly arrived tasks need to be scheduled (i.e.,

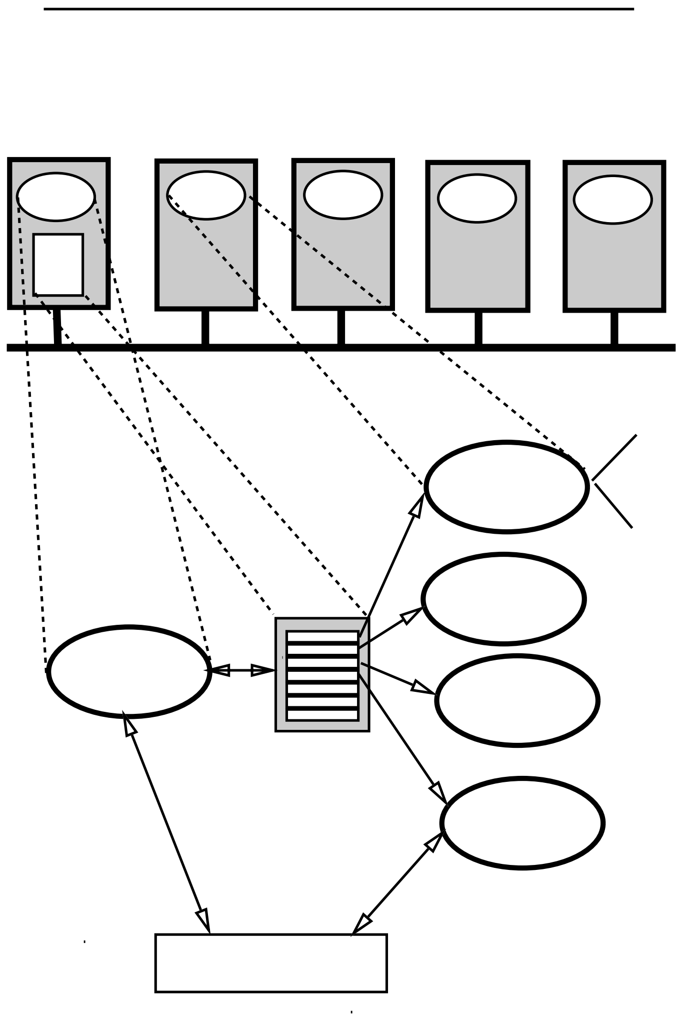


Figure 5.5 The scheduler and dispather interactions on a multiprocessor node.

examined by the scheduler to see if they can be guaranteed), (2) the runtime cost of a scheduler in general cannot be *bounded*, i.e., it takes time relative to the number of tasks to be considered in a schedule for a scheduler to construct a new feasible schedule including the newly arrived task, and (3) the newly arrived task needs to be examined together with the tasks already guaranteed in the existing feasible schedule in order to determine whether the newly arrived task can be guaranteed.

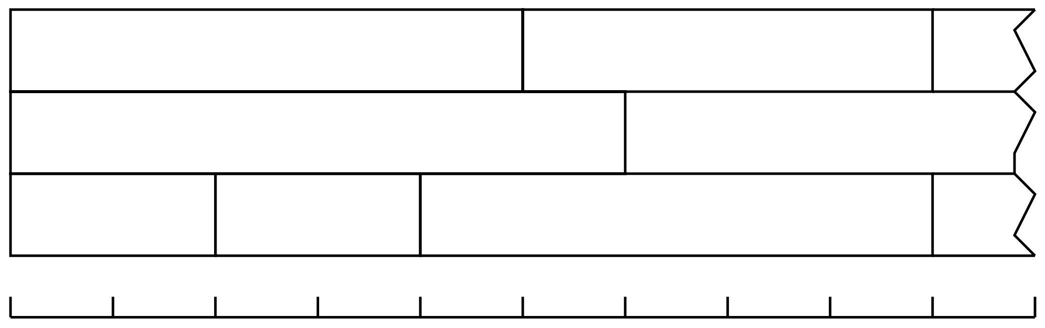


Figure 5.6 The current feasible schedule.

5.4.1 Potential Incorrectness

If the system uses the naive method of simply running the scheduler on the tasks that remain in the schedule (i.e. those whose scheduled start times have not arrived yet) and the newly arrived task, correct functioning of the system is not ensured. Suppose the current feasible schedule is the one in Figure 5.6, and current time is 100. In particular, two incorrect consequences may occur with respect to the following two cases:

- **case 1:** *The execution of the dispatchers continues as the scheduler does the rescheduling.*

  The incorrectness here is a result of the asynchronous concurrent access of the current feasible schedule by both the scheduler and the dispatchers, without imposing consistency constraints. Time is moving forward as the scheduler attempts to construct a new feasible schedule with respect to the newly arrived task and the remaining tasks in the current feasible schedule. The dispatchers, unaware of the scheduling activity going on, will continue to start the execution of tasks whose scheduled start times have arrived according to the current feasible schedule.

Suppose the scheduler executes for 30 time units. Currently tasks T_1, T_3, and T_5 are executing on processors $P1, P2$, and $P3$. And suppose the scheduling algorithm has placed the newly arrived task to start at time 120 on processor P3, i.e. in front of T_6. By the time the scheduler finishes its execution, time would be 130 already since we assumed the scheduler's runtime cost is 30 time units. By this time, task T_6 would have been dispatched on processor P3! Thus the scheduling result, i.e. the newly arrived task to be started at time 120, is incorrect.

- **case 2:** *The scheduler locks the current schedule, so dispatchers stop dispatching tasks for the duration of the scheduler's execution.*

  The incorrectness here is caused by the unpredicted stoppage in the dispatchers such that guaranteed deadlines may be missed.

  Now suppose, as in case 1, that the scheduler executes for 30 time units, and currently tasks T_1, T_3, and T_5 are executing on processors P1, P2, and P3. Moreover, suppose task T_6's deadline is at 140. Since the scheduler has locked the entire feasible schedule, the dispatchers have to wait until the scheduler to finish execution at 130 to start dispatching tasks again. However, by that time, task T_6 would have missed its deadline (assuming T_6's worst case execution time is 20), and thus the system would have violated it guarantee given to T_6.

5.4.2 A Solution

To construct the correct concurrent execution of the scheduler and dispatchers we have developed a strategy in which a *cutoff_line* is used to ensure the correctness of the previous guarantees the system has made in the feasible schedule. This strategy works as follows.

Since the cost of a scheduling algorithm depends on the number of tasks it is going to schedule, e.g. $O(n)$, or $O(n^2)$, we can calculate the run time worst

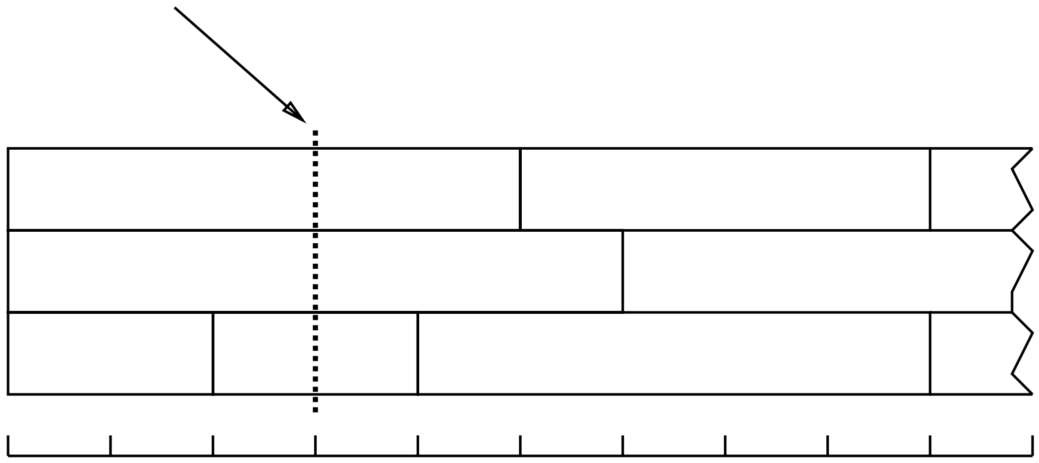


Figure 5.7 The current feasible schedule with the cutoff-line.

case execution time of the scheduler online at the scheduler invocation time. Specifically, we have conducted experiments to derive the worst case cost of the scheduler in terms of a constant portion and a per_task_cost portion. Upon the invocation of the scheduler, the worst case execution time of the scheduler (SC) for this particular invocation can be calculated as: SC = constant portion + n$*$per_task_cost, where n is the number of tasks to be rescheduled. This assumes the scheduling algorithm used is the one presented in [72]. Since we know the scheduler will definitely complete execution in SC time units, we reserve at least SC time units of execution time for guaranteed tasks for the dispatchers

```
Scheduler
    Whenever a task $T_i$ arrives, do
    {
        Calculate the run time cost $SC$ of the scheduling
        algorithm based on the number of tasks in
        the current $PL$ plus the new task arrival;

        $cutoff\_line = current\_time + SC$;

        $\mathcal{T}_{nr} \longleftarrow \{T_j | st_j - reclaim\_\delta < cutoff\_line\}$;

        Calculate the earliest available time
        of each resource and processor,
        based on the resource and processor requirements,
        and $ft_j$ of the tasks $T_j$ in $\mathcal{T}_{nr}$, and the value of $reclaim\_\delta$;

        $\mathcal{T}_r \longleftarrow \{T_j | st_j - reclaim\_\delta > cutoff\_line\}$;

        GUARANTEE($\mathcal{T}_r$, $T_i$);

        If the guarantee is successful
          then
          {
                append the new feasible schedule onto
                the remaining schedule at the $cutoff\_line$;
                broadcast(guarantee_successful);
          }
    }
```

Figure 5.8 Scheduling Dynamic Real-Time Tasks with Resource Reclaiming

to dispatch. This way the scheduler can run in parallel with the dispatchers, and tasks that have already been guaranteed will not miss their deadlines. In particular, the *cutoff_line* = current_time + SC. Tasks in the feasible schedule with scheduled start times less than the *cutoff_line* will not be rescheduled by the scheduler. This is illustrated in Figure 5.7.

This approach can be optimized. If too many tasks exist in the feasible schedule, the scheduler's cost can potentially become very high. To avoid this, we can set a value N as the maximum number of tasks the scheduler is ever going to handle per invocation, i.e., $max(SC)$ will be capped. At each invocation of the scheduler, if n, the number of tasks to be considered, is less than N, then SC is calculated as above. However, if $n > N$, then SC = constant portion + N*per_task_cost.

Figure 5.8 gives the algorithm we use to schedule dynamic task arrivals with resource reclaiming. The first action is to calculate the *cutoff_line*. Although the example in Figure 5.7 illustrates the *cutoff_line* logically as a *line*, in reality, the *cutoff_line* is a vector of time values. Since the tasks with scheduled start times less than the *cutoff_line* will not be rescheduled by the scheduler, the scheduler cannot construct a new schedule *right at* the *cutoff_line*, i.e., the processors and resources needed by these tasks should be reserved up to the maximum scheduled finish times of these tasks. Consequently, based on the *cutoff_line*, we must calculate a vector of the earliest available times of all the processors and resources. For example, suppose processors are the only resources the tasks in Figure 5.7 require. Although the *cutoff_line* is logically placed at time 130, the vector of earliest available times of the processors for processors P_1, P_2, P_3 is <150,160,140>.

After calculating the *cutoff_line*, $\mathcal{T}_{nr}$ contains the set of the tasks reserved for the dispatchers, thus not being rescheduled with the new task arrival. $\mathcal{T}_r$ contains the set of tasks to be rescheduled. The function GUARANTEE() accomplishes both *feasibility checking* and *schedule construction*. The feasibility checking employs the heuristic scheduling algorithm proposed in [72], which has a complexity of $O(n)$. The PL and PPL are established during the schedule construction. If the guarantee of the new task is successful, the following two operations are carried out:

- First, we append the new schedule onto the portion of the original schedule (the portion with the remaining tasks in $\mathcal{T}_{nr}$) at the *cutoff_line*. This *append* operation involves constructing the PL for the entire schedule, and the set of PPL's for each processor.

- Second, we *notify* (e.g., broadcast to) all the processors that a new schedule has been constructed. This notification is necessary to deal with the situation where a dispatcher on processor P_i could have been *pending* on the start time of a task before the invocation of the scheduler, and the start time of the task was later than the *cutoff_line*. Once a new schedule is constructed, the task that processor P_i is pending on may or may not be the first task on PPL_i anymore.

