REAL-TIME SYSTEM DESIGN USING PREEMPTION THRESHOLDS

YUN WANG

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2001

0-612-59233-2

Abstract

Real-Time System Design using Preemption Thresholds

Yun Wang, Ph.D.
Concordia University, 2001

As the real-time embedded systems encountered in applications such as telecommunications, aerospace, defense, and automatic control demand large, complex and multi-tasked software development, a new challenge has emerged for adopting the state-of-the-art software engineering technologies. Object-oriented design provides a scalable methodology with appropriate CASE tools for the design of software systems. Although these tools provides support for visual object-oriented modeling, design, simulation and code generation for general real-time systems, timing analysis is only available after the software is constructed. Consequently, the design-development process involving these tools in real-time systems becomes iterative and time-consuming.

Introducing timing analysis in the design stage encounters a new problem. Traditional scheduling theory assumes a single level of task granularity. However, in industrial practice, common wisdom requires several design level tasks map into one run-time thread to reduce scheduling costs. This warrants a dual-level scheduling: preemptive scheduling between threads and non-preemptive scheduling between tasks in the same thread. Extending the scheduling theory to such an environment forms the scope of this thesis.

Preemption threshold is introduced to control undesirable preemptions. Via a novel application of this concept, this thesis proposes a general scheduling model that subsumes both preemptive and non-preemptive scheduling models as special cases. The new theory deals with both independent and dependent tasks derivable from an object-oriented system model. Motivated by UML-RT modeling, the dependencies in our model include inter-task communication, resource sharing, and precedence. Important design issues covered include task priority and preemption threshold assignment and optimized task to thread mapping with respect of minimum scheduling cost and memory requirement. Quantitative performance evaluation is also conducted via simulation to validate the theory prosposed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The dramatic improvement in the performance and reduction in the cost of microprocessors has led to an explosion of concurrent and real-time applications. With the capability of delivering sophisticated functionality while still meeting stringent performance requirements, embedded real-time applications have gained popularity in commercial, industrial, military, medical, and consumer products. At the same time, the increasing size and complexity of real-time applications poses a challenge to the traditional low-level and unscalable techniques in real-time software development.

Historically, there is a conservative tendency in the real-time community, which reflects the importance of stringent response time requirements, as well as safety and reliability issues in real-time system design. The preference of approaches that have been proved to work over possibly better, but less proven technologies has prevented real-time software designers from adopting state-of-the-art software engineering technology. However, the quick-growing demands for large and extremely complex multi-tasked real-time softwares, especially in applications such as telecommunications, aerospace, defense, and automatic control, urgently requires dramatic improvements in efficiency and productivity of real-time software development.

One of the most promising approaches to managing complexity of software development is model based software development. Various commercial tools have been developed in recent years to support model based development using high level modeling languages like Unified Modeling Language (UML) [RJB99, BRJ99]. Model based development has also been very successful in various specialized domains, for example in telecommunications, and control systems software development. Model based design facilitates the design process by using modeling abstractions that are closer to the design space, and by using visual notations that facilitate understanding of the designs. Furthermore, model based design also facilitated various forms of analysis to help in the design process. Moreover, with the support of code generation, the benefits of modeling may extend through the software's life-cycle by automatically translation the design model into an implementation for a desired target platform [Bel98, SGW94].

1.1 Motivation

Dealing with the stringent time constraints has been recognized as one of the most challenging and critical part of incorporating object-oriented and model based software design methodologies in real-time system design. Historically, timing behavior of the real-time system can only be studied through simulations or experiments on the hardware when the software is implemented. Early consideration of timing issues is desirable and considered an essential aspect of real-time system design since it may reduce the number of iterations in the process and avoid the ad-hoc behaviors in making implementation choices. However, it has not been possible until recently due to the lack of mathematical basis that could support it.

Recently, the maturity of schedulability analysis techniques, and in particular those based on fixed priority scheduling theory [LL73, LSD89, HKL91, TBW94] has facilitated the introduction of timing analysis in the early stages of real-time system design [BW94, Gom93, VB93]. One prominent and representative example is the HRT-HOOD design method for hard real-time systems [BW94]. HRT-HOOD provides design abstractions that are motivated by the tasking models of fixed-priority scheduling theory, thereby providing direct support for the schedulability analysis of HRT-HOOD designs. Such abstractions include the cyclic and sporadic objects, representing periodic and sporadic tasks respectively. However, this approach is limited to a relatively small number of jobs and each job is of a relatively course granularity. And it does not work well with some of the object-oriented design methods; for example ROOM [SGW94] and OCTOPUS [AKZ96]. (Please refer to Section 2.5.1)

Most of these scheduling theories assume a relative simple analysis model, where all the jobs are scheduled concurrently based on their priorities. This preemptive multi-tasking scheduling model abstracts the popular priority-based multi-threaded RTOS (Real-Time

Operating System), such as VxWorks. The high degree of concurrency in the analysis model provides considerable flexibility in the design. However, this model incurs relatively high context switching cost with a large number of tasks and a small granularity of each tasks. In addition, there is a per-task memory cost, which may challenge the constrained resource of most embedded real-time systems. In practice, real-time system designers cluster jobs into tasks to reduce the number of tasks and increase the granularity of each task. This decision is done with some heuristics from experience, for example, task clustering and task inversion [Gom00]. These heuristics look into the temporal behavior of the system to provide hints for merging tasks that will not or should not preempt each other. Period or sample rate, functional relationship, time-criticism, sequential dependence, state-dependent control, and mutual exclusion are the factors to look after while applying these heuristics.

Some object-oriented design methods introduced for modeling real-time systems, for example ROOM [SGW94] and OCTOPUS [AKZ96], view the system as a collection of concurrent (or active) objects that cooperate in implementing system functionality. Thus, each concurrent object participates in multiple system functions, and is subject to multiple timing constraints. Moreover, to maintain internal consistency of the object, the requests on an object are processed in a "run-to-completion" manner, i.e., there is no internal concurrency within an object.

Multiple jobs in one task and run-to-completion process manner of object implies a dual-level scheduling model: while preemptive scheduling is used between threads, non-preemptive scheduling is used between jobs in the same thread. In industry, this implies the adoption of a dual-level multi-tasking architecture. In this architecture, threads are implemented as event-handlers. While threads are scheduled preemptively, the scheduling of events within a thread is done in a non-preemptive manner by the event handling loop of the thread. Thus, this implementation architecture requires relatively small number of threads that reduces preemptive multi-tasking costs. Within a thread, event scheduling involves minimal context-switching costs since events are processed on a "run-to-completion" basis.

There are two major open issues in this scheduling model. First, no scheduling theory has clearly addressed how the dual-level scheduling affects the schedulability of a real-time system. Second, although some heuristics may apply, there is no systematic approach with a sound mathematical basis available to control the preemptability and map jobs in analysis model to tasks in implementation model. These open issues serve as major motivation for this thesis.

1.2 Thesis Overview

In this thesis, we develop techniques that address the open issues mentioned above. Furthermore, we also address the issue of automatic assignment of scheduling attributes, such as priorities, to each job (event) to reduce the ad-hoc decision in design while ensuring the schedulability of the system. We develop a new dual-priority scheduling model that integrates and subsumes fixed priority preemptive and non-preemptive scheduling models and provides the basis for analyzing and synthesizing implementations for the dual-level implementation architecture described above.

The dual priority scheduling model is based on the notion of preemption threshold, and makes use of the following observation: if the priority of a job is raised once it starts executing, then we can limit the number of jobs that can preempt it. This elevated priority for a job is called its preemption threshold. Clearly, in the extreme case if the preemption threshold is set to the maximum priority level then no job can preempt it. The preemption threshold of jobs can then be "tuned" to get the desired degree of preemptability in the system.

An interesting consequence of the preemption threshold scheduling model is that we can identify jobs that cannot preempt each other. Thus, it is possible to group a set of jobs that cannot preempt each other into a "non-preemptive group." Since jobs within a group cannot preempt each other, they can all then be processed within a single thread as described in the dual-level implementation architecture (we assume that each event corresponds triggers a single job).

To simplify the problem, we first develop techniques for independent jobs. For this independent task model, we show how we can compute response times if the scheduling attributes of all jobs are known a priori. Then, we show how to make use of the response time analysis to synthesize scheduling attributes such that the resultant system is feasible (i.e., all jobs meet their response time requirements). Additionally, we show how the jobs can then be grouped together into threads to reduce run-time overheads, while maintaining feasibility.

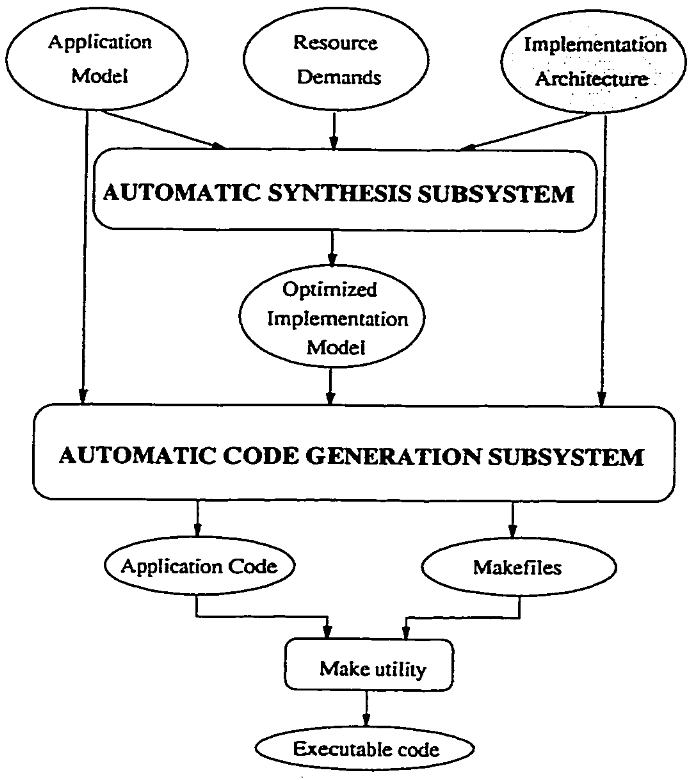
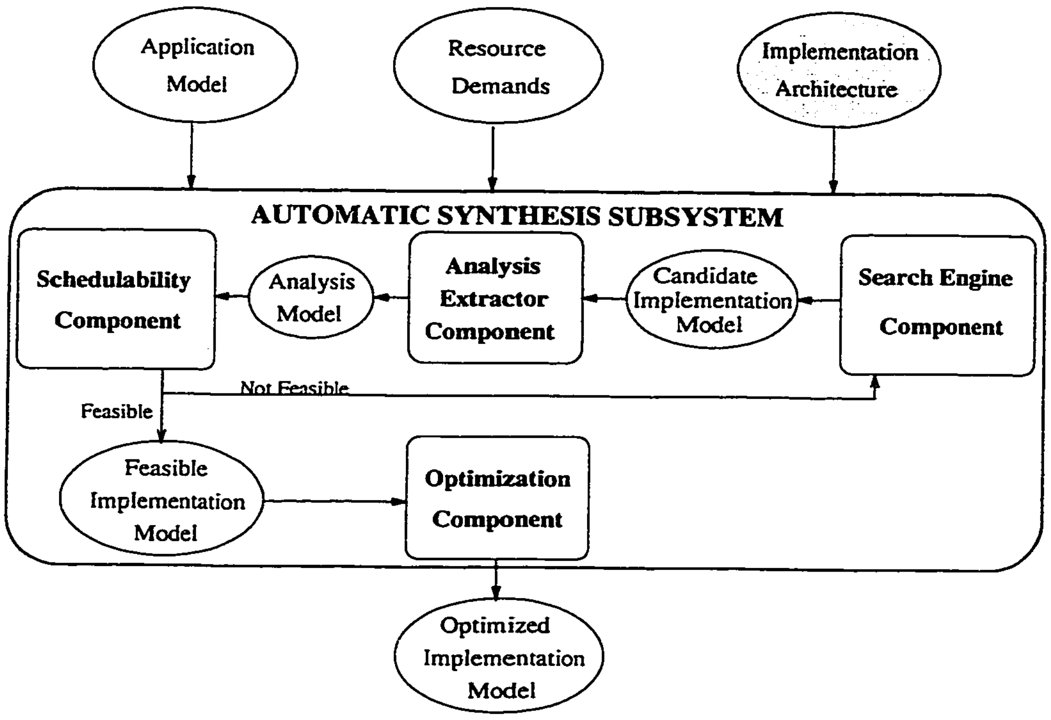

Application Model
Resource Demands
Implementation Architecture
AUTOMATIC SYNTHESIS SUBSYSTEM
Optimized Implementation Model
AUTOMATIC CODE GENERATION SUBSYSTEM
Application Code
Makefiles
Make utility
Executable code

Figure 1.1: Overview of the Implementation Generation

After solving the problem for independent jobs, we show how the results can be generalized to include dependencies between jobs that include precedence constraints and resource sharing. The general model is based on the design models from object-oriented design methods such as ROOM [SGW94]. The techniques developed in this thesis may be viewed as complementary to automatic code generation. They play an important role in our approach of automatic synthesis of multi-tasking implementations for real-time object-oriented models [SKW00]. An overview of the approach is given in Figure 1.1. Since the techniques can be automated, they can be implemented in a tool that automatically synthesizes scheduling attributes and mapping of jobs (events) to threads for a given design model to meet the specified timing requirements (whenever possible). The tool can adopt a structure shown in Figure 1.2.