5.5 Predictable Resource Reclaiming with On-Line Scheduling

5.5.1 *Potential Inconsistency of reclaimed_δ*

When a new task arrives, its worst case execution time, deadline, and resource and processor requirements are assumed to be known. The system will try to guarantee the new task arrival together with all the tasks T_i, in the original feasible schedule, for which $st_i - reclaimed\_\delta \geq cutoff\_line$. With the knowledge of the value of *reclaimed_δ*, i.e., the amount of time that has been reclaimed on all resources and processors, those tasks T_i with $st_i - reclaimed\_\delta < cutoff\_line$ will finish at least *reclaimed_δ* time units earlier than their scheduled finish time ft_i. Thus, in calculating the earliest available time of resources and processors in trying to schedule the new task arrival in Figure 5.8, the scheduler takes the current value of *reclaimed_δ* into consideration.

If the new task arrival is guaranteed, the newly generated feasible schedule S_{new} must be *appended* to the original feasible schedule at the *cutoff_line*. Since the scheduler's cost SC is the scheduler's worst case execution time, it is very likely that there are still tasks in the original feasible schedule before the *cutoff_line* at the time when the scheduler finishes scheduling. Thus for the tasks that are in the section of the feasible schedule before the *cutoff_line*, the value of *reclaimed_δ* is valid. However, for the tasks that are in the section of the feasible schedule produced after the *cutoff_line*, the *reclaimed_δ* has already been taken into consideration in calculating the tasks' scheduled start times. Moreover, there can be more than one *cutoff_line* in a feasible schedule since more than one task can arrive, causing the scheduler to be invoked multiple times during the execution of tasks in a feasible schedule. We must develop a protocol to maintain the correct view of the value of *reclaimed_δ* between the tasks that are before, and that are after, each of the *cutoff-lines*, i.e., between any two portions of the current feasible schedule that have been constructed at two different scheduling instances. Otherwise, inconsistent usage of the value of *reclaimed_δ* may result in incorrect post-run schedules.

5.5.2 A Solution

To handle this problem, we have designed the following protocol.

- Each task T_i in the feasible schedule has a *reset_δ* field.

- The value of this field is zero for all tasks except for the task T_{f_k} which is the first task in the total ordering PL for S_{new_k}, where S_{new_k} is the section of the feasible schedule produced by the kth invocation of the scheduler. *reset_δ*(T_{f_k}) is set to be equal to the value of *reclaimed_δ* that has been assimilated by the kth invocation of the scheduler.

- As soon as T_{f_k} is dispatched, $reclaimed\_\delta = reclaimed\_\delta - reset\_\delta(T_{f_k})$.

This protocol ensures the correct view of the value of *reclaimed_δ* throughout a feasible schedule at any time. One may be tempted to adopt a conceptually simpler protocol, one that explicitly modifies the st_i and ft_i of all the tasks after the *cutoff_line* by the amount of $reclaimed\_\delta - reset\_\delta(T_{f_k})$ at the end of each scheduler's invocation. The *drawback* to this simpler protocol is that its run time cost is $O(n)$ and *reclaimed_δ* must be locked while this protocol is in progress to avoid race conditions between the scheduler and the dispatchers. This means that the dispatchers may have to wait for an amount of time that is $O(n)$, i.e., it is not *bounded*. Because of this, this simple protocol is not acceptable.

To ensure the correctness of the above protocol, none of the tasks in S_{new_k} can ever be dispatched using the original value of *reclaimed_δ*. Moreover, this correctness must be achieved with *bounded cost*. If tasks are dispatched strictly according to the scheduled start time order given in the PL, then the correctness is easily achieved with bounded cost since T_{f_k} is always dispatched *before* any other tasks in S_{new_k} is dispatched.

Observation 3: The Basic Reclaiming Algorithm dispatches tasks strictly according to the order given in the PL.

Although the correctness of the protocol can be ensured with *bounded cost* by the Basic Reclaiming Algorithm, this is not true if Reclaiming with Early Start is used. This is because Basic Reclaiming does not allow any *passing*, so if $st_i < st_j$, then $st'_i < st'_j$. On the other hand, since Reclaiming with Early Start allows *partial passing*, the order of the actual start times of tasks in the post-run schedule produced by Reclaiming with Early Start may not correspond to the order in the PL.

Observation 4: The Reclaiming with Early Start algorithm may dispatch a task T_i (with $st_i > st_{f_k}$) in S_{new_k} before it dispatches T_{f_k}.

If a task T_i in S_{new_k} is dispatched before T_{f_k} starts execution, i.e., before the value of *reclaimed_δ* has been reset, *total passing* may occur, thus an incorrect post-run schedule may be produced. This dispatching anomaly occurs because an incorrect value of *reclaimed_δ* is used for T_i — *reclaimed_δ* is used before the reset value has been applied. If the tasks in S_{new_k} use this value of *reclaimed_δ*, the Reclaiming with Early Start algorithm may see *false* parallelism among tasks. The following protocol ensures the correctness of the dispatching by the Early Start algorithm:

- To examine whether a task T_i (the task to be dispatched) is scheduled in parallel with some other task T_j (line 10 in Figure 4.4 in **Step2.EARLYSTART**):

  If there is at least one task with non-zero *reset_δ* field in the PL between T_j and T_i, and if T_j is scheduled before T_i in the total ordering of PL, the values of these non-zero *reset_δ* fields must be summed up and *added* onto the scheduled start time st_i of T_i[4].

The number of tasks with non-zero *reset_δ* fields can be greater than one because there could have been more than one scheduler invocation between tasks T_j and T_i. Since there can be an arbitrary number of tasks scheduled between T_j and T_i in the PL, we need to bound the cost of this protocol. This can be accomplished by setting a bound b on the number of tasks the protocol will check for non-zero *reset_δ* fields between T_j and T_i. We start at task T_j. If T_i is not reached in less than or equal to b tasks, we simply will not dispatch T_i. This only results in pessimism in the reclamation of resources — the correctness of the system is not affected. Note that task T_i may potentially need to be checked against the first task in the PPL of each processor (see **Step2.EARLYSTART** in Figure 4.4). To efficiently implement this prototcol, only one pass through the PL is required, which accumulates the sum of

[4]Of course, in implementing this protocol, one only operates on a copy of st_i without modifying the true value of st_i assigned by the scheduler.

the *reset_δ* fields and applies them to each PPL head task (recall that PL is a total ordering). In this case, a bound b again needs to be set such that if within b tasks, not all the PPL head tasks have been reached yet (including task T_i since T_i must be the first task in a PPL too), we simply will not dispatch T_i.

5.6 Summary

In this chapter, we have studied various issues that arise in constructing a concurrent multiprocessor real-time system. Solutions were presented that ensure the *time conscientiousness* of the system. In particular, we quantitatively demonstrated the advantage of exploiting the parallelism provided by a distributed memory multiprocessor via a concurrent implementation. A synchronization mechanism with bounded waiting was described to serve as the foundation for shared resource access in a concurrent system. We presented the strategies we have developed to implement the scheduling process, the dispatchers and resource reclaiming in an integrated, predictable fashion. In the next chapter, using the implementation discussed in this chapter, we evaluate the performance of the resource reclaiming algorithms.

CHAPTER 6

PERFORMANCE EVALUATION

We demonstrated in the last chapter the applicability of the resource reclaiming algorithms presented in Chapter 4, by discussing their implementation in an integrated prototype real-time multiprocessor kernel, the Spring Kernel [89]. In addition, in order to understand the dynamic behavior and performance of the resource reclaiming algorithms, and to study the tradeoff between system overhead costs and runtime performance gains due to resource reclamation, we have done extensive simulation studies. Since it is difficult to collect elaborate performance statistics without affecting the true performance of an actual real-time system, we resorted to a software simulator which simulates the multiprocessor Spring Kernel.

In this chapter, in Section 6.1 we first describe our simulation methods, including system and algorithm parameters, overhead costs, and baseline schemes. To verify the validity of the overhead cost measurements of the software simulator, we conducted empirical tests on the Spring Kernel. In Section 6.2, we present the simulator validation results. A unique issue in evaluating algorithms designed for *dynamic* real-time multiprocessor systems is the problem of how to validate that the algorithms do not induce incorrect timing behavior in the system. Section 6.3 briefly discusses the schemes we have developed to deal with this issue. Experimental results are presented in Section 6.4, and Section 6.5 summaries the chapter.

6.1 Simulation Method

Simulation Parameters

In our simulations, the system overhead costs are the worst case costs measured on the Spring Kernel. The scheduler's cost SC is calculated before each invocation of the scheduler as follows: $SC = overhead\_cost + n * per\_task\_cost$, where n is the number of tasks to be scheduled for the current invocation of the scheduler. As mentioned in the previous chapter, in order to bound the cost of running the scheduler, we set a value N as the maximum number of tasks that the scheduler will schedule at a time, i.e., $n \leq N$ in calculating SC. In all the experiments, whenever the resource reclaiming algorithms are used, the cost of the algorithms are added onto a task's worst case execution time before the task is scheduled. Table 6.1 lists the worst case system costs and other simulation parameters respectively used in our simulation.

A 'v' in the *value* column in Table 6.1 means that the simulation parameter is a variable. The values listed for the various parameters are the values used in all or most of the experiments. If a value different from the one stated in Table 6.1 is used, it will be specified in presenting the results for that experiment. We have tested two cases for *wcet_min* and *wcet_max*. One is $wcet\_min = 50$ and $wcet\_max = 150$. The other is $wcet\_min = 50$ and $wcet\_max = 1000$. These two cases represent the two kinds of task systems in which the worst case execution times of tasks have small/large variance. We have found that in most cases, the performance of the resource reclaiming algorithms is almost the same for both cases of tasks' worst case execution times. We also present results for which we linearly increase the value of *wcet_min*, thereby decreasing the ratio of the cost of resource reclaiming to the average worst case execution time among tasks to decrease.

Load Estimation

Table 6.1 Simulation Parameters

parameter	*value*	*explanation*
overhead_cost	4	The portion of the scheduler's cost that is constant for each invocation of the scheduler.
per_task_cost	5	The portion of the scheduler's cost dependent on the number of tasks in the schedule.
Basic Reclaiming	1	The worst case cost of the Basic Reclaiming algorithm.
Early Start	2	The worst case cost of the Early Start algorithm.
number of processors	5	The number of processors used in the simulation.
number of resources	5	The number of resources used in the simulation.
wcet_min *wcet_max*	50 150	Tasks' worst case execution times are uniformly distributed between *wcet_min* and *wcet_max*.
l_{min} l_{max}	9 10	The laxity of a task is calculated based on the worst case execution time of the task, and it is uniformly distributed between l_{min} to l_{max} times the worst case execution time.
P_{use} P_{mode}	0.2 0.5	The probability that a task requires any of the resources. The probability that a task uses a resource in shared or exclusive mode if the task requires that resource.
actual execution time	(50%,90%)	A task's actual execution time, uniformly distributed between 50% and 90% of its worst case execution time.
L_{pi}	1.0	The average load of processor i (as explained in this section).
$\frac{1}{\lambda_i}$	v	The mean interarrival time of tasks on processor i, can be calculated for a given L_{pi} as in formula (6.1).

The combination of the mean interarrival time $\frac{1}{\lambda}$ of tasks, the value of P_{use}, the number of resources s, and *wcet_min* and *wcet_max* determines the average load of the system. In our simulation, tasks arrive as a Poisson process. We generate both *continuous* and *burst* arrivals in each simulation run. Every processor has the same $\frac{1}{\lambda_i}$, for $1 \le i \le m$. We use the following three formulas to measure the average processor load L_{pi}, the average resource load L_{ri}, and the resource conflict probability P_c for two tasks.

$$L_{pi} = \lambda_i * E[wcet] \tag{6.1}$$

$$L_{ri} = P_{use} * \lambda_i * E[wcet] * m \tag{6.2}$$

$$P_c \;=\; 1-(2(1-P_{use})*P_{use}+(1-P_{use})^2+(P_{mode}*P_{use})^2)^s \qquad (6.3)$$

$E[wcet]$ is the expected value of the worst case execution time of a task; thus it is either 100 or 525 for the two kinds of worst case execution times in our simulations. m is the number of processors. The first two formulas are straightforward. Note that the average resource load L_{ri} goes up as P_{use} increases even if the expected worst case execution time $E[wcet]$ and the mean arrival rate λ_i stay the same. In the third formula, P_c is the probability that two tasks will conflict on *any* of the given s resources (as opposed to P_{use} which is the probability that a task will require a resource). Thus P_c is a measure of the resource conflicts in a task load. In order to simulate task arrivals that have sufficient parallelism to be run on a multiprocessor system, we must keep the value of P_c fairly low. A high value of P_c would indicate the inherent resource conflicts among many tasks. P_c is calculated as 1 minus the probability that the two tasks will not conflict on any of the s resources. With respect to any one of the s resources in equation 6.3, the first term in the summation is the probability that only one task will require the resource, the second term is the probability that none of the two tasks will require the resource, and the third term is the probability that both tasks will require it in shared mode. P_c increases when the value of P_{use} or the value of s increases. So if we keep P_{use} the same for all the tasks, the more resources there are in a system, the more resource conflicts tasks will have.