Application Model
Resource Demands
Implementation Architecture
AUTOMATIC SYNTHESIS SUBSYSTEM
Schedulability Component
Analysis Model
Analysis Extractor Component
Candidate Implementation Model
Search Engine Component
Not Feasible
Feasible
Feasible Implementation Model
Optimization Component
Optimized Implementation Model

Figure 1.2: Automatic Synthesis Subsystem

1.3 Thesis Contribution

This thesis provides theories and solutions for two major open issues in real-time software design: scheduling with dual-level multi-tasking architecture and mapping of jobs (events) to threads to meet specific timing requirements. This work established the foundation for a promising approach for automatic synthesis of feasible implementation from real-time object-oriented software design model. My major contribution involves proposing new scheduling theories , designing solutions and algorithms for the feasibility and optimization problems, developing simulation tools, and quantitatively evaluating the performance of the solutions. More specifically, my contributions can be summarized as follows:

- Propose new theories on fixed priority scheduling with preemption threshold for a simplified model with independent jobs
- Propose algorithms for automatic generation of a feasible implementation model (including branch and bound algorithm, greedy heuristic algorithm, and simulated annealing)

- Propose solution for optimizing a feasible implementation model
- Design and implement a simulator including all algorithms mentioned above
- Evaluate the performance of our approach through simulation on randomly generated task sets
- Extend the schedulability analysis with preemption threshold to a generalized object-oriented model featured with end-to-end transaction, communication and resource sharing
- Extend the algorithm for automatic generation of feasible implementation model to the generalized model
- Extend the solution for optimizing implementation model to the generalized model

1.4 Thesis Organization

The organization of the rest of the thesis is as follows. Chapter 2 provides a brief review of related work in real-time scheduling, resource sharing theories and object-oriented real-time system design methods. Chapter 3 proposes a new fixed-priority scheduling model with preemption threshold and derives theories on scheduling the independent task model with the new scheduling policy. Based on the theories presented in Chapter 3, Chapter 4 addresses the issue of generating a feasible implementation model and optimizing the resulting implementation model to improve system performance. Simulation results are also provided to quantitatively measure the advantages of our approach. Chapter 5 illustrates the extension of our approach on a more general model which includes end-to-end transactions, synchronous/asynchronous communication and resource sharing. Finally, Chapter 6 concludes this thesis work and gives a foresight of the future work.

Chapter 2

Related Work

One of the most important goals of real-time system design is to produce a functionally correct system that predictably meets all deadlines. Scheduling policy, parameter assignment and resource allocation are important factors that determine the timing behavior of a real-time system. In this chapter, we give a brief review of related works on schedulability analysis, priority assignment, resource sharing protocols, and software design methodologies for real-time systems. The review focuses primarily on fixed priority scheduling models for hard-real-time systems.

2.1 General Scheduling Principles

In general, a scheduler can be classified to be either *static* or *dynamic*. Static schedulers create the job execution pattern, or *schedule*, off-line and the latter is then used to dispatch jobs at run-time. In contrast, dynamic schedulers determine the schedule on-line, based on specific job characteristics. Historically, the most popular scheduling technique for hard real-time designers is a non-preemptive static scheduling technique, called the *cyclic executive* approach [HG86, BS88]. A cyclic executive is a supervisory control program, which dispatches jobs in an application program of a real-time system based on a cyclic schedule. It typically has several schedules each consisting of a sequence of actions to be taken along with a fixed specification of timing constraints. These schedules are pre-computed and are executed repeatedly to have predictable execution. Long-term external conditions dictate which alternative is chosen for execution. A cyclic schedule is often divided into frames with equal duration. A periodic clock interrupt or some similar mechanism is used to initiate each frame. The individual frames are designed to finish execution within the pre-defined duration. Many systems use variations on the basic cyclic executive approach [Car84, SD88, AL77].

The primary advantages of the cyclic executive approach are that it is easy to understand, simple to implement, efficient, and predictable. Since the scheduling decision is made off-line, context switching between computations is very fast. Resource constraints and precedence constraints can also be embedded in the pre-computed schedule, which results in no overhead at run time for synchronization. However, several problems with this approach have been discovered [HG86, Loc92], such as the difficulties in designing the schedule, complications in maintaining the system, the inefficient use of resources and the inflexibility for adapting to changing system requirements. Thus, such methods have now largely been superseded by priority-driven scheduling approaches.

Priority-driven scheduling is in the category of dynamic scheduling. A priority-driven scheduler makes scheduling decisions at run-time based on the priority of each job. The priority is assigned to each job according to some policy. Based on the time for priority assignment, priority driven schedulers can be classified into two classes: *dynamic priority schedulers*, where job priorities are determined at run-time, typically when the job is invoked, and *fixed priority schedulers*, where job priorities are determined off-line and remain fixed at run-time. In practice fixed-priority schedulers are often employed since they offer a good balance between flexibility, efficient use of resources, and ease of implementation.

From another point of view, priority-driven schedulers can also be classified into *preemptive schedulers* and *non-preemptive schedulers*. With preemptive scheduling, the processing of any job can be interrupted by a higher priority job, while with non-preemptive scheduling, a job will not be interrupted during its processing.

2.2 Feasibility Analysis for Fixed Priority Scheduling

Determining whether a set of jobs is feasible, i.e., whether each job will always meet its deadline, is probably the most important issue in real-time system design. Historically, the feasibility of a set of jobs is determined in two ways: utilization-based test and worst-case response time analysis.

2.2.1 Utilization-based Test

The earliest work on system schedulability under fixed priority scheduling was published by Liu and Layland [LL73]. They analyzed feasibility of independent periodic job sets, scheduled using a fixed priority preemptive scheduler. They developed a sufficient condition for feasible job sets by devising an upper bound for utilization. If the utilization of a job set is less than this upper bound, then the job set is guaranteed to be schedulable. The upper bound is expressed as follows:

$$U_{max} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1}$$

where n is the number of jobs, C_i is the worst-case computation time of job i, and T_i is the period of job i. It is assumed that the deadline of a job equals its period, i.e., each job instance must finish before the arrival of the next instance of the same job.

The utilization-based test provides pessimistic results. It indicates that for job sets with large number of jobs, i.e., as n tends to infinity, if the job set utilization is less than or equal to 69.3% then it is schedulable. However, a job set with a higher utilization may be schedulable since this is a sufficient but not a necessary condition. Later work by Lehoczky, Sha and Ding [LSD89] shows many job sets with a large number of jobs and utilization approaching 90% are schedulable, and thus their schedulability is not adequately captured by the utilization based test of Liu and Layland.

Liu and Layland's identification of the *critical instant* is significant in the literature. For their independent job model, the critical instant is the point in time when all jobs are simultaneously released [LL73]. Their observation about the critical instant simplified the subsequent timing analysis and gave rise to the exact timing analysis. They also showed that the optimal priority assignment under their model is the "rate-monotonic" priority assignment (i.e., higher rate means higher priority), i.e., if a job set if feasible then it is also feasible under the rate-monotonic priority assignment.

2.2.2 Worst-case Response Time Analysis

With this approach, the schedulability of a job set is determined by computing the worst-case response times for all jobs. If the worst-case response times of all jobs are no more than their respective deadlines, then the job set is feasible. Joseph and Pandya [JP86] devised a method to find the worst-case response time of a given job τ_i under a preemptive scheduler, assuming sporadic jobs with minimum inter-arrival times and worst-case computation times. Their analysis assumes deadlines to be less than periods. Under this model, the worst-case response time of a job occurs when it is released simultaneously with all other jobs, i.e., the critical instant from Liu and Layland's work. Then the response time r_i can be iteratively computed using the following equation:

$$r_i = C_i + \sum_{\forall j, \pi_j > \pi_i} \left\lceil \frac{r_i}{T_j} \right\rceil \cdot C_j \qquad (2)$$

where C_i stands for the computation time for job τ_i, π_i stands for the priority of job τ_i, and T_j stands for the period of job τ_j.

Busy Period Analysis

The worst-case response time analysis has since then been extended to incorporate more general models including deadlines larger than periods, resource sharing, job release jitter, precedence constraints, job offsets or phases, varying priorities within a job, and more general job arrival patterns [Leh90, HKL91, TBW94]. While the techniques differ in the details, they are all based on the response-time analysis technique of Joseph and Pandya.

An important notion in the worst-case response time analysis is the notion of a critical instant and the notion of a *busy period* that was introduced in [Leh90]. To calculate the worst-case response time of a job, the busy period analysis essentially simulates the effect of scheduling under a worst-case scenario for the job. Following this approach, the worst-case response time for a job i is found by constructing a busy-period for τ_i, starting from the critical instant for τ_i. The busy period denotes a continuous interval of time during which the CPU is executing jobs at a priority no less than the priority of τ_i.

For independent job sets, the critical instant for a job occurs when it is released simultaneously with all other higher priority jobs. When a job's deadline is no more than its period, the busy period ends when the job finishes, and the length of the busy period is the same as its worst-case response time. However, when jobs have arbitrary deadlines, the busy period can include multiple instances of the job under consideration. Then, the busy period of job τ_i can be iteratively computed by using the following equation [Leh90, TBW94]:

$$w_i(q) = q \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left\lceil \frac{w_i(q)}{T_j} \right\rceil \cdot C_j \qquad (3)$$

In this equation, $w_i(q)$ denotes the length of a busy period for job τ_i, where q instances of τ_i are included in the busy period. The length of the busy period for job τ_i is given by:

$$W_i = \min_{q \in \{1,2,3,\ldots\}} w_i(q) :: w_i(q) \leq q \cdot T_i \tag{4}$$

where $w_i(q)$ is the smallest value of $w_i(q)$ that satisfies Equation 3. Essentially the equation states that during the busy period for τ_i, all instances of τ_i and higher priority jobs that arrive within the busy period must also be executed within the busy period. The busy period length is computed by iteratively computing $w_i(q)$ for $q = 1, 2, 3, \ldots$ using Equation 3 until $w_i(q) \leq q \cdot T_i$.

Then the worst-case response time of τ_i is determined by the longest response time of all instances that arrive and finish in the busy period. Let us denote $\mathcal{F}_i(q)$ (q^{th} finish time) as the smallest value of $w_i(q)$ that satisfies Equation 3, i.e., $w_i(q)$ converges to $\mathcal{F}_i(q)$. Since the q^{th} instance of τ_i arrives at $(q-1) \cdot T_i$, the worst-case response time for τ_i is given by:

$$\mathcal{R}_i = \max_{q \in [1,\ldots,m]} (\mathcal{F}_i(q) - (q-1) \cdot T_i) \tag{5}$$

Even though the analysis described above is done for preemptive scheduling, the same technique, with minor modifications, can be used for non-preemptive scheduling as well [GRS96]. However, there are two major differences. First, since jobs cannot be preempted while started, lower priority jobs also have an effect on a job's response time – this effect is called the *blocking* effect. In the worst-case, a lower priority job may have just started execution prior to the critical instant. Second, once a job gets the CPU, it cannot be preempted by any higher priority job until it finishes execution.

The blocking time from lower priority jobs is easy to incorporate. A job may be blocked by only one lower priority job. In the worst case, this job would have started execution just before the critical instant. Thus, the worst-case blocking time for job τ_i, denoted as B_i is given by:

$$B_i = \max_{\forall j, \pi_j < \pi_i} C_j \tag{6}$$

Thus, Equation 3 can be modified for non-preemptive scheduling as follows:

$$w_i(q) = B_i + q \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{w_i(q) - C_i}{T_j} \right\rfloor\right) \cdot C_j \tag{7}$$

Note that the term for interference from higher priority jobs is modified to include job arrivals up to (and including) time $w_i(q) - C_i$, i.e., when the q^{th} instance job τ_i starts executing.

2.3 Priority Assignment

An important concern in fixed priority scheduling is the assignment of priorities to jobs. We have already mentioned that Liu and Layland [LL73] showed that the optimal priority assignment for independent periodic job sets with deadlines equal to periods is the rate-monotonic (RM) assignment. RM assignment assigns higher priority to jobs with smaller periods. Leung and Whitehead [JJ82] extended their results to include jobs where the deadline can be less than the period, and proved that the deadline monotonic (DM) assignment is optimal. Similar to RM assignment, DM assignment assigns higher priority to jobs with smaller deadlines. Lehoczky [Leh90] points out that neither RM nor DM assignment is optimal for general job sets (i.e. where deadlines can be larger than periods). Finally, Audsley [Aud91a] solves this problem by devising an optimal priority ordering algorithm with complexity of $O(n^2)$.

Audsley's algorithm works by starting assignment of priorities from the lowest priority. The algorithm works by dividing the job set into two parts: a sorted part, consisting of the assigned lower priority jobs, and an unsorted part, holding the remaining unassigned higher priority jobs. The algorithm keeps moving jobs from the unsorted part to add on top of the sorted part if the chosen job is schedulable with the current priority level. The process continues until either all jobs are in the sorted part, which means a feasible assignment is found, or when no job can be moved from the unsorted part to the sorted part, indicating that the job set is not feasible.

While Audsley's algorithm was developed for preemptive scheduling, the same algorithm can be used for non-preemptive scheduling as well [GRS96]. It was also shown that the optimality results of the DM and RM priority assignments from the preemptive model do not apply to the non-preemptive model. Finally, [GRS96] also shows that DM is optimal for job sets in which the deadline of each job is no more than its period, if a larger deadline implies a larger or equal computation time.

2.4 Resource Sharing Protocols

The above schedulability analyses assume independent job sets which implies that jobs may not compete for shared resources. Allowing access to shared resources creates a scheduling scenario in which the execution of a job can directly affect the runnability of other jobs. *Priority inversion* may happen when a *low* priority job blocks a *high* priority job and the low priority job holds the resource that is needed by the high priority job. Priority inversion can be unbounded if *medium* priority jobs are allowed to preempt the low priority job, while it holds the shared resource. For hard real-time systems, determining the schedulability of a set of jobs that require mutually exclusive access to some or all of a set of resources is an NP-hard problem [Mok83]. To avoid the intractability of the optimal solution, many approaches have been proposed to provide sub-optimal solutions. To be applicable to hard real-time systems, these approaches must have two important attributes: *predictability*, which means resource allocation decisions should be known pre-runtime, and *boundedness*, which means the execution time of a job should be bounded and the bound can be calculated pre-runtime.

The resource control techniques in the literature can be classified into two classes: predictable and non-predictable. Non-predictable approaches are not applicable to hard real-time systems and are therefore precluded from our consideration. Predictable approaches can be further divided into two groups: blocking and non-blocking. Non-blocking approaches include a static off-line schedule, which is out of our interest, and a run-time scheduling mechanism called *four-slot mechanism* proposed by Simpson [Sim90], which is associated with the data age problem. Therefore, non-blocking approaches are also precluded from our review. Preventing preemptions while a job is in its critical section can always guarantee mutual exclusive access to shared resources. This approach can be looked as an extension of HLP (Highest Locker Protocol) reviewed in this section. Several resource sharing protocols are reviewed in this section in the context of uni-processor environment. They can be categorized as predictable blocking preemptive approaches. For other resource control protocols, Audsley gives a good review in [Aud91b].

2.4.1 Priority Inheritance Protocol

Priority Inheritance Protocol (PIP) is proposed by Sha, Rajkumar and Lehoczky [SRL90] to solve the uncontrolled priority inversion problem. The basic idea is that when a job τ_j blocks one or more higher priority jobs, it raises its running priority and executes its critical section at the highest priority level of all the jobs it blocks irrespective of its original priority assignment. After exiting its critical section, job τ_j resets its priority to the original priority assignment. This policy introduce two sources of blocking: *direct blocking*, when a higher priority job is trying to access a locked resource, and *push-through blocking*, when a medium priority job is prevented from preempting lower priority job τ_j and therefore avoids priority inversion. The blocking time for a job τ_i with this protocol is bounded by $min(m, n)$ critical sections of lower priority jobs, where n is the number of lower priority jobs that are able to block τ_i and m is the number of resources used by lower priority jobs and can have a higher run priority than the priority of τ_i.

PIP suffers from two major problems, namely deadlock and chained blocking. Deadlock may happen when τ_i locks R1 but gets preempted by τ_j before it can lock R2, and τ_j locks R2 and requests R1. Chained blocking may happen when τ_3 locks R3, then τ_2 preempts τ_3 and locks R2, and then τ_1 preempts τ_2 and requests R3 and R2. These problems are due to the fact that with PIP, a job is able to lock a free resource at any time, regardless of its priority relationship to other jobs that have already locked resources.

2.4.2 Priority Ceiling Protocol

To avoid the formation of deadlocks and chained blocking, Sha, Rajkumar and Lehoczky [SRL90] proposed another protocol, Priority Ceiling Protocol (PCP). By assigning a priority ceiling to each critical section (or semaphore), which equals the highest priority of the jobs that may request this resource (or use this semaphore), PCP ensures a strict priority ordering of critical section executions. A job executes at its original priority unless it blocks a higher priority job, in which case it will inherit the priority of the blocked job. When a job τ_j preempts the execution of another job which is in its critical section, and tries to enter its own critical section, its priority should be higher than all the priority ceilings of the preempted critical section. Otherwise, it will be blocked and the job that blocks τ_j inherits its priority. Besides the two sources of blocking in PIP, PCP introduces a new source of blocking, *ceiling blocking*, which arises when a job is denied access to a free resource due to the higher ceiling priority of a locked resource.

The prevention of deadlock and chained blocking in PCP is not costless. PCP may deny a job's request to a free resource even if it could not possibly lead to deadlock or chained blocking. This results in pessimism in terms of resource access. This problem can not be circumvented without considering the detail of resource access in the critical section (such as the sequence of resource requests and how long a resource is required).

2.4.3 Semaphore Control Protocol

An extension of PCP named Semaphore Control Protocol (SCP) was proposed by Rajkumar, Sha, Lehoczky and Ramamithram [RSLR88]. The only difference between SCP and PCP is the conditions to grant a request for resource. In PCP, the priority of the requesting job must be strictly higher than the ceiling of all currently locked resources. Besides the PCP granting condition, there are two additional situations that the request will be granted in SCP:

(1) if the priority of the requesting job equals the highest ceiling of currently locked resources and the current critical section of the requesting job will not attempt to request resources with ceiling priority equal to its own priority, then the request is granted.

(2) if the priority of the requesting job equals the ceiling priority of the requested resource R, and no other currently active job will request R during the execution of their current critical sections, then the request to R is granted.

This extension to PCP still prevents deadlock and chained blocking. The authors proved that the above rules for granting resources are sufficient and necessary. However, SCP still suffers from the same disadvantage as as we mentioned above for PCP.

2.4.4 Dynamic Priority Ceiling Protocol

The above approaches are based on fixed priority preemptive scheduling. Chen and Lin [CL90] extend PCP for dynamic priority scheduling by combining the ceiling priority and dynamic priority scheduling algorithm such as EDF. To achieve this goal, it is necessary to maintain an effective job set containing one instance of all the jobs in the system. The priority of a job is kept updated so that it always equals the dynamic assigned priority of its currently active execution. Then, the ceiling priority of a resource (or the corresponding semaphore) is set to the highest priority of the jobs in the effective job set that may request the resource. The dynamic job priority makes the ceiling priority of resources also dynamic.

Chen and Lin proved that both deadlock and chained blocking are prevented in DPCP and the maximum blocking time a job may suffer is exactly as in PCP, which is the longest critical section of all jobs with longer periods. The major disadvantage of DPCP is that the implementation cost is too high. Re-evaluating ceiling priorities of resources should be done when a job is released or finishes. In their paper, they proposed a possible implementation to reduce this overhead.

2.4.5 Stack Resource Policy

Baker refined the PCP and proposed a stack-based resource allocation policy named Stack Resource Policy (SRP) [Bak90]. SRP supports multi-unit resources and dynamic priority scheduling. It also reduces memory requirements by sharing runtime stack among jobs.

SRP introduces a notion of *preemption level* as a measure of how jobs can preempt each other. In SRP, each job is assigned a priority (that can be static or dynamic) and a static preemption level. A job is not allowed to preempt another unless its preemption level is greater. The preemption level is assigned based on the relative deadline of the job: the shorter the deadline, the greater the preemption level.

The ceiling for each resource is defined as the maximum preemption level among those jobs that may be blocked on the resource (i.e., requesting more than the available units of that resource). When a job τ_i is released, it can preempt the running job if and only if:

1. the entire resources required by τ_i are available, i.e., its preemption level is less than the ceiling of all resources it requires.
2. the resources required to complete all jobs that can preempt τ_i are also available, i.e. all jobs with higher preemption level have their preemption level less than the ceiling of all resources they require.

SRP can achieve identical performance as PCP in controlling priority inversion. Deadlock and chained blocking are prevented and the maximum blocking time is bounded to the longest critical section of all the lower priority jobs.

The stack sharing aspect of this work is also interesting. It reduces the need for memory space compared to stack allocation on a per-job basis. The saving is related to the number of jobs and preemption levels. The number of separated stack spaces equals the number of different preemption levels while using SRP, since at most one job from each preemption level can be active at any time.

Both SRP and DPCP permits jobs to have dynamic priority. However, SRP will result in a more efficient implementation since the ceiling priority of a resource is based on preemption levels, which are static, of the jobs that may be blocked on the resource. It does not need to re-evaluate the ceiling of resources every time there is a change in priority assignment while in DPCP the re-evaluation is necessary.

2.4.6 Highest Locker Protocol

Highest Locker Protocol (HLP) [KRP+93] is a variation of PCP that sets a job's priority to be equal to the ceiling priority of the resource it granted.[1] This refinement of PCP reduces context switches and reduces the complexity at runtime. For the job that is granted the resource, HLP prevents preemption from jobs with priority less than or equal to the ceiling priority of the resources. This increases the response time of jobs not requesting any locked resources. However, the job holding the resource will respond quicker.

2.4.7 Reservation Protocols

The above protocols are following the same approach: change the priority of the job to control priority inversion. A totally different approach, namely Reservation Protocols (RP), which do not alter job priorities, is presented by Babaoglu et al [BMS90]. This approach uses a reservation graph, in which *priority relation* is defined to indicate higher priority jobs and *wait-for relation* is defined to indicate the jobs that hold the resources a job is waiting for. A loop, or π-*cycle*, indicates a priority inversion. Methods are proposed to avoid such cycles, and hence avoid priority inversions.

RP is found to be deadlock free. The blocking time for each job equals twice the worst-case execution time of all higher priority jobs.

2.5 Design Methodologies

In the early days, real-time software designers developed systems in an ad-hoc manner based on their intuition and some heuristics from their experiences. Over the years, many design methods have been proposed for real-time system development, for instance, JSD, MASCOT, RTSA, DARTS, CODARTS and OCTOPUS [AKZ96, Gom93]. Many recent

[1]This protocol is outlined in [RSL89] with the name Ceiling Semaphore Protocol. It is also called Priority Ceiling Emulation or Immediate Priority Ceiling Protocol.

design methods are based on object-orientation, for example, HRT-HOOD [BW94], OCTOPUS [AKZ96], CODARTS [Gom93], the ROOM method [SGW94], and the Shlaer-Mellor method [SM]. We choose some to give a quick review.

2.5.1 HRT-HOOD

HRT-HOOD (Hard Real-Time Hierarchical Object Oriented Design) [BW94] was designed for hard real-time systems, and is heavily influenced by the development in real-time scheduling theory. It supports three basic software engineering principles: (a) abstraction, information hiding and encapsulation, (b) hierarchical decomposition, and (c) control structuring. Object attributes have been added so that designers can specify the real-time characteristics (e.g. deadline, period, worst-case execution time). An object in HRT-HOOD has *static* and *dynamic* properties. The static properties describe the object's interface. The dynamic properties describe the effect of an operation through *sequentiai flow* and *parallel flow*. There are five basic object types: *passive*, *active*, *protected*, *cyclic*, and *sporadic*. An end-to-end transaction is first designed as a single object, cyclic or sporadic and then decomposed into a set of terminal precedence (*before* and *after* relationship) constrained objects.

HRT-HOOD is one of the first structured design methods which seriously attempts to address the issue of designing predictable real-time systems. The abstractions of HRT-HOOD directly map to the concepts in real-time scheduling theory, thus making the design analyzable for timing requirement. The approach followed in HRT-HOOD works well when the system can be decomposed into a relatively small number of periodic and sporadic jobs, and when each job has a relatively coarse granularity. Both these limitations relate to the nature of the underlying scheduling model, i.e., preemptive multi-tasking. Preemptive multi-tasking incurs relatively high costs in context-switching and these costs become significant when job granularity is small (since the context switching overhead is amortized over a smaller execution time) and when there is a large number of jobs (more jobs would typically result in increased context switching). In addition, there is a per-job memory cost, largely due to the need to maintain a separate stack for each job. While these run-time costs may be irrelevant in most non-embedded environments, they play a significant role in many embedded real-time systems since such systems often tend to be resource constrained.