Performance Metrics and Statistics

The performance metric we use is the *guarantee ratio* of an algorithm with respect to dynamic task arrivals. The *guarantee ratio* (GR) is defined as

$$GR = \frac{\text{the number of tasks guaranteed}}{\text{the number of tasks arrived}}.$$

In all the simulation experiments, each data point consists of five to ten runs. Our requirement on the statistical data is to generate 95% confidence intervals for the guarantee ratio whose width is less than 5% of the point estimate. To evaluate the effectiveness of the proposed resource reclaiming algorithms, we have also implemented the following three schemes for comparison purposes:

- **guarantee with actual execution time**: This is an ideal scheduling scenario. In this scheme, when a task arrives, the scheduler omnisciently knows the *actual*, rather than the *worst case*, execution time of the task. Therefore, resource reclaiming is not necessary.
- **rescheduling**: In the rescheduling scheme, whenever a task executes less than its worst case execution time, the scheduler is invoked to reschedule the tasks in the existing schedule in the same manner as when a new task arrives. The scheduler is invoked to do resource reclaiming only if the difference between the worst case execution time and the actual execution time of the completed task is greater than the scheduler's cost.
- **no resource reclaiming**: Here no resource reclaiming is done. Tasks are dispatched according to their scheduled start times. The case of no resource reclaiming provides a lower bound on performance.

6.2 Simulator Validation

To verify the validity of the performance and overhead cost measurements of the software simulator, we conducted empirical tests on the Spring Kernel. Table 6.2 shows the results of two task loads tested with the number of task arrivals and resources listed in the table. Each task load was tested on the actual Spring Kernel, as well as on the simulator with respect to *no resource reclaiming*, *using the Basic Reclaiming algorithm*, and *using the Early Start algorithm*. Two types of system overhead costs (including the costs of the scheduler and the resource reclaiming algorithms) were used for the simulator — the *average* and the *worst case* costs, both being the measurements from the actual kernel on a VME-bus based Motorola 68020 multiprocessor with a 16.67 MHz CPU. As shown in Table 6.2, when the average cost is used, the guarantee ratios produced by the simulator is very close to those of the Spring Kernel. And as expected, when the worst case cost is used in the simulator, the guarantee ratios are lower than those of the Spring Kernel. Since the objective of our simulation studies in the rest of this section is to evaluate the effectiveness of the resource reclaiming algorithms, it is important that the amount of performance gain/loss obtained in using the simulator is a good approximation of the actual kernel. Thus in the last two columns in Table 6.2, the difference in the guarantee ratios between the resource reclaiming algorithms and no resource reclaiming is shown. It is clear from these two columns that the performance gain of employing either of the resource reclaiming algorithms in the simulation with the worst case costs matches closely with the performance gain obtained in the actual kernel.

Table 6.2 Simulator Validation. NR = no resource reclaiming; BR = basic reclaiming; ES = early start.

	parameters			*Guarantee Ratios*			*Performance Gain*	
test	# tasks	# resources		NR	BR	ES	BR−NR	ES−NR
1	583	5	Spring Kernel	71.0	73.4	88.6	2.4	17.6
			sim(avg. cost)	71.0	73.4	91.4	2.4	20.4
			sim(worst cost)	58.0	61.0	72.0	3.0	14.0
2	566	7	Spring Kernel	66.8	69.9	80.0	3.1	13.2
			sim(avg. cost)	65.7	70.0	84.3	4.3	18.6
			sim(worst cost)	55.0	59.0	67.0	4.0	12.0

6.3 Post-Run Timing Verification: Understanding the Dynamic Behavior of the Algorithms

Besides performance issues (i.e., the statistical aspect of performance), there are two unique issues in the evaluation of concurrent on-line real-time algorithms:

(1) how to examine and understand the dynamic behavior of the algorithms, and

(2) how to verify the run time correctness of the dynamic behavior of the algorithm implementation.

The asynchrony and concurrency among the various components in both the algorithms and the integrated system make this examination and verification very difficult. However, one needs to be sure that tasks' timing and resource constraints are not violated by any of the algorithms used. We have designed and implemented schemes to collect run time task information to enable the understanding of the dynamic behavior of the algorithms, as well as to accomplish *post-run checking* of the correctness of the resource reclaiming algorithm implementation. We record the times when each task is actually dispatched, and when each task is completed in a post-run log file. This information is then used by two functions:

- One function, called the **Self-Checker**, takes as inputs the post-run log file and the original workload file in which each task's arrival time, deadline, resource requirements, and processor requirement are specified. The **Self-Checker** does the following two verifications:
  - A comparison of each task's start time and finish time in the post-run log file with the task's arrival time and deadline in the workload file. This comparison reveals whether the dispatching time of the task violates the task's timing constraints.

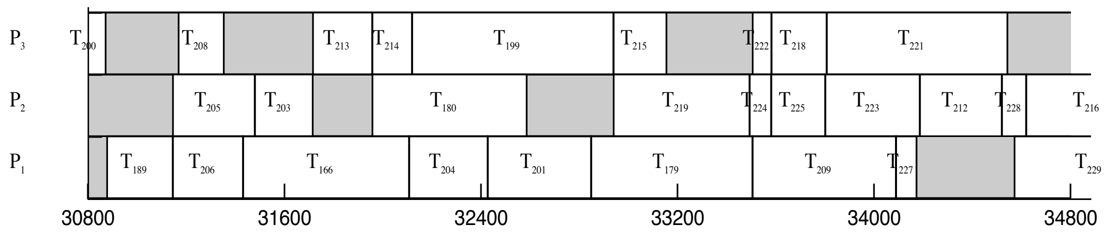


Figure 6.1 A post-run schedule when the **actual execution times** are known.

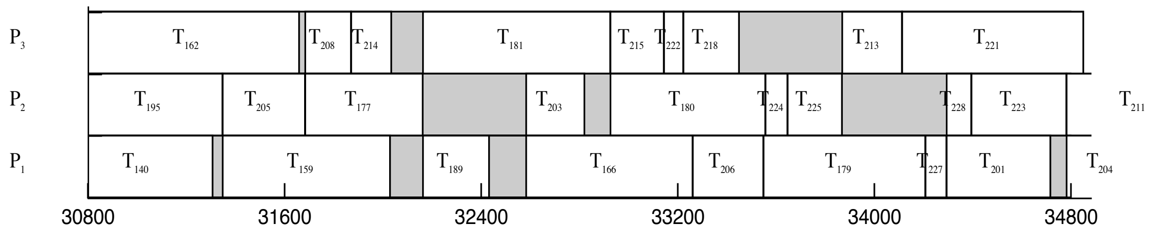


Figure 6.2 A post-run schedule generated by the **Early Start** reclaiming algorithms.

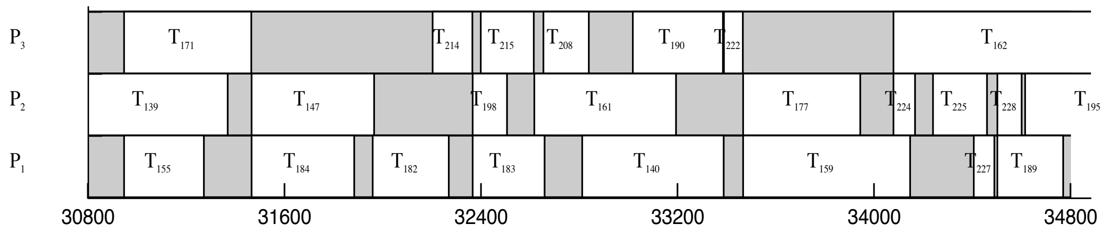


Figure 6.3 A post-run schedule generated by the **Basic** reclaiming algorithm.

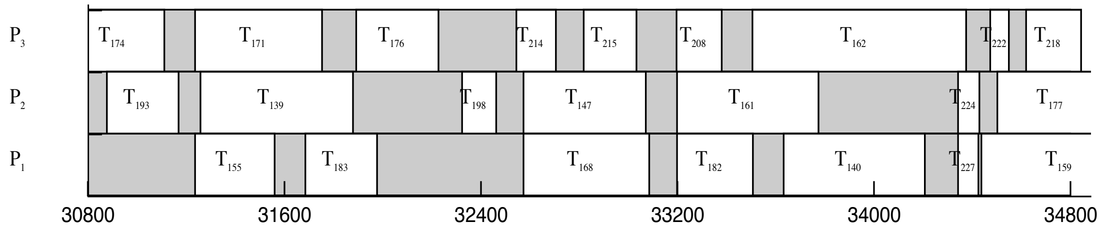


Figure 6.4 A post-run schedule generated by the **rescheduling** strategy.

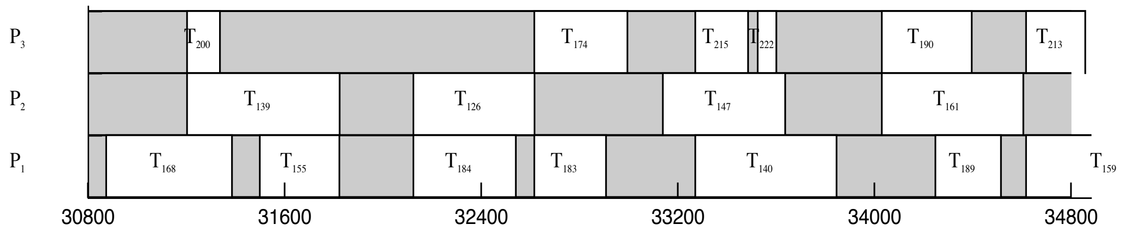


Figure 6.5 A post-run schedule generated with **no resource reclaiming**.

  - A comparison of the resource requirements among all the tasks whose run time executions overlap with each other. This comparison reveals whether any of the resource constraints are violated at run time, i.e., whether any two tasks with resource conflicts are executed in parallel.

- The second function, called the **Post-Run Schedule Generator**, takes as input the post-run log file, and generates a post-run schedule in the forms shown in Figures 6.1 through 6.5.

  This post-run schedule generation is extremely useful in observing and understanding the actual run time behavior of an algorithm. For example, Figures 6.1 through 6.5 show the post-run schedules of a portion of the simulation runs with respect to each of the resource reclaiming schemes we have tested from one simulation run. In these figures, we have generated each of the post-run schedules in the same time interval (from time 30800 to 34800)[1], so one can examine the task execution activity of the system in any time interval among different schemes. In particular, one can observe the following properties in the post-run schedules in Figures 6.1 to 6.5:

  - There is a large difference in the amount of idle time between the post-run schedule generated by using the Early Start reclaiming algorithm and the post-run schedule due to no resource reclaiming in Figures 6.2 and 6.5.
  - Since Reclaiming with Early Start allows *partial passing* while Basic Reclaiming does not allow any kind of *passing*, the difference in the amount of time reclaimed between the two resource reclaiming algorithms in Figures 6.2 and 6.3 attests to their effectiveness.

[1]The total length of this particular simulation is 85000 time units.

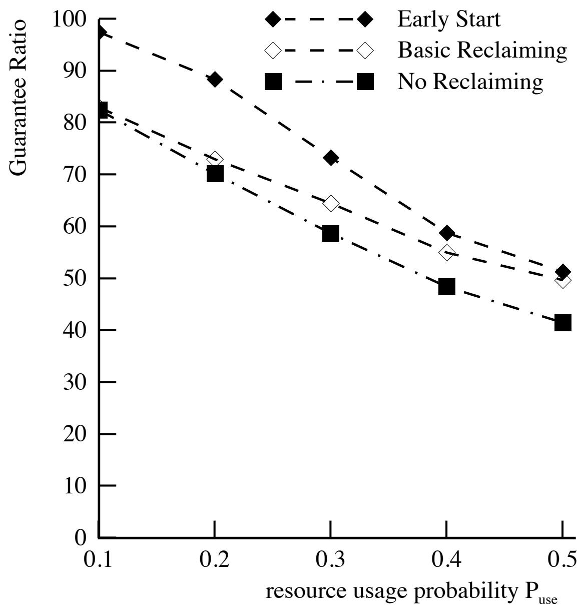


Figure 6.6 Performance of Basic Reclaiming and Reclaiming with Early Start

- When either resource reclaiming algorithm is used, there is no time interval during which *all* processors are idle. However, this is not true for the post-run schedules produced from using the rescheduling strategy or no resource reclaiming.

Although this post-run schedule generation enables us to better understand the dynamic behavior of an algorithm, it does not suffice as conclusive evidence to assert the performance of an algorithm. We proceed next to present statistical evidence that resource reclaiming greatly improves the performance of the system.