2.5.2 MetaH

MetaH [VB93, Cen], an architecture description language, and associated tool set are designed to specify, analyze, and automatically assemble software for real-time, fault-tolerant, secure, multi-processor systems. One important goal is to provide design-time analysis that accurately characterizes the behavior of the actual implementation. It allows system architects to specify software and hardware architectures for computer control systems. Developers use MetaH to specify how code modules written in traditional languages are combined to form an application, to specify the structure of a particular hardware target system, and to specify how the software is allocated to hardware. In MetaH, a process is the fundamental unit of scheduling, allocation to hardware processors, and fault and security containment. A process may communicate with other processes through input and output ports, which corresponding to buffer variables within the source code. Processes are scheduled using preemptive fixed priority discipline.

MetaH provides both graphical and textual syntax and tools that allows a specification to be viewed and edited interchangeably in either format. The tool set also includes code generator and assembler, application builder, timing analyzer. Using these tools, an executable image can be automatically generated. The use of exact schedulability analysis enables the generation of a timing report for the designer, which helps a lot in designing hard real-time system. MetaH has been used to develop, analyze and execute demonstrations of portions of a number of application. However, the priority assignment used in MetaH, which is rate monotonic, restricts its application since it is only optimal for independent jobs with deadlines equal periods.

2.5.3 ROOM and ObjecTime

ROOM (Real-time Object Oriented Modeling) [SGW94], a modeling language, and ObjecTime Developer, a CASE tool that provides a fully integrated development environment to support the ROOM methodology with features such as graphical and textual editing for actor construction and C++ code generation from the model, originated in the research and development lab in Bell-Northern Research. It has been in practical use since 1989 and has been applied successfully to a wide variety of large and small industry projects. As UML gains its popularity in industry, ObjecTime has cooperated with Rational Software to develop UML-RT, which uses UML's in-built extensibility mechanisms to integrate ROOM concepts within UML. UML-RT and the code generation technology of ObjecTime Developer has been integrated into Rational Rose, in the new product Rational Rose Real-Time.

ROOM uses active objects, called *actors*, to model a real-time system. These are encapsulated, concurrent objects that communicate asynchronously by sending and receiving messages through distinct interfaces called *ports*. Priorities are used to identify the significance of messages. The behavior of actors is modeled using *ROOMcharts*, which is an extended finite state machine that may include composite states as well as guard conditions. It is based on the statechart formalism [Har87]. Sending an event to an actor may initiate the execution of an action. The action specification is thought of as a fine-grained detail and thus, can be specified using a programming language, such as C++. Code involved with message transfers is also specified within actions.

ROOM provides a generic run-time system, which is also incorporated in the ObjecTime Developer tool set. The generic run-time system models threads as event handlers. ROOM uses a *run-to-completion* paradigm for event execution in a thread, i.e., if an event is being processed in a thread when a higher priority event arrives for execution in the same thread, the latter will not get processed until the former has completed. Actors are also mapped to threads. Therefore, the thread that an actor resides in should handle all events associate with this actor. However, thread priorities within ObjecTime are statically managed, which leads to unbounded priority inversion in a multi-threaded implementation. ROOM's generic run-time system also includes a dedicated thread, which is used to insert periodic or timer messages to actors.

ROOM has features that, on one hand, enable high-level modeling of complex real-time applications (e.g., the use of hierarchy in structure diagrams and state machines, layering, dynamic structures and inter-connections) and, on the other hand, allow fine-grained details to be specified (e.g., use of a programming language, like C++, to specify the action to be taken to handle an event). However, ROOM and ObjecTime Developer provide little support for schedulability analysis.

Chapter 3

Scheduling Theories for A Simplified Model with Independent Job

In this chapter, we introduce a new scheduling model with the notion of *preemption threshold*. This new model subsumes both preemptive and non-preemptive schedulers as special cases. With this model, the schedulability of a set of jobs is improved when compared with both preemptive and non-preemptive schedulers. This model enables our automatic synthesis process illustrated in later chapters. Subsequently, we present the schedulability analysis for the new scheduling model over a simplified model where each job (action) is independent, including equations for calculating the worst-case response time.

3.1 Scheduling with Preemption Threshold

Since the pioneering work of Liu and Layland [LL73], much work has been done in the area of real-time scheduling, and in the analysis techniques to *a priori* predict schedulability of job sets under a specific scheduling discipline. In particular, significant progress has been made in schedulability analysis of job sets under fixed priority preemptive scheduling [HKL91, JP86, LSD89, Leh90, TBW94]. The benefits of fixed priority preemptive scheduling include relatively low run-time overheads (as compared to dynamic priority schemes, such as Earliest Deadline First) and the ability to support tighter deadlines for urgent jobs (as compared to non-preemptive scheduling).

While preemptability is often necessary in real-time scheduling, it is fallacious to assume that it always results in higher schedulability. Indeed, it can be shown that, in the context of fixed priority scheduling, preemptive schedulers do not dominate non-preemptive schedulers, i.e., the schedulability of a set of jobs under non-preemptive scheduling does not imply the schedulability of the same set of jobs under preemptive scheduling (and vice-versa). Moreover, preemptive schedulers have higher run-time overheads as compared to non-preemptive schedulers.

In this section, we propose a generalized model of fixed priority scheduling that integrates and subsumes both preemptive and non-preemptive schedulers. The model uses the notion of *preemption threshold*, which was introduced by Express Logic, Inc. in their ThreadX real-time operating system to avoid unnecessary preemptions [Lam]. In our new scheduling model, each job has a preemption threshold, in addition to its priority. In essence, this results in a dual priority system. Each job has a regular priority, which is the priority at which it is queued when it is released. Once a job gets the CPU, its priority is raised to its preemption threshold. It keeps this priority, until the end of its execution. For recurring jobs, this process repeats each time the job is released.

The preemption threshold scheduling model is designed for modeling and analyzing the dual-level scheduling in practical real-time software design and can be used to get the benefits of both preemptive and non-preemptive scheduling. By choosing the preemption threshold of a job to be higher than its priority, a job avoids getting preempted by any job that has a priority lower than its preemption threshold. By varying the preemption thresholds of jobs, the desired amount of non-preemptability may be achieved. In this way, the preemption threshold model may be viewed as introducing non-preemptability in a controllable manner. A suitable setting for the preemption thresholds can thus be used to get "just enough preemptability" needed to meet the real-time responsiveness requirements; thereby eliminating run-time overheads arising from unnecessary preemptability in the scheduling model. It is also easy to see that both preemptive and non-preemptive scheduling are special cases of scheduling with preemption threshold. If the preemption threshold of each job is the same as its priority, then the model reduces to pure preemptive scheduling. On the other hand, if the preemption threshold of each job is set to the highest priority in the system, then no preemptions are possible, leading to fixed priority non-preemptive scheduling.

Task	Comp. Time C_i	Period T_i	Deadline D_i
τ_1	20	70	50
τ_2	20	80	80
τ_3	35	200	100

Table 3.1: An Example Job Set.

3.1.1 A Motivating Example

Before we delve into a theoretical treatment of the new scheduling model, it is instructive to take a look at a simple example that shows how schedulability can be improved with this new scheduling model. We consider a job set with 3 independent periodic jobs, as shown in Table 3.1. Each job is characterized by a period (T_i), a deadline (D_i) and a computation time (C_i).

The scheduling attributes for each job include its priority (π_i) and its preemption threshold (γ_i). Assuming fixed-priority scheduling, the optimal priority ordering for these jobs is deadline monotonic ordering with both preemptive scheduling [LW82] and non-preemptive scheduling [GRS96]. Under this priority ordering[1], the worst-case response times for the jobs are shown in Table 3.2. We can see that τ_3 misses its deadline under preemptive scheduling, while τ_1 misses its deadline under non-preemptive scheduling. Since the priority ordering is optimal, this implies that the job set is not schedulable under either fixed-priority preemptive scheduling, or fixed-priority non-preemptive scheduling.[2]

When we use preemption threshold, we can make the job set schedulable by setting the preemption threshold for τ_2 as 3, and τ_3 as 2. By setting the preemption threshold of τ_3 to 2, we allow it to be preempted by τ_1, but not by τ_2. This effectively improves the response time of τ_1 (as compared to the non-preemptive case) since it can no longer be blocked by τ_3. At the same time, it improves the response time of τ_3 (as compared to the preemptive case) since it cannot be preempted by τ_2 once τ_3 has started running. The resultant response times are also shown in Table 3.2[3].

[1]Throughout this thesis, we use higher numbers to denote higher priorities.

[2]Note that a slight modification to this example will show that the feasibility under preemptive scheduler does not imply feasibility under non-preemptive scheduler (by changing the deadline of τ_3 to 120), and vice versa (by changing the deadline of τ_1 to 60).

[3]The worst-case response time of jobs with preemption threshold is done using the theories we present later in this chapter.

Task	π_i	WCRT Preemptive $\gamma_i = \pi_i$	WCRT Non-Preemptive $\gamma_i = 3$	γ_i	WCRT Preemption-Threshold
τ_1	3	20	**55**	3	40
τ_2	2	40	75	3	75
τ_3	1	**115**	75	2	95

Table 3.2: Worst-case Response Times for Jobs under Different Schedulers

Figure 3.1 illustrates the run-time behavior of the system with preemption threshold, and how it helps in improving schedulability. In the figure, the arrows indicate arrival of jobs. The figure shows the CPU scheduling starting from a critical instant of τ_3, which occurs when all jobs arrive simultaneously (time 0). Our theories presented later shows that it is the worst-case response scenario for τ_3 and thus the response time is actually the worst-case response time. We can see that at time 70, a new instance of τ_1 arrives. Since the priority of τ_1 is higher than the preemption threshold of τ_3, τ_3 is preempted. At time 80, a new instance of τ_2 arrives. It can not preempt the execution of τ_1. However, at time 90, when τ_1 finishes, a pure preemptive scheduler would have run τ_2, delaying τ_3. In contrast, by setting the preemption threshold of τ_3 to 2, we have τ_3 scheduled at time 90 under our scheduling model. This effectively improves the worst-case response time of τ_3, making it schedulable. Note however that this also adds blocking time to τ_2 (as compared to the preemptive case), which increases its worst-case response time, but does not affect its schedulability in this example.

The use of preemption thresholds also reduces the run-time overheads associated with preemptions and the associated context-switches. This is due to the introduction of some non-preemptability into the scheduling model. We simulated the execution of jobs in this example for one LCM length (i.e., 2800 time units). When all jobs were released simultaneously at time 0, we find that preemptive scheduling results in 17 preemptions, while with preemption thresholds we get 8 preemptions. If we stagger the release times, such that τ_3 is released at time 0, τ_2 at time 1, and τ_1 at time 2, then the number of preemptions with preemptive scheduling is 30, while using preemption thresholds reduces it to 10.

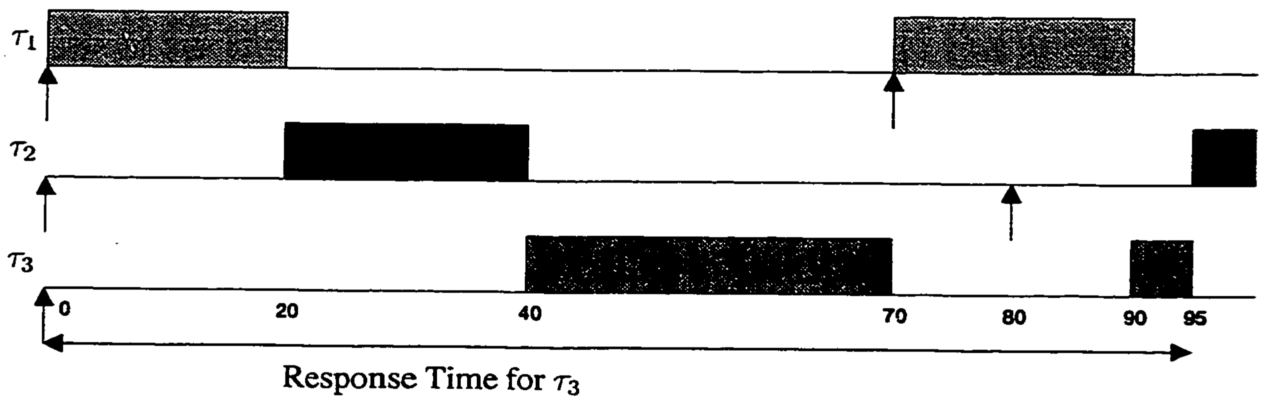


Figure 3.1: Run-time Behavior with Preemption Threshold

3.1.2 Describing the Simplified Model

We consider an event-based model consisting of a set of events and the computations (actions) triggered by the events. For simplicity, we assume that events arrive either periodically (i.e., with a fixed inter-arrival time) or sporadically (i.e., with a minimum inter-arrival time). We use the term *job* to refer to an event and its corresponding computation. Thus, we describe our simplified model as a set of N independent periodic or sporadic jobs $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each job τ_i is characterized by a 3-tuple $\langle C_i, T_i, D_i \rangle$, where C_i is its computation time, T_i is its period (or minimum inter-arrival time), and D_i is its relative deadline. We assume that (a) jobs are independent (i.e., there is no blocking due to shared resources), (b) jobs do not suspend themselves, other than at the end of their computation, and (c) the overheads due to context switching, etc., are negligible (i.e., assumed to be zero).

Each job is also characterized by its scheduling attributes, which include a (nominal) priority $\pi_i \in [1, \ldots, N]$ and a preemption threshold $\gamma_i \in [\pi_i, \ldots, N]$. These attributes are not known to begin with, and must be derived to meet the timing requirements. We assume that these scheduling attributes are determined off-line, and are fixed at run-time. Finally, each job is also mapped to a *thread* in the implementation. We will use the notation $\psi(i)$ to denote the thread assignment for job τ_i. Again, this assignment is not known to begin with, but is determined off-line, and remains fixed during run-time (i.e., each arrival of the job is processed in the same thread).

We formally define an implementation model to include an assignment of scheduling attributes and mapping from jobs to threads. For the purpose of this chapter we will assume that an implementation model is already given.

Definition 3.1 (Implementation Model) *An implementation model, denoted as $\mathcal{I}$, for a given model $\mathcal{M}$ containing a set of N independent periodic or sporadic jobs $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, is defined by a 3-tuple: $\langle \Psi, \Pi, \Gamma \rangle$, where*

- $\Pi : \mathcal{M} \longrightarrow [\pi_1, \ldots, \pi_N]$ *is a priority assignment for the jobs,*
- $\Gamma : \mathcal{M} \longrightarrow [\gamma_1, \ldots, \gamma_N]$ *is a preemption threshold assignment for the jobs, and*
- $\Psi : \mathcal{M} \longrightarrow [\psi(1), \ldots, \psi(M)]$ *is a thread assignment for the jobs*

Also, for simplicity we will assume that each job is assigned to its own thread. We call this the "nominal thread assignment" as defined below. With this simplification, we do not need to worry about the two-level scheduling architecture at this time. Later in this chapter we will show that with certain restrictions on thread assignment, the response times of jobs can be made independent of the thread assignment for jobs, which facilitates the modeling and analyzing the two-level scheduling architecture.

Definition 3.2 (Nominal Thread Assignment) *A thread assignment is called nominal if each job is mapped to its own thread. That is, the following predicate is true in the nominal thread assignment. We will use the notation $\hat{\Psi}$ to denote the nominal thread assignment.*

$$(\forall \tau_i) \quad \psi(i) = \hat{\Psi}(\tau_i) = i$$

3.2 Schedulability Analysis

Now, we consider the problem of assessing schedulability of a set of jobs under the preemption threshold scheduling model, assuming that the scheduling attributes are already known. Let Π be a priority assignment, and Γ be a preemption threshold assignment. Let $\mathcal{R}_i(\Pi, \Gamma)$ denote the worst-case response time of a job τ_i under the given assignments. Then, the schedulability of a set of jobs is defined by the following boolean predicate:

$$sched(\mathcal{T}, \Pi, \Gamma) \overset{\text{def}}{=} (\forall i :: 1 \le i \le n)\ \mathcal{R}_i(\Pi, \Gamma) \le D_i \tag{8}$$

3.2.1 Overview

By definition, the schedulability of a set of jobs is determined by the worst-case response time of each job. The response time analysis employed in this section is an extension of the well-known level-i busy period analysis [HKL91, JP86, LSD89, Leh90, TBW94], in which the response time is calculated by determining the length of the busy period, starting from a critical instant. The busy period at level-i is defined as a continuous interval of time during which the processor is continuously executing jobs of priority i or higher.

We modify the traditional level-i busy period analysis to account for the changing priorities of jobs. Critical instant and level-i busy period are redefined for this purpose. Critical instant, by definition, is a instant that leads to the scenario, in which the worst-case response time of a job will occur. In our model, while considering a specific job τ_i, its execution may be suspended by execution of both lower priority jobs and higher priority jobs. For clarity, we refer the suspension caused by lower priority jobs as *blocking* and the suspension caused by higher priority jobs as *interference*. Since blocking and interference plays a critical role in the response time of any specific jobs, it is necessary to analyze them before defining critical instant and level-i busy period in our new scheduling model.

The introduction of preemption threshold makes blocking and interference in our model different from both preemptive and non-preemptive scheduling. A job τ_i may suffer blocking before it starts from lower priority jobs only if their preemption thresholds are higher than π_i while in non-preemptive scheduling all lower priority jobs may contribute to blocking. In a similar way, only jobs with higher priority than γ_i may cause interference after τ_i starts executing while in preemptive scheduling all higher priority jobs will interfere.

For simplicity and clarity, we explicitly define and compute both the q^{th} start time for a job τ_i (denoted by $\mathcal{S}_i(q)$) and the q^{th} finish time (denoted by $\mathcal{F}_i(q)$). After we get the q^{th} finish time by the modified busy period analysis, the worst-case response time for τ_i is calculated as follows:

$$\mathcal{R}_i = \max_{q\in[1,\ldots,m]} (\mathcal{F}_i(q) - (q-1)\cdot T_i) \tag{9}$$

where T_i represents the period or minimum arrival interval of τ_i.

3.2.2 Blocking Analysis

In our scheduling model, when job τ_i is the job with highest nominal priority queued to be processed, if a lower priority job is running with an effective priority higher than $\pi(\tau_i)$(i.e., its preemption threshold is higher than $\pi(\tau_i)$), it cannot be preempted by τ_i, leading to blocking time in the response time of τ_i. Now we analyze the blocking that τ_i may suffer to find out the worst case or the upper bound. In our analysis, we define **Blocking Range** and **Active Job** as follows.

Definition 3.3 (Blocking Range) *The blocking range of a job τ_i is defined as the range of priorities given by $[\pi_i,\gamma_i]$.*

Definition 3.4 (Active Job) *A job is called active if it has started execution, but is not finished yet.*

Lemma 3.1 *There is no overlapping of blocking ranges between the active jobs at any instant of time.*

Proof: By contradiction. Suppose τ_i and τ_j are both active at a specific time, and their blocking ranges overlap. Without loss of generality assume that τ_i started execution first. Then, for τ_j to start running before τ_i finishes, it must be the case that $\gamma_i < \pi_j$. That is $\pi_i \leq \gamma_i < \pi_j \leq \gamma_j$. Thus, the blocking ranges can not overlap. □

Lemma 3.2 *A job τ_i can be blocked by at most one lower priority job τ_j.*

Proof: A new arriving job τ_i will preempt a lower priority active job τ_j unless π_i falls in the blocking range of τ_j, i.e., $\pi_j < \pi_i \leq \gamma_j$. From Lemma 3.1, we know that blocking ranges of active jobs will not overlap. Therefore, τ_i will only fall into at most one of these blocking ranges, i.e., be blocked by the owner of that blocking range. Furthermore, it is easy to see that any lower priority jobs that have not started execution before the arrival of τ_i, as well as lower priority jobs that arrives after τ_i will not start before τ_i starts. Therefore, they will not block τ_i. □

Lemma 3.2 shows that in computing the blocking time for a job τ_i, we need to consider blocking from only one lower priority job τ_j such that $\gamma_j \geq \pi_i$. Therefore, the maximum blocking time of a job τ_i, denoted by $B(\tau_i)$, is given by:

$$B(\tau_i) = \max_{\forall j, \gamma_j \geq \pi_i > \pi_j} C_j \qquad (10)$$

3.2.3 Defining Critical Instant and Level-i Busy Period

Lehoczky's definition of critical instant and level-i busy period is no longer valid in our scheduling model due to the introduction of preemption threshold. We extend the definition and prove that worst-case response time can be calculated in our modified level-i busy period. We first define the effective priority of a job. We can consider each job to be in one of three states: 'ready', 'running', or 'preempted'. A job instance is 'ready' when it first arrives. When it gets the CPU for the first time, its state changes to 'running'. If it gets preempted, its state changes to 'preempted'. A job instance cycles between 'preempted' and 'running' states until it finishes and leaves.

Definition 3.5 (Effective Priority) *The effective priority of a job equals its (nominal) priority if it is in the state 'ready,' and equals its preemption threshold if it is in the 'preempted' or 'running' state.*

A formal definition of level-i busy period is given in 3.6 using the concept of effective priority. Note that in a level-i busy period, there may be several instances of the same job with priority i executed.

Definition 3.6 (Level-i Busy Period) *A level-i busy period is a time interval $[a, b]$ within which only jobs with effective priority i or higher are processed, but no jobs with effective priority i or higher are processed in $(a - \epsilon, a)$ or $(b, b + \epsilon)$ for sufficiently small $\epsilon > 0$.*

Lemma 3.3 *In a level-i busy period, only the first instance of job τ_j whose priority $\pi_j = i$ will suffer blocking from jobs with lower priority. Any instance after the first instance will not suffer from blocking. Furthermore, at most only one instance of a job τ_k with $\pi_k < \pi_j$ and $\gamma_k \geq \pi_j$ may contribute to blocking.*

Proof: It is easy to see that the start of processing first instance of job τ_j ensures the finish of all active instances of jobs with lower priority and higher preemption threshold. Since in a level-i busy period, there is always at least one job instance with effective priority higher or equal to i, no instance of job with lower priority will start. Therefore, after the start of processing first instance of job τ_j, the following instance of τ_j will not suffer from blocking.

Furthermore, using Lemma 3.2, we can easily see that at most only one instance of a job τ_k with $\pi_k < \pi_j$ and $\gamma_k \geq \pi_j$ may contribute to blocking. □

Based on the lemma above, we come up with the most important theorem in our schedulability analysis, Theorem 3.1. Here, we use I_i to indicate the phasing of τ_i to some fixed time origin.

Theorem 3.1 *The worst-case response time for a job τ_i occurs during a busy period initiated by a critical instant, at which all jobs τ_j with $\pi_j \geq \pi_i$ have $I_j = I_i = 0$ and the longest job τ_k with $\pi_k < \pi_i$ but $\gamma_k \geq \pi_i$ has $I_k = 0 - \epsilon$ for sufficiently small $\epsilon > 0$.*

Proof: Let [0,b] be a busy period, and suppose τ_i arrives at $I_i > 0$. Only jobs with higher effective priority than τ_i are processed during [0,I_i). Thus, if I_i were changed to any value in [0,I_i), each instance of τ_i in [0,b] would finish at the same time, thus increasing each of the τ_i response times. The maximum response occurs when I_i is as small as possible, namely $I_i = 0$.

If $I_j > 0$ for some τ_j with $\pi_j \geq \pi_i$, then it is obvious that reducing I_j serves to increase (or leave unchanged) the processing requirements of τ_j during [0,t] for every $t \in [0, b)$, thus increasing (or leave unchanged) the response time of instances of τ_i. The longest response time is achieved by setting I_j to their smallest value, that is, $I_j = 0$.

As we proved in Lemma 3.3, there will be at most one job with lower priority and higher preemption threshold can block τ_i. It is easy to see that the worst case is the longest one starts at $0 - \epsilon$. □

To calculate the worst-case response time of a job, the busy period analysis essentially simulates the effect of scheduling under a worst-case scenario for the job. The busy period for job τ_i is constructed by starting from a *critical instant (time 0).* The critical instant occurs when (1) an instance of each higher priority job comes at the same time (time 0), and (2) the job that contributes the maximum blocking time $B(\tau_i)$ has just started executing prior to time 0. Furthermore, to get the worst-case response time, all jobs are assumed to arrive at their maximum rate. The following computation of the q^{th} start time and the q^{th} finish time assumes this worst case scenario.

3.2.4 Computing the q^{th} Start Time:

Before a job τ_i starts execution, there is blocking from lower priority jobs and interference from higher priority jobs. As we proved in Theorem 3.1, the worst case scenario will happen during a busy period starts from an instant when all higher priority jobs arrive at the same time as τ_i and when the longest job with lower priority and higher preemption threshold just started. All higher priority jobs that come before the q^{th} start time $\mathcal{S}_i(q)$ and any earlier instances of job τ_i before instance q should be finished before the q^{th} start time. Therefore, $\mathcal{S}_i(q)$ can be computed iteratively using the following equation.

$$\mathcal{S}_i(q) = B(\tau_i) + (q-1) \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor\right) \cdot C_j \qquad (11)$$

3.2.5 Computing the q^{th} Finish Time:

Once the q^{th} instance of job starts execution, we have to consider the interference to compute its finish time. From the definition of preemption threshold, we know that only jobs with higher priority than the preemption threshold of τ_i can preempt and get the CPU before τ_i finishes. Furthermore, we only need to consider new arrivals of these jobs, i.e., arrivals after $\mathcal{S}_i(q)$. Based on this, we get the following equation for computing $\mathcal{F}_i(q)$:

$$\mathcal{F}_i(q) = \mathcal{S}_i(q) + C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{\mathcal{F}_i(q)}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor\right)\right) \cdot C_j \qquad (12)$$

3.3 Properties of the Model

3.3.1 Properties of the Scheduling Model

With the response time analysis we have done above, we notice that this generalized fixed-priority scheduling model has some interesting properties. Assuming that the priorities of jobs are fixed, these properties help us reason about the effect of changing preemption thresholds of jobs. Furthermore, these properties will help us to understand the relationship between the assignment of scheduling attributes (i.e., priority and preemption threshold) of each job and the scheduling behavior of the model, especially the worst case response time of each job. Therefore, these properties may serve as guidelines while designing solutions for the feasibility problem and optimization problem discussed in Chapter 1.