6.4 Simulation Results

6.4.1 Performance comparison of the two resource reclaiming algorithms

In Figure 6.6, the guarantee ratios from using the two resource reclaiming algorithms are plotted with that of using no resource reclaiming. In this simulation,

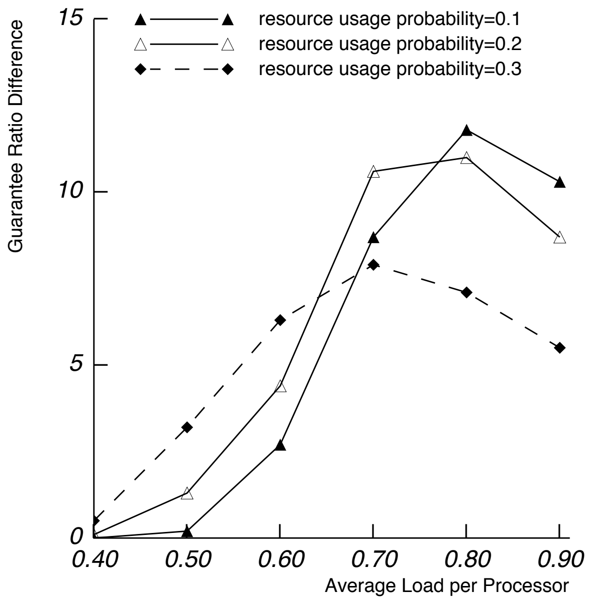


Figure 6.7 Guarantee (GR) ratio difference: GR of Early Start − GR of Basic Reclaiming.

$L_{pi} = 0.75$, and P_{use} varies from 0.1 to 0.5. This represents a heavy to overloaded system. For example, when P_{use} is 0.3, L_{ri} is 1.13, and when P_{use} is 0.5, L_{ri} is 1.9. Reclaiming with Early Start is very effective for all the resource usage probabilities. Its guarantee ratio is 18.4% higher than that of no resource reclaiming when $P_{use} = 0.2$. When the resource conflict is small (i.e., when $P_{use} \leq 0.3$ and thus $P_c \leq 0.3$), Reclaiming with Early Start performs much better than Basic Reclaiming since it can exploit more parallelism. When the value of P_{use} is 0.5, the performance of the Basic Reclaiming algorithm approaches that of Reclaiming with Early Start. When the value of P_{use} is too high, P_c is even larger, indicating high resource conflicts among tasks, and thus little parallelism among tasks. For example, for $P_{use} = 0.5$, $P_c = 0.65$. In this case there is a very high probability that any two tasks in the schedule will have resource conflicts. This will result in schedules in which very few tasks can be run in parallel. Since in using a multiprocessor system, one would expect a certain amount of parallelism to exist among the tasks, it is more appropriate to keep the value of $P_{use} \leq 0.3$ (thus, $P_c \leq 0.3$) in the rest of our experiments.

In Figure 6.7, the difference in guarantee ratios from using the two different resource reclaiming algorithms is plotted with respect to different processor loads L_{pi} and resource usage probabilities P_{use}. We vary the value of L_{pi} from lightly loaded (0.4) to heavily loaded (0.9). Then the guarantee ratio from using Basic Reclaiming is subtracted from the guarantee ratio from using Reclaiming with Early Start. The three curves correspond to simulations with three values of P_{use}. Based on the simulation results, we make the following observations:

- When the system is very lightly loaded, the difference between the two algorithms is negligible. Their difference peaks when L_{pi} ranges from 0.7 to 0.8, depending on the value of P_{use}. This is because when the system is lightly loaded, few tasks are simultaneously in the system, so Basic Reclaiming is just as effective.
- When the system is heavily loaded, many tasks will be scheduled in parallel. In this case the amount of resource reclaimed by the Reclaiming with Early Start algorithm is much larger than by the Basic Reclaiming algorithm.

From the results in Figures 6.6 and 6.7, we see that Reclaiming with Early Start outperforms Basic Reclaiming for most of processor loads and resource usage probabilities. Thus in the following experiments, we concentrate on evaluating the performance of Reclaiming with Early Start.

6.4.2 Performance Comparison with Rescheduling

The scheduler has a more *global* view of the tasks in the schedule than the resource reclaiming algorithm does, but it also has a higher run time cost. The purpose of this study is to answer the following question: 'Suppose we can reduce

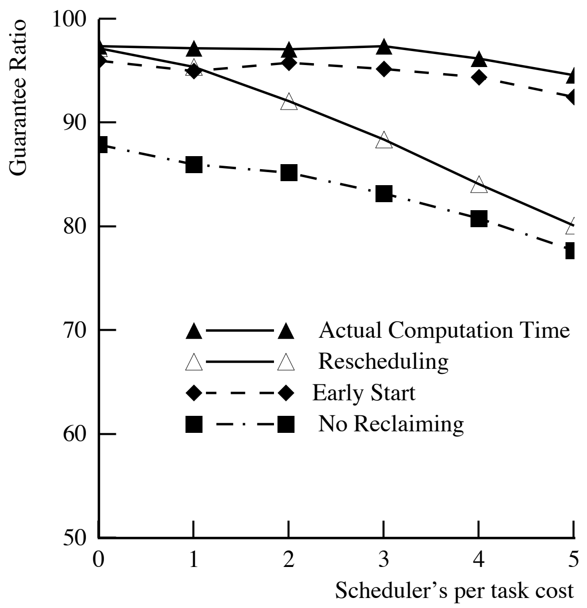


Figure 6.8 Effects of Scheduler's Runtime Cost

the cost of the scheduler. Will the rescheduling scheme be a better choice?' We compare the performance of the rescheduling scheme with that of (1) guarantee with actual execution time, (2) Reclaiming with Early Start, and (3)the no reclaiming schemes. Here we artificially vary the scheduler's *per_task_cost* from 0 to 5, where 5 is the actual worst case cost we have measured on the Spring Kernel.

The simulation results in Figure 6.8 indicate that the performance of rescheduling degrades by 17.1% when the cost of the scheduling algorithm increases from 0 to 5. Only when the scheduler's *per_task_cost* is zero, does rescheduling perform better than Reclaiming with Early Start. In real systems, the cost of the scheduler will be nonzero. So the rescheduling scheme is not a practical choice. The performance of Reclaiming with Early Start is very close to the performance of the guarantee with actual execution time scheme no matter what the cost of the scheduler is. This demonstrates that low complexity run time local optimization, such as the one used in Reclaiming with Early Start, can be very effective in a dynamic real-time system.

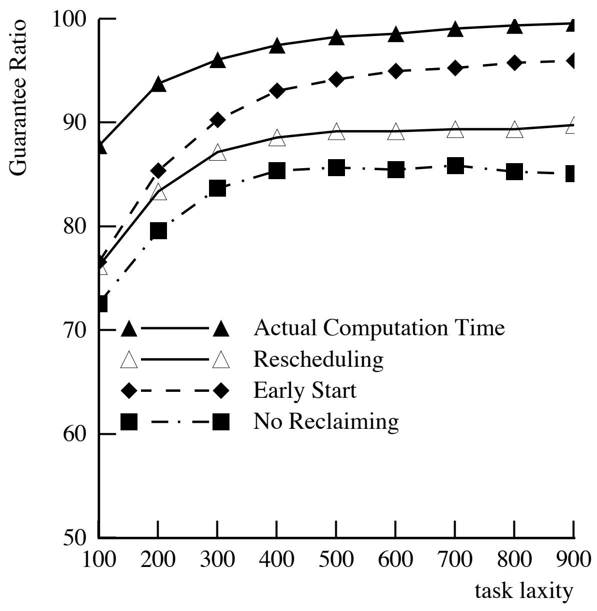


Figure 6.9 Effects of Task Laxity

6.4.3 Effects of Task Laxity

We now examine the performance of the various schemes with respect to different task laxities. Figure 6.9 shows the results of the experiments. Here tasks' laxities are plotted along the X-axis. At each x point, a task's laxity is drawn from a uniform distribution between $x\% * wcet$ and $x + 100\% * wcet$, where $wcet$ is the median of the worst case execution time of tasks. With tight task laxities, e.g., $x \leq 200$, resource reclaiming is not very effective, since, in this case, tasks arrive at the system with very small laxities, thus many of them cannot even be guaranteed with respect to their worst case requirements. As the laxities of the tasks are relaxed, the performance of Reclaiming with Early Start approaches the performance of the guarantee with actual execution time scheme, and is much better than that of rescheduling and no resource reclaiming. At $x = 900$, the difference between the guarantee ratios of using Reclaiming with Early Start and of using no resource reclaiming is 11%. On the other hand, rescheduling performs as well as

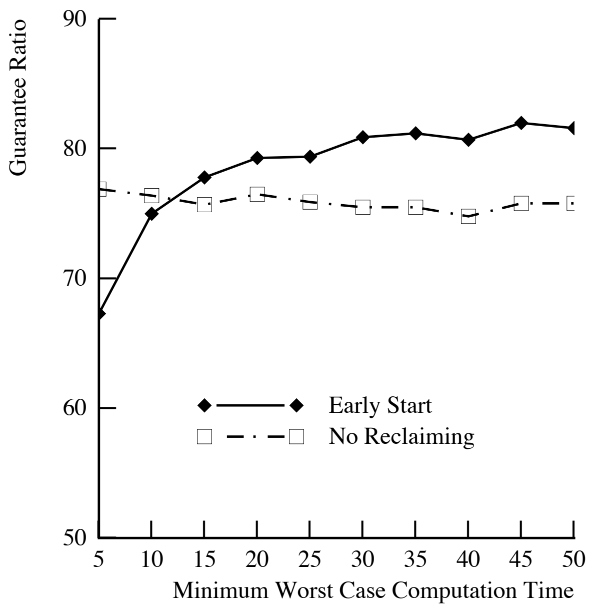


Figure 6.10 Effects of worst case execution time to resource reclaiming cost ratio.

Reclaiming with Early Start only when the laxity is very tight, i.e., when $x = 100$. It performs poorly as the laxity increases since the more tasks there are in the feasible schedule, the more rescheduling will cost. With larger task laxities, more tasks can be guaranteed, thus the feasible schedule contains more tasks.

6.4.4 Effects of Worst Case Execution Time

In Figure 6.10, we compare the performance of Reclaiming with Early Start with no reclaiming with respect to different worst case execution times. As the worst case execution times of tasks increase, the ratio $\frac{\text{resource reclaiming cost}}{\text{worst case execution time}}$ decreases. Recall that the run time cost of Reclaiming with Early Start is 2 (milliseconds). So, for the two kinds of worst case execution times we have tested so far, i.e., uniformly distributed between (50, 150) and between (50, 1000), the resource reclaiming overhead cost is at most 0.4% of a task's worst case execution time (since the minimum worst case execution time $wcet\_min = 50$ in both cases and $2/50 = 0.4$). What happens to the performance of resource reclaiming if $wcet\_min$ is smaller so that the ratio of the resource reclaiming overhead to the minimum worst case execution time becomes larger? In this experiment, we vary *wcet_min* from 5 to 50, and the worst case execution time of a task is uniformly distributed between *wcet_min* and $2 * wcet\_min$. P_{use} is set to 0.3. We did not include any scheduling overhead in this experiment for the purpose of examining the *pure* effects of the resource reclaiming overhead costs. In Figure 6.10, we plot the values of *wcet_min* on the X-axis. When $wcet\_min = 5$, the resource reclaiming overhead ranges from 20% to 40% of tasks' worst case execution times. When $wcet\_min = 50$, the resource reclaiming overhead is only 0.2% to 0.4% of tasks' worst case execution times. As one can see, if the resource reclaiming overhead can be more than 10% of tasks' worst case execution time, i.e., when $wcet\_min < 20$ on the X-axis, the guarantee ratio using Reclaiming with Early Start can be even worse than without any resource reclaiming. For the parameter settings tested in this experiment, the results show that it pays to do resource reclaiming only if one can ensure that the overhead cost of the resource reclaiming algorithm is below a reasonable percentage of tasks' worst case execution times, such as below 10%.