Lemma 3.4 *Changing the preemption threshold of a job τ_i from γ_1 to γ_2 may only affect the worst-case response time of job τ_i and those jobs whose priorities are between γ_1 and γ_2.*

This can be seen by examining the equations developed for calculating response times. The preemption threshold of a job τ_i determines which (higher priority) jobs may be blocked by τ_i. These jobs are those whose priorities fall in the range $[\pi_i, \gamma_i]$. Therefore, the response time of all these jobs may be affected when τ_i's preemption threshold is modified. It also may affect its own response time since it changes the set of jobs that can preempt it once it has started running. Note that the preemption threshold of τ_i doesn't affect the interference from τ_i on any lower priority job τ_j (which depends on τ_j's threshold, and τ_i's priority). Therefore, changing the preemption threshold doesn't affect any lower priority jobs. A useful corollary directly follows from this lemma, and is presented below.

Corollary 3.1 *The worst-case response time of job τ_i will not be affected by the preemption threshold assignment of any job τ_j with $\pi_j > \pi_i$.*

Corollary 3.1 is useful in developing a strategy for optimal assignment of preemption thresholds. It shows that the schedulability of a job is independent of the preemption threshold setting of any job with higher priority. Therefore, this suggests the threshold assignment should start from the lowest priority job to highest priority job.

Furthermore, Theorem 3.2, presented below, helps us determine the optimal preemption threshold assignment for a specific job. The preemption threshold of a job can range from its own priority to the highest priority in the job set. From the equations for worst-case response time analysis, we can see that a job may reduce its worst-case response time by increasing its preemption threshold, which restricts the set of (higher priority) jobs that can preempt it. However, this is done at the cost of a possible increase in the blocking time of higher priority jobs which may lead to increased worst-case response time of higher priority jobs. Therefore, if there is a set of preemption threshold values that can make a job schedulable, choosing the minimum of them will maximize the chances of finding a feasible preemption threshold assignment.

Theorem 3.2 *Consider a set of n jobs $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and a set of scheduling attributes $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ and $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$, such that the job set is schedulable with Π and Γ (i.e., $sched(\mathcal{T}, \Pi, \Gamma)$ is true). Then, if changing only the preemption threshold of τ_j from γ_j to γ_j' ($\gamma_j' < \gamma_j$), can still make τ_j schedulable, the whole system is also schedulable by setting γ_j' as the preemption threshold of τ_j. That is,*

$$\mathcal{R}_j(\Pi, \Gamma(\gamma_j/\gamma_j')) \leq \mathcal{D}_j \Rightarrow sched(\mathcal{T}, \Pi, \Gamma(\gamma_j/\gamma_j'))$$

Proof: When the preemption threshold of τ_j changes from γ_j to γ'_j ($\gamma_j > \gamma'_j$), the worst-case response time of any job τ_k with $\pi_k < \pi_j$ or $\pi_k > \gamma_j$ will not change. The worst-case response time of a job τ_k such that $\gamma'_j \geq \pi_k > \pi_j$ will also stay the same. Furthermore, any job τ_k with priority $\gamma'_j < \pi_j \leq \gamma_j$ will have no worse worst-case response time with γ'_j than with γ_j. Moreover, we already know that τ_j is schedulable with γ'_j. Therefore, if the whole system is schedulable with γ_j, it is also schedulable with γ'_j. □

A given job set may be unschedulable with any preemption threshold assignment. The following theorem gives a sufficient condition to claim a job set to be unschedulable.

Theorem 3.3 *For any given priority assignment, if there exists a job τ_i, such that setting the preemption thresholds of jobs with lower priorities to the minimum schedulable value and setting its preemption threshold equal to the highest priority in the system can not make the specific job τ_i schedulable, then the job set is unschedulable with this priority assignment.*

Proof: Comparing Equation 11, 12 with Equation 3, we can see that preemption threshold reduces the worst-case response time of a job τ_i by preventing the interference from some higher priority jobs after τ_i starts execution or reduce the blocking. By setting the preemption thresholds of jobs with lower priorities to the minimum value to keep them schedulable and setting the preemption threshold of τ_i to the highest priority in the system gives the maximum reduction to the worst-case response time of τ_i. If τ_i is still not schedulable, since the priority is predefined, there is no way to make it schedulable. Furthermore, the job set will also be unschedulable. □

3.3.2 Properties of A Feasible Implementation Model

In our schedulability analysis presented in the previous section, we assume that each job is assigned its own thread. The worst-case response time is calculated based on the scheduling behavior with this assumption. However, our implementation architecture assumes each thread serves as an event handler for several events, i.e., a thread can hold several jobs. As we mentioned earlier, the queued events in a thread are processed in a run-to-completion manner, which means the jobs in the same thread are scheduled in a non-preemptive manner. If we select two jobs so that one is able to preempt the execution of the other in the nominal thread assignment and put them into one thread to generate a new thread assignment, then the worst-case response time of these two jobs will be changed due to the non-preemptability within a thread.

One motivation behind our implementation architecture is to allow the merging of two (or more) jobs into a single thread, whenever it does not introduce any additional non-preemptability or priority inversions as compared to an implementation model with nominal thread assignment. In this way, we can reduce multi-tasking costs, whenever possible, without sacrificing schedulability.

To make sure our implementation architecture works as designed, we must disallow merging of two jobs into a single thread whenever the scheduling attributes of the jobs allow one job to preempt another. Consider two jobs, τ_i and τ_j. The preemption threshold scheduling model allows τ_i to preempt τ_j in the nominal thread assignment only if $\pi_i > \gamma_j$. We say that two jobs τ_i and τ_j are mutually non-preemptive if τ_i cannot preempt τ_j, and τ_j cannot preempt τ_i in the nominal thread assignment. The following proposition tells us when two jobs are mutually non-preemptive.

Proposition 3.1 *Two jobs, τ_i and τ_j, are mutually non-preemptive if $(\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)$.*

Based on this, we can define a valid thread assignment that precludes two jobs from being mapped to the same thread whenever their scheduling attributes allow one of them to preempt the other. Likewise, we say that an implementation model is valid if the thread assignment in the implementation model is valid. In the rest of this thesis, we restrict our attention to valid implementation models only since it abstracts the dual-level scheduling in practice. It can be easily seen that nominal thread assignment is a valid thread assignment.

Definition 3.7 (Valid Thread Assignment) *A thread assignment Ψ is valid if whenever two jobs are mapped to the same thread, then the two jobs are mutually non-preemptive. That is, in any valid thread assignment the following holds true:*

$$(\forall \tau_i)\ (\forall \tau_j) \quad (\ \psi(i) = \psi(j) \Rightarrow (\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)) \tag{13}$$

Additionally, any grouping of jobs into threads must still allow a job τ_i to preempt a job τ_j, whenever the scheduling attributes allow, i.e., when $\pi_i > \gamma_j$. For this to happen, the thread priorities must be dynamically managed based on the rules below. We use the concept of effective priority defined in Definition 3.5 to explain the priority management of a thread in our implementation architecture.

- When a job τ_i is queued at a thread, then the thread's priority is set to the maximum of its current priority and the priority of the job being queued,
- When a thread removes a job for processing, the thread priority is set to the job's preemption threshold,
- When a thread finishes processing a job, it changes its priority to the highest priority pending job in its queue.

When thread priorities are dynamically managed as above, we can relate a thread priority in an implementation model with a valid thread assignment to the priorities of the job instances active in the thread by the following proposition.

Proposition 3.2 (Thread Priority) *The priority of a thread, in any valid implementation model, is the maximum of the effective priorities of all jobs that have arrived on the thread and not yet finished.*

Proof: Due to the run-to-completion scheduling behavior within each thread, a thread can only have one job that is in 'running' or 'preempted' state. All other jobs in the same thread will be in 'ready' state, with their effective priorities equal to their (nominal) priorities. Suppose τ_i is the preempted or running job in the thread. Its effective priority equals its preemption threshold. Then, following the Equation 13 for valid thread assignments, we can conclude that its effective priority is the highest among all other jobs in the thread. Therefore, since the thread's priority is changed to the job's effective priority when it is picked to execute, the proposition holds true. It is also straightforward to see that if there is no preempted or running job, then the thread priority equals the (effective) priority of the highest priority job that is queued up in the thread. □

Proposition 3.3 *In any valid implementation, the currently running job is always the one with the highest effective priority.*

Proof: By Contradiction. Let τ_i be the currently running job and τ_j have a higher effective priority. Then, from Proposition 3.2, τ_i and τ_j cannot be in the same thread. Also, if they are in different threads, then thread $\psi(j)$ must have a higher priority. Since thread scheduling is preemptive, this could not be true if $\psi(i)$ is running. □

Proposition 3.4 *The scheduling behavior of any valid implementation model* $\mathcal{M} = \langle \Psi, \Pi, \Gamma \rangle$ *for a job set* $\mathcal{T} = \langle \tau_1, \ldots, \tau_n \rangle$ *is identical to the scheduling behavior of the implementation model* $\hat{\mathcal{M}} = \langle \hat{\Psi}, \Pi, \Gamma \rangle$ *with nominal thread assignment.*

Proof: We prove it by showing that if the scheduling behavior is identical up to a point, then the same scheduling decision is taken next. Let S be the scheduler for $\mathcal{M}$ and $\hat{S}$ be the scheduler for $\hat{\mathcal{M}}$. For any identical initial state, the two schedulers will make the same scheduling decision (from proposition 3.3), i.e. select the pending job with the highest priority to execute. We know that job arrivals are independent of the schedulers, thus the jobs arrive at exactly the same time for the two. Since we ignore scheduling overheads, we can assume that the behavior of the two schedulers is identical up to a point. Then the next scheduling point (either because of a new job arrival, or because of the termination of the current job under execution) will come at the same time for the two schedulers, with an identical set of ready, preempted, and running jobs. From proposition 3.3, the same job (the one with the highest effective priority) will be chosen to execute in both S and $\hat{S}$. □

As Proposition 3.4 proved, the scheduling behavior of all valid thread assignments are the same, and thus the worst-case response time of a job in any valid thread assignment is guaranteed to be the same. This gives rise to the following theorem:

Theorem 3.4 *Let* $\mathcal{M} = \langle \Psi, \Pi, \Gamma \rangle$ *be a valid implementation model for a job set* $\mathcal{T} = \langle \tau_1, \ldots, \tau_n \rangle$. *Then, consider* $\hat{\mathcal{M}} = \langle \hat{\Psi}, \Pi, \Gamma \rangle$, *which has the same scheduling attributes, but with the nominal thread assignment. Then, the following predicate is true:*

$$feasible(\mathcal{M}) \iff feasible(\hat{\mathcal{M}})$$

Proof: Follows trivially from proposition 3.4. □

An important conclusion that can be drawn from this theorem is that if a job set is not feasible when each job runs in its own thread, then the job set cannot be made feasible by reducing the number of threads. This implies that the schedulability of a job set can be assessed by simply assuming a nominal thread assignment. Thus, if we are only interested in finding a feasible implementation model, we can restrict our attention to implementation models with nominal thread assignment. In this way, the search space of possible solutions is vastly reduced, making the problem simpler.

Chapter 4

Synthesis of Implementation Models

In the previous chapter, we proposed a new general fixed-priority scheduling model using the notion of preemption threshold. We also showed how the worst-case response times for jobs with pre-defined priorities and preemption threshold can be computed in a simplified model with independent jobs. In this chapter, we address the synthesis problem, i.e., the problem of generating an implementation model. Clearly, we want a synthesized implementation model to be feasible, i.e., the response times of jobs under the synthesized model must be no more than their respective deadlines. We refer to this as the feasibility problem.

Feasibility Problem. *Given a set of jobs $\mathcal{T} = \{\tau_i = \langle C_i, T_i, D_i \rangle \mid 1 \leq i \leq N\}$, find a feasible implementation model, if one exists.*

Additionally, the synthesized implementation model must result in low overheads. This can be achieved by reducing the number of threads in the synthesized implementation. A smaller number of threads reduces inter-thread context switches and also minimizes the per-thread system resources, most notable the memory space associated with the stack of each thread. We refer to this as the optimization problem.

Optimization Problem. *Given a feasible implementation model, optimize it (reduce the number of threads) while maintaining its feasibility such that there exists no other feasible implementation model with fewer threads.*

4.1 Solution Overview

The feasibility problem and the optimization problem are inter-related. While the two problems can be combined into a single optimization problem, we take a two-step approach. The first step is to find a feasible implementation model with nominal thread assignment and the second step is to merge the threads to minimize the number of threads. Theorem 3.4 motivated our approach. It enables solving the feasibility problem regardless of the thread assignment and also allows us to reduce the number of threads in an implementation model while maintaining the feasibility. The reason to take this pragmatic two-step sub-optimal approach is that we know from our simulation experience that the time required for finding an optimal solution makes it inapplicable in the design of real systems. To incorporate our approach in the early stage of the design process of an application, we focus more on cost effectiveness of solutions than on their optimality.

Audsley proposed an efficient optimal algorithm to solve the feasibility problem for the models with independent jobs under preemptive scheduling policy [Aud91a], and its optimality has been proved under non-preemptive scheduling policy [GRS96]. It assumes each job has a unique priority. However, a more general model should allow jobs to have equal priority. Later in this chapter, we demonstrate an example showing that allowing equal priority increases the feasibility of a model under preemptive scheduling while it does not change the feasibility under non-preemptive scheduling and our generalized fixed-priority scheduling model with preemption threshold. Furthermore, we prove that our solution will keep its optimality while equal priority is allowed.

Quantitative assessment is used to show the gains in schedulability of a job set using our new scheduling model compared to traditional preemptive and non-preemptive scheduling model. As run-time overhead is another issue that we are interested in, quantitative assessment is also provided. A non-preemptive scheduler will result in low runtime overheads. Since our scheduling model subsumes non-preemptive schedulers, our scheduling model will incur the same overheads as a non-preemptive scheduler for job sets that are schedulable with non-preemptive scheduling. Therefore, our interest focuses on assessing run-time overheads due to preemptions and comparing them with a preemptive scheduler.

4.2 Generating A Feasible Implementation Model

Theorem 3.4 states that the feasibility of an implementation model with valid thread assignment is independent of the thread assignment. Thus, we can narrow down our search to an implementation model with nominal thread assignment. In this case, the feasibility problem becomes the problem of finding an assignment of scheduling attributes (i.e., priority and preemption threshold) that makes the model schedulable if possible. This problem is an optimization problem with an objective function as follows. When this function reaches its minimum value, 0, a feasible solution is found.

$$f(\mathcal{I}) = \sum_{i=1}^{n} Max(0, \mathcal{R}_i - D_i) \tag{14}$$

where $\mathcal{I}$ represents the implementation model, $\mathcal{R}_i$ stands for the worst-response time of job τ_i, and D_i stands for the deadline of τ_i.

In this section, we will first address the problem of feasible preemption threshold assignment with pre-defined priorities. Then, based on this solution, we develop a branch-and-bound algorithm for priority and preemption threshold assignment. To improve the efficiency of our solution, we also present a greedy algorithm and adopt the simulated annealing approach for this problem.

4.2.1 Feasible Preemption Threshold Assignment with Given Priorities

Based on the results developed in Section 3.3.1, we have developed an algorithm that finds an feasible preemption threshold assignment assuming the priorities are known and fixed. The algorithm is optimal in the sense that the algorithm will alway find a feasible preemption threshold assignment, if one exists.

Figure 4.1 gives the pseudo code of the preemption threshold assignment algorithm. The algorithm assumes that the jobs are numbered $1, 2, \ldots, n$, and that $\pi_i = i$. The algorithm considers the preemption threshold assignment of one job at a time starting from the lowest priority job. For each job considered, it finds the lowest preemption threshold assignment that will make the job schedulable. This is done by computing the worst-case response time of the job using the function **WCRT(job, threshold)**, and comparing it with its deadline. Note that, the response time calculation is possible even with a partial assignment since we consider jobs from low priority to high priority.

```
Algorithm: AssignThresholds
     // Assumes that job priorities are already known
(1)  for ($i$ := 1 to $n$)
(2)      $\gamma_i = \pi_i$ // start from lowest value
         // Find worst-case response time based on the tentative assignment
(3)      $\mathcal{R}_i$ = WCRT($\tau_i$,$\gamma_i$) ;
(4)      while ($\mathcal{R}_i > D_i$) do    // while not schedulable
(5)          $\gamma_i$++ ; // increase preemption threshold
(6)          if $\gamma_i > n$ then
(7)              return FAIL; // system not schedulable.
(8)          endif
(9)          $\mathcal{R}_i$ = WCRT($\tau_i$,$\gamma_i$) ;
(10)     end
(11) end
(12) return SUCCESS
```

Figure 4.1: Algorithm for Preemption Threshold Assignment

It is easy to see the worst-case search space for this algorithm is $O(n^2)$. The optimality of the algorithm is given in the following theorem.

Theorem 4.1 *Given a fixed priority assignment, the algorithm* **AssignThresholds** *will find a feasible preemption threshold assignment, if one exists.*

Proof: Assume a set of n jobs $\mathcal{T} = \{\tau_i \mid 1 \leq i \leq n\}$, with a priority assignment $\Pi = \langle \pi_i, \ldots, \pi_n \rangle$. Without loss of generality, assume that the jobs have been sorted and labeled such that $\pi_i = i$ and $\pi_i > \pi_j$ if $i > j$. Furthermore, let the job set be schedulable with a set of preemption threshold values $\Gamma = \{\gamma_i \mid (1 \leq i \leq n)\}$.

The algorithm assigns preemption thresholds to the jobs starting from τ_1 and going up to τ_n. Let the preemption thresholds found by the algorithm be labeled $\hat{\gamma}_1, \hat{\gamma}_2$, etc. Assume that job τ_i is the first job such that the preemption threshold found by the algorithm is different from the given feasible assignment, that is, $\hat{\gamma}_i \neq \gamma_i$. Then, it must be the case that $\hat{\gamma}_i < \gamma_i$, otherwise, our algorithm will find γ_i rather than $\hat{\gamma}_i$. Based on Theorem 3.2, we know that the job set will still be schedulable if we use $\hat{\gamma}_i$ to replace γ_i in the above feasible preemption threshold assignment.

By repeatedly using the above argument, we can see that the algorithm will also find a feasible preemption threshold assignment $\hat{\Gamma} = \{\hat{\gamma}_i \mid \hat{\gamma}_i \leq \gamma_i, 1 \leq i \leq n\}$. □

4.2.2 Feasible Assignment of Priority and Preemption Threshold

Now we address the general problem of determining an optimal (i.e., one that ensures schedulability) priority and preemption threshold assignment for a given job set. We give a branch-and-bound search algorithm that searches for the optimal assignment. Whether more efficient algorithms can be found for this problem remains an open question at this time. Our algorithm borrows the basic ideas from the optimal priority assignment algorithm presented in [Aud91a, TBW94] for preemptive priority scheduling of a job set. Unfortunately, the introduction of preemption thresholds not only adds another dimension to the search space but also brings more branches into the search, making the search space exponential in size.

Our search algorithm, presented in Figure 4.2, proceeds by performing a heuristically guided search on "good" priority orderings, and then when a priority ordering is complete, it uses the algorithm presented in the previous section to find a feasible threshold assignment. If a feasible threshold assignment is found then we are done. If not, the algorithm backtracks to find another priority ordering.

The algorithm works by dividing the job set into two parts: a sorted part, consisting of the lower priority jobs, and an unsorted part, containing the remaining higher priority jobs. The priorities for the jobs in the sorted list are all assigned. The priorities for the jobs in the unsorted list are unassigned, but are all assumed to be higher than the highest priority in the sorted list. Initially, the sorted part is empty and all jobs are in the unsorted part. The algorithm recursively moves one job from the unsorted list to the sorted list, by choosing a candidate job based on heuristics, as described below. When all jobs are in the sorted list, a complete priority ordering has been generated, and the threshold assignment algorithm is called.

When considering the next candidate to move into the sorted list, all jobs in the unsorted list are examined in turn. To make the search more efficient, we select the "most promising" candidate first, using a heuristic function, described below. If the algorithm fails to find a solution with that partial assignment, it will backtrack and then select the next job. Additionally, we prune infeasible paths by not considering jobs that cannot be made schedulable at the current priority level.

Figure 4.2 gives the pseudo code of the search algorithm, which is presented as a recursive algorithm. It takes two parameters: $\mathcal{T}$, which is the unsorted part (containing all the jobs waiting for priority assignment), and π, which is the next priority to assign. The

```
Algorithm: AssignSchedAttributes($\mathcal{T}$, $\pi$)
     /* Terminating Condition; assign preemption thresholds */
(1)  if ($\mathcal{T}$ == {}) then
     /* Use algorithm in Figure 4.1 for preemption threshold assignment */
(2)     return AssignThresholds()
(3)  endif
     /* Heuristically generate a priority assignment */
(4)  L := {} ;
(5)  foreach $\tau_k \in \mathcal{T}$ do
(6)     $\pi_k := \pi$;   $\gamma_k := n$;   $\mathcal{R}_k$ := WCRT($\tau_k$);
(7)     if $\mathcal{R}_k > D_k$ then Continue ; /* prune */
(8)     $\gamma_k := \pi_k$ ;   $\mathcal{R}_k$ := WCRT($\tau_k$);
     /* Assign Heuristic Value to Each Job */
(9)     if $\mathcal{R}_k \leq D_k$ then
(10)       $H_k$ := GetBlockingLimit($\tau_k$);    /* positive value */
(11)    else
(12)       $H_k := \mathcal{D}_k - \mathcal{R}_k$;   /* negative value */
(13)    endif
(14)    L := L + $\tau_k$;
(15)    $\pi_k := n$ ;   /* reset */
(16) end
     /* Recursively perform depth first search */
(17) while (L != {}) do
(18)    $\tau_k$ := GetNextCandidate(L) ;   /* Select the job with the largest heuristic value next */
(19)    $\pi_k := \pi$;
(20)    if AssignSchedAttributes($\mathcal{T} - \tau_k$, $\pi$+1) == SUCCESS then
(21)       return SUCCESS ;
(22)    endif
(23)    L := L - $\tau_k$;
(24) end
(25) return FAIL
```

Figure 4.2: Algorithm for Feasible Assignment of Priority and Preemption Threshold

jobs that have been assigned priorities are kept separately (and not explicitly shown in the pseudo-code) for preemption threshold assignment at the end of the algorithm. The list of candidates to search is created in L. The computation of worst-case response times assumes that all jobs with unassigned priorities have the highest priority, and that all unassigned preemption thresholds are equal to the job priority.

Pruning Infeasible Paths. First, we tentatively assign a job the current priority and compute its response time with its preemption threshold set to the highest priority in the system. If its computed response time exceeds its deadline, then the job cannot be made schedulable at this priority level (Theorem 3.3). This is because, we assume at this stage that all lower priority jobs have preemption thresholds equal to their priorities. Therefore, we prune such a branch to make the search more efficient.

Heuristic Function. To compute the heuristic function, we compute the response time for the job by tentatively assigning it the current priority and assuming that the preemption threshold equals its priority. Let $\mathcal{R}_k$ be the computed response time for a job τ_k in this manner. Then, the heuristic function is given by:

$$H_k = \begin{cases} BL_k & \textbf{if } \mathcal{R}_k \leq D_k \\ D_k - \mathcal{R}_k & \textbf{else} \end{cases} \tag{15}$$

where BL_k denotes the blocking limit for τ_k. The blocking limit represents the maximum blocking that the job can get while still meeting its response time. Note that at this stage since we have assumed that priorities equal preemption thresholds, there is no blocking. However, once the priorities are fully assigned, it is possible that in the preemption threshold assignment stage a lower priority job may be assigned a threshold that is higher than this job, and can cause blocking. The blocking limit captures the maximum blocking that a job can tolerate while still meeting its deadline. This can be computed by assigning a blocking term to the job, repeating the worst-case response time computation, and checking if it still meets the deadline.

The blocking limit is meaningful if $\mathcal{R}_k \leq D_k$. Otherwise, it is still possible that τ_k may be schedulable at this priority with an appropriate preemption threshold. Thus jobs that need a smaller reduction in interference from higher priority jobs are better candidates for selection. Accordingly, we assign a heuristic value of $D_k - \mathcal{R}_k$ for each job. Note that these values are negative, while BL_k is positive. Thus, such jobs have a lower heuristic value, which is as desired.

4.2.3 A Greedy Algorithm

The efficiency of the optimal algorithm shown above depends heavily on the characteristics of the job set. In the worst case, it has exponential search space in terms of the number of jobs. Clearly, this algorithm becomes infeasible to use, even with a modest number of jobs. Therefore, we have developed a greedy-heuristic algorithm, which we use in our simulations for schedulability comparison. The basic idea of this greedy algorithm is the same as the optimal algorithm. The only difference lies in the branching part. The optimal algorithm will try all possible branches before it finds a solution. However, the greedy algorithm will only try the one that is most promising, or in other words, the one at the head of the candidate list.