6.4.5 Effects of Average Processor Load

In all the above experiments, we have simulated heavy load situations. In Figure 6.11, we examine the performance of Reclaiming with Early Start with respect to different average processor loads L_{pi}. We vary the value L_{pi} from heavily loaded (1.0) to lightly loaded (0.3). A task's laxity is uniformly distributed between 1 to 10 times its worst case execution time, so that no matter what the average processor load is, tasks arrive with a large variance of laxities. We compare the performance of Reclaiming with Early Start with the performance of guarantee with actual execution time and no resource reclaiming. As the performance graphs indicate, the guarantee ratio of Reclaiming with Early Start follows closely to that

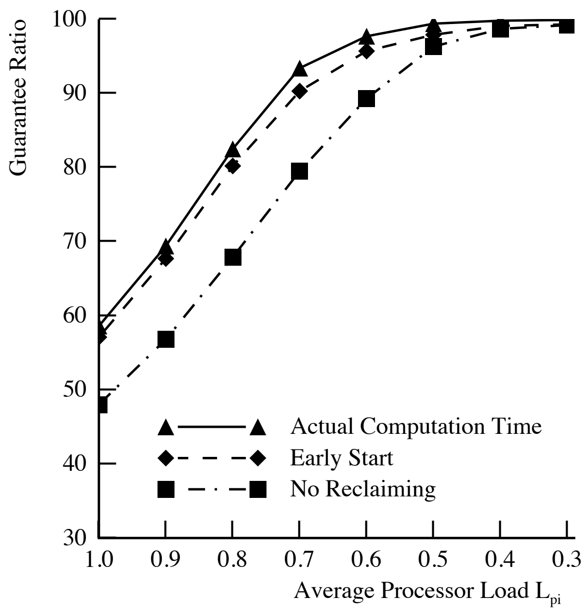


Figure 6.11 Effects of Average Processor Load

of guarantee with actual execution time for all the different loads. Except when the system is very lightly loaded, i.e., when $L_{pi} < 0.4$, Reclaiming with Early Start has a much higher guarantee ratio than with no resource reclaiming. At $L_{pi} = 0.8$, the difference between the guarantee ratios of Reclaiming with Early Start and no resource reclaiming is 14.3. When the load of the system is extremely low, e.g., at $L_{pi} = 0.3$, resource reclaiming is not necessary.

6.4.6 Effects of Actual Computation Time to Worst Case Computation Time Ratio

In all the simulations presented above, the actual execution time of a task is between 50% to 90% of its worst case execution time, drawn from a uniform distribution. Figure 6.12 shows the results for the case in which, for each simulation point, all the tasks in a task load have the same *ratio* of actual execution time to worst case execution time. This ratio is varied from 100% to 10%. We plot the percentage of the unused execution time on the x axis. This test studies the effect

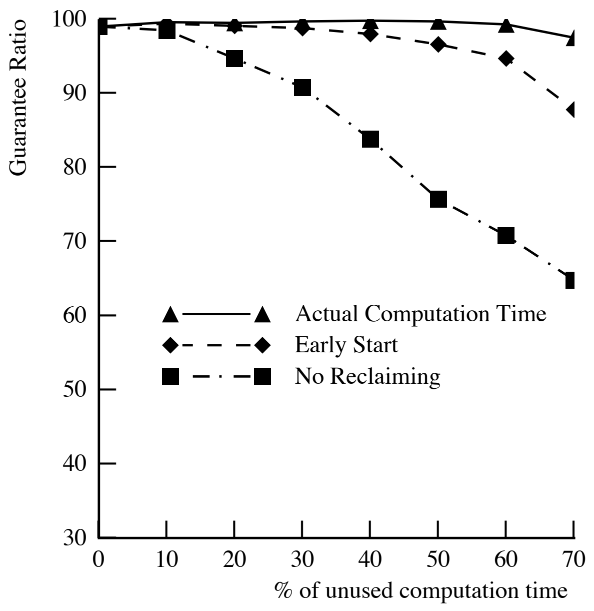


Figure 6.12 Effects of different actual to worst case execution time ratios

of the accuracy of worst case execution times upon performance. Note that for each test, even if all the tasks have the same actual execution time to worst case execution time ratio, their actual execution times are still very different due to the uniform distribution of their worst case execution times. P_{use} is set to 0.2. The average processor load has been calculated according to tasks' actual execution times rather than their worst case execution times, i.e., $L_{pi} = \lambda_i * E[\text{actual execution time}]$. At each simulation point, we generated the same average processor load $L_{pi} = 0.6$ with respect to the expected actual execution time, so that if we had known the actual execution times of tasks, the task load was mostly feasible as demonstrated by the performance of guarantee with actual execution time. However, since when using Reclaiming with Early Start and no resource reclaiming we do not know the actual execution times at schedule time, the smaller the ratio of the actual execution time to the worst case execution time (as the tasks leave more unused execution time), the larger the worst case load the system has to handle.

The simulation results indicate that

(1) For a large range of the accuracy of worst case execution time estimation (from 50% to 100%), Reclaiming with Early Start performs very close to that of the guarantee with actual execution time scheme.

(2) The improvement on the guarantee ratio of Reclaiming with Early Start over no resource reclaiming is substantial. The guarantee ratio improves by 23.9% when tasks' actual execution time is 40% of their worst case execution times.

6.4.7 Positive Side Effects of Resource Reclaiming

In this section, we show two positive side effects of resource reclaiming on the system's performance.

Limiting the Length of the Schedule

In Figures 6.13 and 6.14, we compare the performance of using the Early Start algorithm with that of guarantee with actual execution time. All experimental parameters are the same for these two figures, except the distribution of the worst case execution times of tasks, and the laxities. In Figures 6.13, tasks' worst case execution times are uniformly distributed between 450 and 600, while in Figure 6.14, they are between 50 and 1000. So, the median task worst case execution times are the same for both experiments, but in one the tasks' worst case execution times have much larger variance than in the other. A task's laxity is uniformly distributed between 9 to 10 times its worst case execution time, thus smaller tasks would have much smaller absolute laxities than large tasks.

As Figure 6.14 show, the performance of guarantee with actual execution times can even drop slightly below that of using Reclaiming with Early Start. This happens at the higher resource usage probabilities when the values of tasks' worst case execution times vary greatly, and with them their laxities. This unexpected performance 'anomaly' is a positive side effect of resource reclaiming. At higher

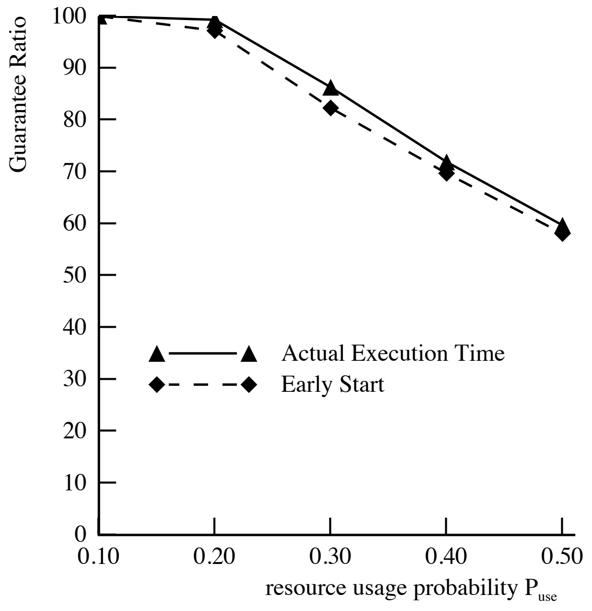


Figure 6.13 Guarantee Ratios: *wcet_min* = 450, *wcet_max* = 600.

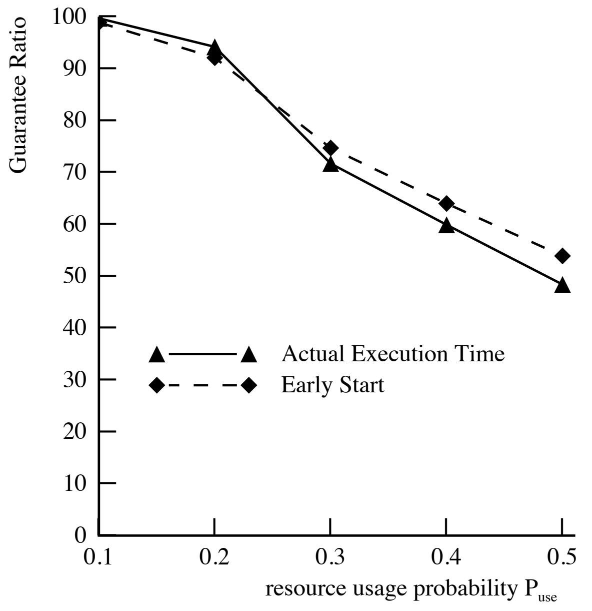


Figure 6.14 Guarantee Ratios: *wcet_min* = 50, *wcet_max* = 1000.

resource usage probability, few tasks can execute in parallel, and thus guaranteed tasks would most likely be scheduled near their deadlines in the resulting schedule. Task arrivals with small laxities need to be scheduled close to the current time in order to meet their deadlines — this implies that the worst case execution time of the scheduler can influence the schedulability of the smaller tasks, because the placement of the *cutoff_line* imposes constraints on how close to the current time a task can be guaranteed in a schedule. We have observed from the simulation outputs for the experiments shown in Figures 6.13 and 6.14 that the number of tasks prior to the *cutoff_line* is on the average smaller when using the Early Start reclaiming algorithm than when guaranteeing with the actual execution time. This is because, in using resource reclaiming, the scheduler needs to schedule tasks with respect to their worst case execution times. When the larger tasks finish earlier than their worst case execution time, they in effect free up portions of time that are close to the current time in the feasible schedule. This makes it possible for many of the newly arrived small tasks to be guaranteed, thus increasing the guarantee ratio. Since the small tasks finish executions very fast, by the time the next scheduler invocation occurs, the small tasks have already been completed, thus not contributing to the calculation of the *cutoff_line*. This phenomenon causes the performance 'anomaly' shown in Figure 6.14.

Reclaiming with Early Start mitigates the effects of the variance in the scheduler's worst case execution time estimation

We saw that resource reclaiming can overcome the potential performance degradation due to the use of worst case execution times for scheduling. The same applies to worst case estimation of the scheduler's costs to compute the *cutoff_line*. As we have described in Chapter 5, at each invocation of the scheduler, the worst case execution time of the scheduler must be calculated and is used to determine the *cutoff_line*. Most of the time, the scheduler does not execute up to its worst case execution time, and finishes the scheduling activity earlier than the *cutoff_line* indicates. Without a resource reclaiming scheme, the time interval between the actual completion of the scheduler and the *cutoff_line* can be idle due to early completions of application tasks. When a resource reclaiming algorithm is in use, as soon as the scheduler completes execution, dispatching and resource reclaiming can take effect.

6.5 Summary

In summary, the simulation results show that the resource reclaiming algorithms proposed are very effective with respect to a wide range of system and task parameters. In particular, the following can be observed:

- Good local optimization can be very effective in a dynamic real-time system.
- In a real-time system, it is important to employ run time algorithms with *bounded* time complexity. The complexity of the algorithm should be *independent* of the number of tasks.
- Beside having *bounded* time complexity, it is essential for a resource reclaiming algorithm to be *inexpensive* in terms of overhead cost. Our simulation results indicated that it only pays to do resource reclaiming if one can ensure that the overhead cost of the resource reclaiming algorithm is below 10% of tasks' worst case execution times.
- Resource reclaiming is very useful for real-time systems that have to guarantee tasks with respect to their worst case execution times. For a large range of accuracy of the worst case execution time estimation (from 30% to 100%) that we have experimented with, Reclaiming with Early Start performs very close to that of an ideal scheduling scenario – *guarantee with actual execution time.*
- Even though Reclaiming with Early Start has a higher run time cost than that of Basic Reclaiming, it performs much better than Basic Reclaiming in most of the task and system parameter settings.

- However, there are two situations in which the performance of the resource reclaiming algorithms falls much below that of scheduling with ACET. The first is when the task laxities are small (below 2 to 3 times that of tasks' average WCET) as shown in Figure 6.9, and the second is when the worst case execution time is much larger than the actual execution time (with greater than 70% inaccuracy) as shown in Figure 6.12. In the next chapter, we show how these can be dealt with by using early notification of a reduction in a task's worst case execution time *during* (rather than at the end of) the execution of the task.
- Simple resource reclaiming algorithms are needed most when the system is heavily loaded and the invocation of the scheduling algorithm is expensive compared with the resource reclaiming algorithms.
- When the load of the system is extremely low, e.g., $L_{pi} \leq 0.3$, resource reclaiming is not necessary.

Most importantly, the simulation results show that, for independent tasks with resource constraints, resource reclaiming can greatly compensate for the performance loss due to the inaccuracies of the estimation of the worst case execution times of real-time tasks for a wide range of system parameter values.

CHAPTER 7

Resource Reclaiming: Not Just at Task Completion

So far in this dissertation, we have assumed that the worst case execution time of a task in a schedule does not change throughout the execution of a task, i.e., no information during the execution of a task is used to reduce the worst case execution time of the task. The resource reclaiming algorithms presented in Chapter 4 are designed to recognize and utilize the time left *after* a task completes execution. One is prompted to ask here that "*Can we do better?*". In this chapter, we present extensions to the resource reclaiming problem that show that we can.

7.1 The Task Execution Time Life Cycle

In a dynamic real-time system, the execution time of a task goes through a *life cycle* as Figure 7.1 shows:

- $WCET^T$ — *Theoretical worst case execution time*, which can be analyzed and derived during task compilation and decomposition (based on assumptions about the underlying system).

- $WCET^E$ — *Effective worst case execution time*, which can be calculated after the placement/allocation of tasks and resources is known.

- $WCET^I$ — *Invocation worst case execution time*, which can be calculated at the task invocation/scheduling time when the resource requirements and other information pertinent to a *particular* invocation of the task are available.

- $WCET^R$ — *Runtime worst case execution time.* This may be updated from time to time during a task's execution depending on the evaluations of branching conditions and loop control variables in a program.
- $ACET$ — *Actual execution time*, which is only known at task completion time.

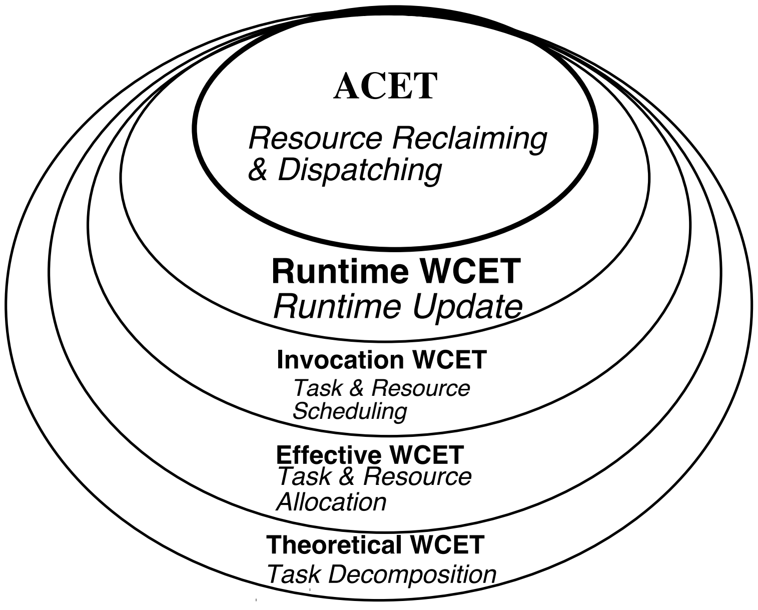


Figure 7.1 The life-cycle of the execution time of a task.

So clearly,

$$WCET^T \geq WCET^E \geq WCET^I \geq WCET^R \geq ACET.$$

Dynamically, there are two different points in time when the system can be notified of a change in a task's execution time from its invocation worst case execution time $WCET^I$:

- **case 1**: At the task completion time — the actual execution time (ACET) of a task cannot be known until a task completes its execution.

- **case 2**: At the end of the evaluation of some branching condition in a task's program — the branch of computation a task is going to take may reduce the worst case execution time of the task from $WCET^I$ task to $WCET^R$ (runtime worst case execution time), which is smaller than the $WCET^I$.

Thus, in case 2, even after partial execution of a task, it is possible to reduce the invocation $WCET$. By recognizing when reductions of $WCET^I$ are possible, we can further alleviate the negative effects due to worst case execution assumptions on the system performance. Note here that even though the $WCET^I$ may be reduced at this point, the actual execution time ($ACET$) at the task completion may still be different from the $WCET^R$. In the rest of this chapter, we extend resource reclaiming to deal with case 2.

7.2 Early Notification of Reduction in $WCET^R$

The value of *reclaimed_δ* (recall that the value of *reclaimed_time* is maintained by the resource reclaiming algorithms and contains the value of the amount of time the system has reclaimed up to now on all the processors and resources with respect to the current feasible schedule) can be updated only at each task completion. In case 2 above, the change in the value of $WCET^R$ does not affect the existing schedule and the value of *reclaimed_δ* immediately, i.e., the reduced execution time cannot be reclaimed immediately. If there is no new task arrival until the completion of the task execution, then the function and performance of the resource reclaiming algorithms are the same as before — the early completion of the task will be recognized by the resource reclaiming algorithms. However, if a new task arrival occurs, the amount of time reduced can be potentially used by the scheduling process immediately.

This *early notification* of a reduction in the worst case execution time of a task during its execution can be facilitated as follows:

(1) For each task T_i:

- After each *test and branch* statement for a *conditional* (i.e., an *if-then-else*), if the shorter execution path of the two branches is to be taken, the program can be annotated by inserting a statement indicating the amount of reduction in the worst case execution time.
- Before each *loop control* statement, once the value of the number of loop iterations is known, the program can be annotated by inserting a statement indicating the amount of reduction in the worst case execution time.

(2) If the value of reduction is greater than some threshold (this threshold should be determined statically with respect to the cost of the following described system call), a system call can be made to *record* this reduction in a new *expected* finish time of task T_i in the schedule (recall that we do not remove a task from the PL until it completes execution). The execution time of this system call can be bounded and inexpensive since (1) only one *expected* finish time needs to be modified, and (2) the PL does not need to be locked because one single write-operation is atomic. Thus, in the worst case, the scheduler would simply *miss* the update of this new *expected* finish time, and the feasible schedule up to the *cutoff_line* is still correct.

It should be pointed out here that some of the values of conditionals and loop control variables can be determined at task invocation time, and thus are already reflected in the value of $WCET^I$. However, it is not feasible to extract the values of all possible occurrences of conditionals and loop control variables in a task at invocation time since this would essentially require the exploration of an *exponential* number of paths. Thus run time reduction of $WCET^I$ is the only choice.

7.3 Integrating Early Notification with the Reclaiming-with-Early-Start Algorithm

Since the simulation studies have shown that the resource reclaiming with *partial passing*, i.e., the Reclaiming-with-Early-Start algorithm, outperforms the Basic-Reclaiming algorithm most of the time, in the following we discuss the issues in integrating *early notification* and Reclaiming-with-Early-Start.

To support Reclaiming-with-Early-Start, the new *expected* finish time of task T_i should be recorded in an attribute eft_i of T_i, but the original ft_i of T_i cannot be modified. This is because the original ft_i contains local information of both possible resource conflicts and permissible parallelism between T_i and its neighboring tasks in the feasible schedule. This information is used by the resource reclaiming/dispatching process to make the correct decision on whether a task can be dispatched earlier than its original scheduled start time. If the original ft_i were modified, the crucial information about permissible parallelism among tasks in the schedule would have been altered. This is because the scheduled start times and scheduled finish times assigned by the scheduler are guaranteed to satisfy tasks resource constraints[1]. This in turn would severely limit the performance of the Reclaiming-with-Early-Start algorithm which allows *partial passing.*

Therefore, when combining *early notification* and Reclaiming-with-Early-Start, each task T_i in the schedule has a ft_i (scheduled finish time) and an eft_i (expected finish time) associated with it. With these two attributes, the following modifications need to be incorporated into the integrated scheduling and dispatching system:

- Initially, $eft_i = ft_i$.
- During T_i's execution, eft_i may be reduced.

[1]However, the resource conflict information will not be affected even if the original ft_i is modified, because eft_i must be less than or equal to ft_i, i.e., for some task T_j, if $st_j > ft_i$, then $st_j > eft_i$.

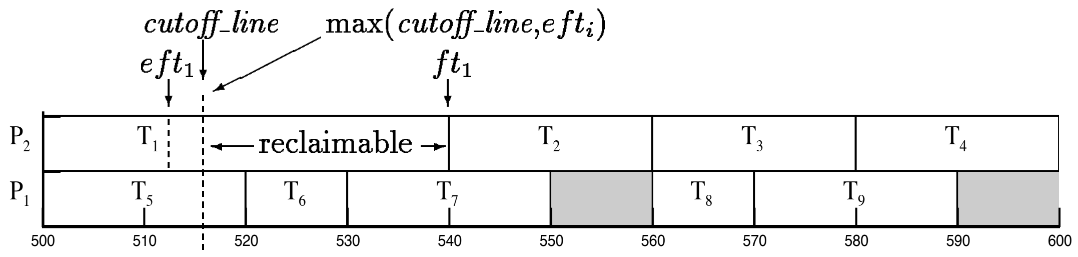


Figure 7.2 Example of Early Notification

- Whenever the scheduler is invoked during the execution of T_i, the scheduler uses the eft_i to calculate the *cutoff_line* and the earliest available times of resources.
- When T_i completes execution, the resource reclaiming and dispatching decisions are still based on ft_i.

Let us suppose the task in question is T_i with an original scheduled finish time ft_i. Now let us suppose that at a branching statement, we know that in the worst case T_i will complete by eft_i where $eft_i < ft_i$. (eft_i can be thought of as a new *expected* finish time of task T_i.) If, after eft_i is known, a new task arrives and the scheduler is to be invoked, as long as the *cutoff_line* is less than ft_i, the time interval $\tau = [max(cutoff\_line, eft_i),\ ft_i]$ can be used by the scheduler in constructing the new schedule, i.e., τ is reclaimed such that the earliest available time of the processor and resources T_i uses can be set to $max(cutoff\_line, eft_i)$ in the earliest available time vectors for the scheduling algorithm. For example, in Figure 7.2, at time 500 the evaluation of a conditional branch in task T_1 reduces its scheduled finish time $ft_1 = 540$ to $eft_1 = 512$. Suppose a scheduler invocation occurs at this point and the *cutoff_line* is placed at time 516. Then $\tau = [max(cutoff\_line, eft_1),\ ft_1] = [cutoff\_line,\ ft_1] = 24$ is reclaimable on processor P_2 and the resources needed by task T_1, and can be used by the scheduler in the scheduling process.

7.4 Performance Implications

The simulation studies of the resource reclaiming algorithms presented in the previous chapter show that Reclaiming-with-Early-Start performs very well compared to scheduling with ACET under most of the system parameter settings. However, there are two situations in which its performance falls much below that of scheduling with ACET. The first is when the task laxities are small (below 2 to 3 times that of tasks' average WCET) as shown in Figure 6.9, and the second is when the worst case execution time is much larger than the actual execution time (with greater than 70% inaccuracy) as shown in Figure 6.12.

In designing a system for a real-time application, a desirable property is to have very low run time cost for the scheduler, i.e., the ratio between the scheduler's execution cost and the average task execution time should be very small. In a nonpreemptive scheduling environment, this means that we would like the scheduler to complete execution before the currently executing application task(s) finish execution. Thus, in our case it is desirable that the *cutoff_line* can be drawn at the end of the tasks still in execution on the multiprocessor (i.e., only the tasks in execution are not being considered in rescheduling together with the new task arrival(s)). In our simulation studies, the ratio of the average WCET of tasks and the scheduler's cost is such that this property is true. In such a system, *early notification* could alleviate some of the performance degradations under the two situations mentioned above:

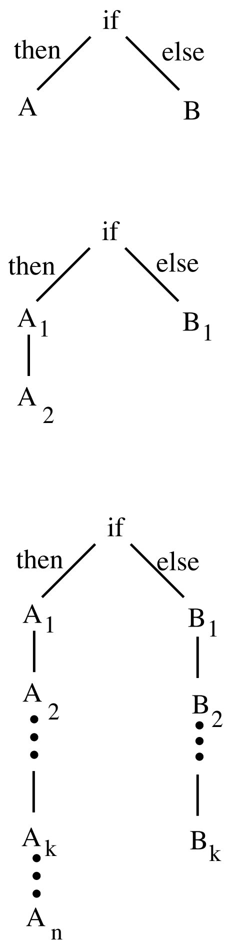
- With very small task laxities, if a task can be guaranteed at all, it must have been scheduled fairly close to the current time. So tasks need to be placed in the *front* portion of a schedule to meet their deadlines. With early notification, when a new task arrives, there can be *more* available time that is close to the current time for the task to be guaranteed. In this case, the early notification of a reduction in the WCET of the tasks in execution should improve the guarantee ratio.

- With very large variance between the worst case and the actual case execution time of tasks, the *cutoff_line* could have been drawn at a much later time than needed by the worst case execution time of the scheduler due to nonpreemption of the tasks in execution. This would obviously decrease the guarantee ratio. The early notification of a reduction in the WCET of the tasks in execution will cause the actual *cutoff_line* to be drawn at a time before the tasks' original scheduled finish time (i.e., we can reclaim the time interval $\tau = [max(cutoff\_line, eft_i),\ ft_i])$. Again, there is *more* time reclaimed for task arrivals during scheduling. This can potentially remedy the performance degradation.

7.5 Early Notifications of $WCET^R$ Reduction for More Than One Task

The solution described above achieves early notifications of a $WCET^R$ for a single task. However, it is conceivable that the evaluation of a branching condition during one task's execution may result in the reduction of the $WCET^R$ of more than one task (e.g., a set of tasks that belong to the same task graph), or may even result in the deletion of some tasks already guaranteed in the schedule. This may occur when some of the computation paths in a program are very long, and multiple tasks with multiple scheduling points can be derived at task decomposition time. This implies that the $WCET^R$ values of more than one task in the feasible schedule can depend upon the evaluation of the *same* branching condition.

Suppose a program $\mathcal{P}$ contains an *if-then-else* conditional statement whose evaluation may result in one of the two alternative execution paths A and B to be taken at run time. Figure 7.3 illustrates three possible task representations of $\mathcal{P}$.



$$WCET_{T_{(A,B)}} = max(WCET_A, WCET_B)$$

$$WCET_{T_{(A_1,B_1)}} = WCET_{A_1} = WCET_{B_1}$$

$$WCET_{T_{A_1}} = WCET_{A_2}$$

$$WCET_{T_{(A_1,B_1)}} = max(WCET_{A_1}, WCET_{B_1})$$

$$\vdots$$

$$WCET_{T_{(A_k,B_k)}} = max(WCET_{A_k}, WCET_{B_k})$$

$$WCET_{T_{A_{k+1}}} = WCET_{A_{k+1}}$$

$$\vdots$$

$$WCET_{T_{A_n}} = WCET_{A_n}$$

Figure 7.3 Three possible task decompositions of program $\mathcal{P}$.

Let $T_{(A,B)}$ be the task containing both computation paths, and where each of A and B is to be decomposed into multiple schedulable portions. Let $T_{(A_i,B_i)}$ be the task containing the i^{th} portions of computation paths A and B. Also let $WCET_U$ be the theoretical worst case execution time (we omit the superscript here) of U where U is either a task or a portion of a computation path. Depending on the semantics and nature of the computation of A and B, the task decomposition process (such as the one proposed in [63]) may produce any one of three task combinations and their corresponding $WCET$ for program $\mathcal{P}$ as shown in Figure 7.3:

(1) $\mathcal{P}$ can be represented as one task with its worst case execution time equal to the maximum of A and B, i.e., $T_{(A,B)}$ such that $WCET_{T_{(A,B)}} = max(WCET_A, WCET_B)$. This is the simplest case and is what we have assumed up to now.

(2) In the case when the execution path A is much longer than B, and the first schedulable portion of A has exactly the same worst case execution time as that of execution path B, $\mathcal{P}$ can be decomposed into two tasks $T_{(A_1,B_1)}$, and T_{A_2}, where $T_{(A_1,B_1)}$ has $WCET_{T_{(A_1,B_1)}} = WCET_{A_1} = WCET_{B_1}$, and T_{A_2} has $WCET_{T_{A_2}} = WCET_{A_2}$.

(3) Execution path A is equal to or longer than B. In the most general case, we cannot make the assumption that A and B can be always decomposed into respective portions of computations such that the i^{th} portion of A has the same worst case execution time as the i^{th} portion of B. The *exact* match in the worst case execution times between portions of two different execution paths is very unlikely due to resource requirements in the computation. Therefore, in the most general case, $\mathcal{P}$ can be decomposed into two sets of tasks $\{T_{(A_1,B_1)}, ..., T_{(A_k,B_k)}\}$ and $\{T_{A_{k+1}}, ..., T_{A_n}\}$, For $1 \leq i \leq k$, $WCET_{T_{(A_i,B_i)}} = max(WCET_{A_i}, WCET_{B_i})$, and for $k+1 \leq j \leq n$, $WCET_{T_{A_j}} = WCET_{A_j}$.

Then the question is: *Can we accomplish the modification of more than one eft* predictably *at run time*? The answer is *yes*. Since at task decomposition, the number of tasks per execution branch is known, the maximum number of modifications of eft is also known. So the cost X of the multiple eft modifications can be incorporated into the $WCET$ of the task containing the branch condition. The multiple updates of eft values in the schedule should be constructed within a critical section, otherwise race conditions can occur between this updating action and the scheduler's setting of the *cutoff_line* right before scheduling a newly arrived task. When the *cutoff_line* is set, the tasks being rescheduled are severed from the dispatching queues. Without using a critical section, the eft updating process can be accessing invalid task entries.

Suppose task T_i contains a branching condition whose evaluation at run time can potentially reduce the $WCET^R$ of k_i tasks, including that of task T_i itself. Then the cost X_i of updating the eft of the k_i tasks includes the following components:

(A) The cost x of the system call that updates one $WCET^R$, as described in section 7.2.

(B) The cost of a critical section to access the schedule:

(B.1) the amount of time $\pi_{p(),v()}$ for $P()$ and $V()$ operations, and

(B.2) the amount of time for updating k_i $WCET^R$'s.

So the total cost is $X_i = k_i * x + \pi_{p(),v()}$ for task T_i. The value of X_i can be used to determine the threshold to control whether a multiple $WCET^R$ updating action should be carried out. In many situations, the threshold can be checked ***statically*** since at each conditional branch, the number of potential $WCET^R$ reductions is known, thereby avoiding run time checking cost.

When reducing the $WCET^I$ of a task T_i that will be dispatched at some future point in time, we need to modify not only the eft_i of the task, but also the value of T_i's current $WCET^I$. When a task arrival occurs, the reduced value of T_i's $WCET^I$ can be used by the scheduler in the scheduling process, thereby in effect reclaiming the reduced portion of the invocation worst case execution time. If the evaluation of a conditional branch results in the removal of a task from the schedule, the value of the task's $WCET^I$ can be simply set to zero. From a performance point of view, the resource reclaimed due to the reduction in $WCET^I$ values of tasks and the deletion of tasks in the existing feasible schedule can be very beneficial and even crucial for the guarantee of incoming new tasks, since the number of tasks and/or the amount of computation/resource requirements have been reduced for the tasks already guaranteed.

7.6 Summary

We have described a mechanism for the early notification of reductions in a task's worst case execution time during the execution of the task. The possible impact on the performance of resource reclaiming has been identified. Besides the worst case execution time reduction of a single task, we have also considered the reduction of the worst case execution times of multiple tasks due to the evaluation of the same conditional branch. We demonstrated that this multiple modification can be accomplished with bounded cost.

CHAPTER 8

Conclusion and Future Work

8.1 Dissertation Summary

In this dissertation, we have taken an integrated approach to attack the problems of algorithm design for dynamic multiprocessor real-time systems that require the properties of being *time conscious*, *time conscientious*, and *time conserving.* Real-time scheduling algorithms require the use of worst case execution times of tasks. However, the worst case execution time is an upper bound, and the actual execution time of a task at run time varies between some minimum value and this upper bound. The variance in task execution time can be caused by both the computer architecture and the software features. The problem of *on-line* resource reclaiming in a multiprocessor real-time system has not been addressed previously. The research presented in this dissertation represents an initiative effort in characterizing and solving this dynamic resource reclaiming problem.

We have presented a predictable integration of scheduling, dispatching and resource reclamation for *guarantee-oriented* distributed memory real-time multiprocessor systems. We have demonstrated that, for a dynamic real-time system, it is not sufficient to simply analyze and prove the *static* properties of an on-line algorithm in isolation of the rest of the system components. The sharing and contention of resources, such as memory, shared system bus, and more importantly *time*, mandates the algorithm designer to take an *integrated* view of the system as a whole, considering the interrelationships of all the system components (be it software, or hardware) that have an effect on the dynamic timing properties of the algorithm at hand. In the following, we briefly summarize the specific contributions and results of this dissertation.

In particular:

- We have analyzed the worst case run time anomalies that can occur in a multiprocessor schedule where real-time tasks have both resource and processor constraints, and the necessary conditions under which timing anomalies can occur in a resource constrained multiprocessor schedule if a *work-conserving* scheme is used. We have also studied the complexity and optimality issues of multiprocessor resource reclaiming.

- We have developed two resource reclaiming algorithms, Basic Reclaiming and Reclaiming with Early Start. While these two algorithms both have *low* and *bounded* execution time, they vary in complexity. These two algorithms employ strategies that are a form of on-line *local optimization* on a feasible multiprocessor schedule. We also analyzed the correctness and complexity of these algorithms.

- Simulation studies of the resource reclaiming algorithms for a five processor multiprocessor system have been carried out. We tested a wide range of task parameters and compared the performance of the resource reclaiming algorithms to that of three other baseline schemes. The simulation results show that

  - (1) resource reclaiming is very useful for real-time systems that have to guarantee tasks with respect to their worst case execution times,

  - (2) algorithm Reclaiming with Early Start outperforms Basic Reclaiming, and,

  - (3) for a large range of accuracies of the worst case execution time estimations (from 30% to 100%) that we have experimented with, Reclaiming with Early Start performs very close to that of an ideal scheduling scenario.

- Solutions that extend resource reclamation to not only at a task's completion, but also during its execution have also been proposed.

- *Predictable* integration of multiple functional components is a challenge unique to real-time systems, and is required to support the *time conscientious* and *time conserving* properties of a real-time system. This challenge is exacerbated by the difficulties brought about by the concurrent and asynchronous nature of *multiprocessor* systems.

  - We have presented a concurrent, on-line, bounded-time construction of the scheduler and dispatchers for a distributed memory multiprocessor real-time system.

  - The feasibility of on-line resource reclaiming is demonstrated via an integration with the scheduler and dispatcher processes implemented on a prototype (Motorola 68020, VME based) multiprocessor real-time kernel — the Spring Kernel.

8.2 Future Work

In a multiprocessor environment, synchronized clocks are easily supported. The integrated scheduling-dispatching-resource-reclaiming approach presented in this dissertation assumes the availability of such a synchronized clock on each multiprocessor node. In a distributed environment, synchronizing the scheduling and dispatching operations of the entire system will result in a very inflexible implementation, if possible at all. The on-line resource reclaiming problem for distributed real-time systems need further research efforts.

8.2.1 Resource Reclaiming for Tasks with Communication Constraints

In the resource reclaiming algorithms presented in this dissertation, tasks will be started earlier than their scheduled start time if possible. The correctness criterion only applies to tasks that do not have synchronization/communication constraints.

Now, suppose two tasks in two different feasible schedules, i.e., tasks resident on different multiprocessor nodes in a distributed system, have communication constraints — one task ends with a SEND and another task starts with a RECEIVE instruction. If the two tasks are on the same node, then the resource reclaiming algorithms presented can still be used as they are. However, if the two tasks are scheduled on different nodes in a network, the resource reclaiming algorithms may start a receiving task before its corresponding sender task. One possible extension is to set up synchronization *barriers* between the sender and the receiver, such that the resource reclaiming process will not *start* the receiver task earlier on one node than the sending task on the other node. With synchronization *barriers*, the performance of resource reclaiming can be limited without allowing *total passing.* Thus, in a dynamic distributed system where tasks have communication constraints, in order to overcome the performance problem due to worst case execution assumptions, a better choice may be to design scheduling schemes with an *implicit* schedule construction.

8.2.2 Distributed Resource Reclaiming with Fault Tolerance Constraints

Multiple copies of tasks may reside on different multiprocessor nodes in a distributed system. Replications of tasks may be used for two types of fault tolerant task structures — (1) voting, and (2) alternative versions. In voting, all the replicas of a task must be executed, and their outputs voted on. In the alternative scheme, only one copy of the task needs to be executed. Once one copy of a task successfully completes execution, the other copies (or alternative versions) can be removed from the current schedules on the other nodes.

The removal of copies of a task brings new issues to the resource reclaiming problem — we must be able to ensure that (1) only *one* copy is executed, and (2) we must be able to remove the other copies from the schedules on other nodes. The unique problem here is that resource reclamation can proceed as long as none of the copies of the task has been dispatched. However, as soon as one copy starts execution, the execution or removal of the other copies is contingent upon whether the started copy can complete successfully. How to accomplish resource reclaiming in this scenario is an open question. An alternative is to take an *at-least-once* semantics of the executions of the alternative versions of a task, thus resource reclaiming can be carried out as it is.

BIBLIOGRAPHY

[1] Agne, R. Global Cyclic Scheduling: A Method to Guarantee the Timing Behavior of Distributed Real-Time Systems. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 3(1), March 1991.

[2] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. Technical Report 88–09–04, University of Washington, September 1988.

[3] Baker, T. P. Stack Based Scheduling of Realtime Processes. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 3(1), March 1991.

[4] Baker, T. P. and Shaw, A. The Cyclic Executive Model and Ada. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 1(1), 1989.

[5] Baruah, S., G. Koren, D. M., Mishra, B., Raghunathan, A., Rosier, L., Shasha, D., and Wang, F. On the Competitiveness of On-Line Real-Time Task Scheduling. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 4(2), June 1992.

[6] Bettati, R. and Liu, J. W. S. Algorithms for Flow-Shop Scheduling to Meet Deadlines. *the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[7] Blake, B. A. and Schwan, K. Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[8] Blazewicz, J. Deadline Scheduling of Tasks With Ready Times and Resource Constraints. *Information Processing Letters*, 8(2), February 1979.

[9] Blazewicz, J. Solving the Resource Constrained Deadline Scheduling Problem via Reduction to the Network Flow Problem . *European Journal of Operational Research*, 6, 1981.

[10] Blazewicz, J., Drabowski, M., and Weglarz, J. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, 35(5), May 1986.

[11] Blazewicz, J., Lenstra, J., and Kan, A. R. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[12] Burns, J. E. Mutual Exclusion with Linear Waiting using Binary Shared Variables. *SIGACT News*, 10(2), Summer 1978.

[13] Carlow, G. D. Architecture of the Space Shuttle Primary Avionics Software System. *Communications of the ACM*, 27(9), September 1984.

[14] Cheng, S.-C., Stankovic, J. A., and Ramamritham, K. *Hard Real-Time Systems*, chapter 5.1 Scheduling, pages 150–173. Computer Society Press of the IEEE, 1988.

[15] Chetto, H. and Chetto, M. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transaction on Software Engineering*, 15(10), Oct. 1989.

[16] Coffman, Jr., E. G. *Computer and Job-Shop Scheduling Theory.* J. Wiley & Sons, 1976.

[17] Cristian, F., Dancey, B., and Dehn, J. Fault-Tolerance in the Advanced Automation System . In *20th Annual International Symposium on Fault-Tolerant Computing*, June 1990.

[18] Dertouzos, M. L. and Mok, A. K.-L. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks . *IEEE Transactions on Software Engineering*, 15(12), Dec. 1989.

[19] Desrochers, A. A., editor. *Modeling and Control of Automated Manufacturing Systems* . IEEE Computer Society Press, 1990.

[20] Dinning, A. A Survey of Synchronization Methods for Parallel Computers. *Computer*, 22(7), July 1989.

[21] Du, J. and Leung, J. Y.-T. Scheduling Tree-Structured Tasks With Restricted Execution Times. *Information Processing Letter*, 28, July 1988.

[22] EcElvany, M. C. Guaranteed Deadlines in MAFT. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1988.

[23] Garey, M. R. and Graham, R. L. Bounds for Multiprocessor Scheduling with Resource Constraints . *SIAM Journal on Computing*, 4(2):200–187, June 1975.

[24] Garey, M. R. and Johnson, D. S. Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM Journal on Computing*, 4(4), Dec. 1975.

[25] Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[26] Gillies, D. W. and Liu, J. W. Greed in Resource Scheduling . In *Proceedings of the 10th IEEE Real-Time System Symposium*, 1989.

[27] Goli, P., Kurose, J., and Towsley, D. Approximate Minimum Laxity Scheduling Algorithms for Real-Time Systems. Technical Report 90-88, University of Massachusetts, 1990.

[28] Gottlieb, A., Grishman, R., Kruskal, C., McAuliffe, K., Rudolph, L., and Snir, M. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, c-32(2), February 1983.

[29] Govindan, R. and Anderson, D. P. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Oct. 1991.

[30] Graham, R. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.*, 17(2), March 1969.

[31] Graham, R. Bounds on the Performance of Scheduling Algorithms. In Coffman, Jr., E. G., editor, *Computer and Job-Shop Scheduling Theory*, chapter 5. J. Wiley & Sons, 1976.

[32] Gunterberg, H. Case Study on Rapid Software Prototyping and Automated Software Generation: An Inertial Navigation System. Master's thesis, Naval Postgraduate School, 1989.

[33] Han, C.-C. and Lin, K.-J. Scheduling Parallelizable Jobs on Multiprocessors. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1989.

[34] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[35] Holmes, V. P. and Harris, D. L. A Designer's Perspective of the Hawk Multiprocessor Operating System Kernal. *ACM Operating System Review*, 23(3), July 1989.

[36] Hong, J., Tan, X., and Towsley. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Transactions on Computers*, 38(12), December 1989.

[37] Hong, K. S. and Leung, J. Y.-T. On-Line Scheduling of Real-Time Tasks. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec 1988.

[38] Hugue, M. M. and Stotts, P. D. Guaranteed Task Deadlines for Fault-Tolerant Workloads with Conditional Branches. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 3(3), September 1991.

[39] Hunter & Ready, Inc. *VRTX/68020*. Hunter & Ready, Inc., 1986.

[40] Jeffay, K., Stone, D. L., and Smith, F. D. Kernel Support for Live Digital Audio and Video. In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Nov. 91.

[41] Jensen, E. D., Locke, C. D., and Tokuda, H. A Time-Driven Scheduling Model for Real-Time Operating Systems. *Proceedings of IEEE Real-Time Systems Symposium*, dec 1985.

[42] Kawaguchi, T. and Kyan, S. Deterministic Scheduling in Computer Systems: A Survey. *Journal of the Operations Research Society of Japan*, 31(2), June 1988.

[43] Klingerman, E. and Stoyenko, A. D. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, Sept 1986.

[44] Komoda, N., Kera, K., and Kubo, T. An Autonomous, Decentralized Control System for Factory Automation. *IEEE Computer*, December 1984.

[45] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. Distributed Fault-Tolerant Real-Time Systems: the MARS Approach. *IEEE Micro*, Feb. 1989.

[46] Kurose, J. F. and Chipalkatti, R. Load Sharing in Soft Real-Time Distributed Computer Systems. *IEEE Transactions on Computers*, c-36(8), Aug 1987.

[47] Lawler, E. L. and Martel, C. U. Scheduling periodically Occurring Tasks on Multiple Processors . *Information Processing Letters*, 12(1), Feb. 1981.

[48] Lehoczky, J. P., Sha, L., and Strosnider, J. K. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *Proceedings of IEEE Real-Time Systems Symposium*, December 1987.

[49] Leinbaugh, D. W. Guaranteed Response Times in a Hard-Real-Time Environment. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.

[50] Leung, J. Y.-T. Bounds on List Scheduling of UET Tasks with Restricted Resource Constraints. *Information Processing Letters*, 9(4), Nov. 1979.

[51] Leung, J. Y.-T. and Merrill, M. L. A Note on preemptive Scheduling of Periodic, Real-Time Tasks . *Information Processing Letters*, 11(3), Nov. 1980.

[52] Levi, S.-T., Tripathi, S. K., Carson, S. D., and Agrawala, A. K. The MARUTI Hard Real-Time Operating System. *ACM Operating System Review*, 23(3), July 1989.

[53] Liu, C. L. and Layland, J. W. Scheduling Algorithms for multiprogramming in a Hard-Real-Time Environment. *Journal of ACM*, 20(1), 1973.

[54] Liu, J. W. S., Lin, K.-J., and Natarajan, S. Scheduling Real-Time, Periodic Jobs Using Imprecise Results. In *Proceedings of IEEE Real-Time Systems Symposium*, 1987.

[55] Locke, C. D. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 4(1), March 1992.

[56] Locke, D. C. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1986.

[57] Manacher, G. K. Production and stabilization of real-time task schedules. *Journal of the ACM*, 14(3), July 1967.

[58] Mehrotra, R. and Varanasi, M. R., editors. *Multirobot Systems.* IEEE Computer Society Robot Technology Series. IEEE Computer Society Press, 1990.

[59] Menga, G., Bruno, G., Conterno, R., and Dato, M. A. Modeling FMS by Closed Queuing Network Analysis Methods. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, CHMT-7(3), September 1984.

[60] Molesky, L. D., Shen, C., and Zlokapa, G. Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems . *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 2(3):163–180, Sept. 1990. (Also published as COINS Tech. Report 90-30).

[61] Motorola Inc. *MVME135, MVME135-1, MVME135A, MVME136, and MVME136A 32-Bit Microcomputers User's Manual.* Motorola Inc., 1989.

[62] Nain, P. and Towsley, D. Properties of the ML(n) Policy for Scheduling Jobs with Real-Time Constraints. In *in the Proceedings of 29-th IEEE Control and Decision Conference*, December 1990.

[63] Niehaus, D. Program Representation and Translation for Predictable Real-Time Systems. In *IEEE Real-Time Systems Symposium*, Dec. 1991.

[64] Nirkhe, V. and Pugh, W. A Partial Evaluator for the Maruti Hard Real-Time System. In *IEEE Real-Time Systems Symposium*, Dec. 1991.

[65] Norbis, M. *Heuristics for the Resource Constrained Scheduling Problem.* PhD thesis, University of Massachusetts, 1987.

[66] Park, C. and Shaw, A. C. Experiments With A Program Timing Tool Based On Source-Level Timing Schema. In *IEEE Real-Time Systems Symposium*, Dec. 1990.

[67] Peterson, J. L. and Silberschatz, A. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, 1985.

[68] Peterson, J. L. and Silberschatz, A. *Operating System Concepts.* Addison-Wesley Publishing Company, Inc., 1987.

[69] Puschner, P. and Koza, C. Calculating the maximum execution time of real-time programs. *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 1(2), Sept. 1989.

[70] Rajkumar, R., Sha, L., and Lehoczky, J. P. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec 1988.

[71] Ramamritham, K. Allocation and Scheduling of Complex Periodic Tasks. *10th International Conference on Distributed Computing Systems*, June 1990.

[72] Ramamritham, K., Stankovic, J. A., and Shiah, P.-F. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[73] Ramaritham, K. and Stankovic, J. A. Dynamic Task Scheduling in Distributed Hard Real-Time Systems. *IEEE Software*, 1(3), July 1984.

[74] Rangan, P. V. and Vin, H. M. Designing File Systems for Digital Video and Audio. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[75] Ready, J. F. VRTX: A Real-Time Operating System for Embedded Microprocessor Applications. *IEEE MICRO*, August 1986.

[76] Richards, P. Parallel Programming. Technical Report TD-B60-27, Technical Operations Inc., 1960.

[77] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* PhD thesis, Stanford University, 1989.

[78] Schwan, K., Bihari, T., Weide, B. W., and Taulbee, G. High-Performance Operating System Primitives for Robotics and Real-Time Control Systems. *ACM Transactions on Computer Systems*, 5(3), August 1987.

[79] Sequent Computer Systems Inc. *Sequent Symmetry Technical Summary.* Sequent Computer Systems, Inc., 1988.

[80] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority Inheritance Protocols—An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[81] Sha, L., Rajkumar, R., Lehoczky, J. P., and Ramamritham, K. Mode Change Protocols for Priority-Drive Preemptive Scheduling . *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 1(3), Dec. 1989.

[82] Shaw, A. C. Reasoning about time in higher-lever language software. *IEEE Transactions on Software Engineering*, 15(7), July 1989.

[83] Shen, C., Ramamritham, K., and Stankovic, J. A. Resource Reclaiming in Multiprocessor Real-Time Systems . *to appear in Transactions on Parallel and Distributed Systems*, 1992.

[84] Shih, W.-K., Liu, J. W. S., and Chung, J.-Y. Fast Algorithms for Scheduling Imprecise Computation. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[85] Shin, K. G. and Epstein, M. E. Intertask Communications in an Integrated Multirobot System. *IEEE Journal of Robotics and Automation*, RA-3(2), April 1987.

[86] Sprunt, B., Sha, L., and Lehoczky, J. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 1(1), June 1989.

[87] Stankovic, J. A. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, Oct 1988.

[88] Stankovic, J. A., Ramamritham, K., and Cheng, S. C. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transaction on Computers*, 34(12), Dec 1985.

[89] Stankovic, J. A. and Ramamrithm, K. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3), May 1991.

[90] Stewart, D. B. and Khosla, P. K. Real-Time Scheduling of Sensor-Based Control Systems. *the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[91] Tarjan, R. E. Decomposition by clique separators. *Discrete Matchematics*, 55, 1985.

[92] Xu, J. and Parnas, D. L. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3), March 1990.

[93] Zhao, W. *A Heuristic Approach to Scheduling Hard Real-Time Tasks with Resource Requirements in Distributed Systems.* PhD thesis, University of Massachusetts, Amherst, 1986.

[94] Zhao, W. and Ramamritham, K. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints. *Journal of Systems and Software*, 1987.

[95] Zhao, W., Ramamritham, K., and Stankovic, J. A. Preemptive Scheduling under Time and Resource Constraints. *IEEE Transaction on Computers*, 36(8), August 1987.

[96] Zhao, W., Ramamritham, K., and Stankovic, J. A. Scheduling Tasks with Resource Requirements in Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12, May 1987.

[97] Zhao, W. and Stankovic, J. A. Performance Analysis of FCFS and Improved FCFS Scheduling Algorithms for Dynamic Real-Time Computer Systems. In *in the proceedings of IEEE Real-Time Systems Symposium*, December 1989.

[98] Zhou, H. and Schwan, K. Dynamic Scheduling for Hard Real-Time Systems Toward Real-Time Threads. *the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.