This greedy algorithm dominates the preemptive scheduling algorithm, i.e., if a job set is schedulable with preemptive scheduling, then the algorithm will be able to find a feasible assignment as well. This is not surprising since the algorithm extends Audsley's optimal algorithm for priority assignment. On the other hand, there are cases when the algorithm is not able to find a feasible assignment, when a non-preemptive priority assignment algorithm is able to find a feasible assignment. Since a non-preemptive priority assignment is also a feasible solution for our model, the algorithm can be trivially extended to use the non-preemptive priority assignment algorithm first, and then use this algorithm. Without actually doing so, we assume that this is the case, and this extended algorithm is used in our simulations. In this way, our extended algorithm dominates over both preemptive and non-preemptive scheduling algorithms.

4.2.4 Simulated Annealing

Our final approach is the use of simulated annealing to find feasible scheduling attributes. Simulated annealing is a global optimization technique that attempts to find the lowest point in an energy landscape [KGV83]. In developing this algorithm, we again make use of the optimal preemption threshold assignment algorithm. Thus, instead of searching over all possible priority and preemption threshold assignments, we only search over the space of priority assignments. The algorithm is presented in Figure 4.3, and described below.

We use the deadline monotonic priority assignment as an initial starting point for the search. Simulated annealing uses the notion of "energy" of a solution, and the objective is to find a minimum energy solution. For any given priority assignment we calculate the energy of a solution by using a modified form of the optimal preemption threshold assignment algorithm. In this modified algorithm, if no preemption threshold value makes a job feasible, then its preemption threshold is set to the maximum value. The energy of a job τ_i is calculated as $Max(0, \mathcal{R}_i - D_i)$, and the energy of a solution is simply the sum

```
(1)  $P_{old}$ := Deadline monotonic priority ordering
(2)  $C$ := 2 * log(Number of Jobs)) * Maximum Period // *Starting Temperature*
(3)  $E_{old}$ := Energy of $P_{old}$
(4)  **while** (($C_0 > 0.01*$ Minimum Period) )
(5)      **while** (Thermal equilibrium is not reached)
(6)          Generate $P_{new}$, a neighbour of $P_{old}$ by randomly swapping priorities of two jobs.
(7)          $E_{new}$ := Energy of $P_{new}$
(8)          **if** ($E_{new} == 0$) **stop** // *We are done.*
(9)          **else if** $E_{new} < E_{old}$ **then**
(10)             $P_{old} = P_{new}$ ; $E_{old} = E_{new}$ ; // *Always take downward energy jumps*
(11)         **else**
(12)             x := $\frac{(E_{old} - E_{new})}{C_n}$ ; // *Upward energy jump; take it sometimes*
(13)             **if** ($e^x \leq random(0,1)$) **then** $P_{old} = P_{new}$ ; $E_{old} = E_{new}$ ; **endif**
(14)         **endif**
(15)     **end**
(16)     $C = C * 0.96$ ; // *Temperature Cooling*
(17) **end**
```

Figure 4.3: Simulated Annealing Algorithm

of all job energies. Thus, if the energy of a job is 0, then the job is schedulable, and if the energy of a solution is larger than 0, then the solution is infeasible. Larger energy values indicate poorer solutions.

The algorithm moves from one priority assignment to the next using a randomized scheme. First a new neighbour is generated by swapping the priorities of two randomly selected jobs. If the new solution has a lower energy then it is selected as the next candidate. If not, then the neighbour is selected as a candidate probabilistically. The probability of such upward energy jumps reduces with a control parameter (C) – the temperature – which is slowly reduced. At each setting of the control parameter, the solution space is explored until a so-called thermal equilibrium is reached. In our implementation, a thermal equilibrium is reached when either the number of downward jumps exceeds $log(2 * N)$ or when the number of solutions explored exceeds N^2. At any time if we find a solution with zero energy then we stop. Otherwise, the algorithm stops when the temperature is reduced to a point where there are virtually no upward or downward jumps – indicating that no feasible solution could be found.

4.3 Optimality with Equal Priority

Most results shown about optimal priority assignment in the literature assume unique priority for each job. For instance, rate monotonic and deadline monotonic priority assignment, as well as Audsley's optimal priority ordering algorithm, assigns unique priority to each job. With no exception, our optimal algorithm for priority and preemption threshold assignment also assigns unique priority for each job. Nonetheless, a more general model will allow jobs to have equal priority. We are interested in finding out whether these algorithms are still optimal while the model allows equal priority.

4.3.1 Preemptive Scheduler

The traditional worst-case response time analysis does not consider the case for equal priority jobs. Our worst-case response time allows equal priority jobs by assuming that only the instances of the same job will be queued in a FIFO manner, which means before a job instance starts, all job instances with equal priority but from other sources will be scheduled to execute[1]. However, after a job instance starts, equal priority jobs can not preempt its execution. With the preemptive scheduler, the following theorem shows that algorithms that assign unique priorities are no longer optimal. Assigning equal priorities may make a job set that is found to be not schedulable by these algorithms schedulable. Our analysis shows that allowing equal priority may improve the schedulability of a set of jobs with preemptive scheduling while maintains the same schedulability for non-preemptive scheduling and our new scheduling model.

Theorem 4.2 *Allowing equal priority can improve the schedulability of a job set under a preemptive scheduler.*

Proof: It is easy to see that if a job set is schedulable with unique priority for each job then it will be schedulable while allowing equal priority because unique priority is a special case of the latter.

We will use a counter example to prove that a job set that is schedulable while allowing equal priority may not be schedulable if each job has unique priority.

The job set displayed in Table 4.1 is schedulable with a preemptive scheduler allowing equal priority however is not schedulable with unique priority. In the case that each job has

[1]This gives rise to a pessimistic analytical result for worst-case response time.

Job	C_i	T_i	D_i	π_i	WCRT	π_i	WCRT
τ_1	20	70	50	3	20	2	20
τ_2	20	110	100	2	40	1	95
τ_3	35	200	105	1	**115**	1	95

Table 4.1: Improving Schedulability by Allowing Equal Priority: Preemptive Scheduler

a unique priority, the deadline monotonic priority ordering is optimal since the deadline of each job is no more than its period.

From the above example, we can see that allowing equal priority may improve the schedulability of a job set under a preemptive scheduler.

□

4.3.2 Non-preemptive Scheduler

While allowing equal priorities in the preemptive scheduler may increase the schedulability of a job set compared with the case of unique priorities, the same claim can not be made for non-preemptive scheduler. Theorem 4.3 shows that allowing equal priority can not bring any improvement to the schedulability of job sets with non-preemptive schedulers.

Theorem 4.3 *Allowing equal priority will not improve the schedulability of a job set with a non-preemptive scheduler.*

Proof: We prove this theorem by showing that any job set that is schedulable while allowing equal priority under a non-preemptive scheduler will also be schedulable with an assignment with unique priority.

Assume a job set with n jobs $\{\tau_1, \tau_2, \ldots, \tau_n\}$ in which jobs are sorted by non-descending priority. Let $\tau_k, \tau_{k+1}, \ldots \tau_{k+j}$ represent all the jobs with an equal priority m, and they are sorted in non-ascending computation time order, i.e., $C_k \geq C_{k+1} \geq \ldots \geq C_{k+j}$.

We modify the priority and threshold assignment of the job set following the policies below:

(1) For jobs $\tau_1, \ldots, \tau_{k-1}$ keep their priority unchanged.

(2) For jobs $\tau_k, \tau_{k+1}, \ldots, \tau_{k+j}$, assign priority $m, m+1, \ldots, m+j$ to them respectively.

(3) For jobs $\tau_{k+j+1}, \ldots, \tau_n$, add j to their priority.

It is obvious that for jobs $\tau_1, \ldots, \tau_{k-1}$, and for jobs $\tau_{k+j+1}, \ldots, \tau_n$, their worst-case response times remain unchanged. This can be seen by looking at the blocking and interference. In a non-preemptive scenario, the blocking is the computation time of the longest job with lower priority, which is not changed in this case. The interference before the job starts comes from higher priority jobs and is determined by the computation time and arrival pattern of higher priority jobs, which is also unchanged. Furthermore, in a non-preemptive scenario, there is no interference after the job starts. Thus, the worst-case response times of these jobs remain unchanged.

Since we are using a non-preemptive scheduler, for jobs $\tau_k, \tau_{k+1}, \ldots, \tau_{k+j}$, there will be no interference after the job starts execution. This is true for both assignments. However, the blocking term and the interference before the job starts may be different.

In the original priority assignment, $\tau_k, \tau_{k+1}, \ldots, \tau_{k+j}$ are assigned the same priority, m. The blocking terms for them are all the same. For the new priority assignment, there are two cases:

(1) If this blocking term is larger than the largest computation time of these $j+1$ jobs, i.e. larger than C_k, then the blocking term of all these jobs remain unchanged. In this case, it is easy to see that only τ_k has the same interference before it starts, and every one else has less interference. Therefore, the worst-case response time is better or at least no worse than with the original priority assignment.

(2) If the condition in (1) fails, that means C_k is larger than the previous blocking term. The for job τ_k, the analysis in (1) still holds. For jobs in $\tau_{k+1}, \ldots, \tau_{k+j}$, the blocking term is C_k. Compared with the original priority assignment, its interference is reduced by at least C_k and its blocking term is increased by at most C_k. Therefore, its worst-case response time will be no worse than before.

Therefore, with the new priority assignment in which each job is assigned a unique priority, the worst-case response time of each job is no more than the original priority assignment. Thus schedulability is maintained. □

4.3.3 Scheduling with Preemption Threshold

As we can see, allowing equal priority may increase the schedulability of a job set with preemptive scheduling policy, but brings no improvement with non-preemptive scheduler. Our new scheduling model with preemption threshold is more complex than these two and subsumes both as special cases. Fortunately, we found out the optimality of our priority and preemption threshold assignment is not affected by allowing equal priority in our model. Theorem 4.4 shows that the feasibility of a job set will not change if we allow equal priority in the job set. Therefore, there is no need for us to amend our algorithm shown in Section 4.2 to extend to a more general model that allows equal priority.

Theorem 4.4 *Allowing equal priority or not will not affect the schedulability of a job set under our model with preemption threshold.*

Proof: It is easy to see that if a job set is schedulable with unique priority for each job then it will be schedulable while allowing equal priority because unique priority is a special case of the latter.

Assume a job set with n jobs $\{\tau_1, \tau_2, \ldots, \tau_n\}$ in which jobs are sorted by non-descending priority. Let $\tau_k, \tau_{k+1}, \ldots \tau_{k+j}$ represent all the jobs with equal priority m, and they are sorted in non-ascending computation time order, i.e. $C_k \geq C_{k+1} \geq \cdots \geq C_{k+j}$.

We modify the priority and threshold assignment of the job set following the policies below:

(1) For jobs $\tau_1, \ldots, \tau_{k-1}$ keep their priority unchanged. And if the preemption threshold of a job is higher than or equal to m, then add j to its preemption threshold. Otherwise, keep it unchanged.

(2) For jobs $\tau_k, \tau_{k+1}, \ldots, \tau_{k+j}$, add j to their preemption threshold and assign priority $m, m+1, \ldots, m+j$ to them respectively.

(3) For jobs $\tau_{k+j+1}, \ldots, \tau_n$, add j to their priority and preemption threshold.

Now, we create a new priority and preemption threshold assignment that gives these $j+1$ jobs different priorities, and we will prove that the schedulability of the job set is maintained.

For jobs $\tau_1, \ldots, \tau_{k-1}$, and for jobs $\tau_{k+j+1}, \ldots, \tau_n$, the worst-case response time analysis remain unchanged.

For jobs $\tau_k, \tau_{k+1}, \ldots, \tau_{k+j}$, the interference after the job starts execution remains unchanged. However, the blocking term and the interference before the job starts may be different.

With the original priority and preemption threshold, since $\tau_k, \tau_{k+1}, \ldots, \tau_{k+j}$ have the same priority, they have the same blocking term. We divide this situation into two cases:

(1) If this blocking term is larger than the largest computation time of these $j + 1$ jobs, i.e. larger than C_k, then the blocking terms of all these jobs are unchanged. In this case, it is easy to see that only τ_k has the same interference in before it starts, and every one else has less interference, therefore, the worst-case response time is better or at least no worse than with the original priority and preemption threshold.

(2) If the condition in (1) fails, that means C_k is larger than the previous blocking term. Then for job τ_k the analysis in (1) still holds. For jobs $\tau_{k+1}, \ldots, \tau_{k+j}$, the blocking term is C_k. Compared with the original priority and preemption threshold assignment, its interference before the execution starts is reduced by at least C_k and its blocking term is increased by at most C_k. Therefore, its worst-case response time will be no worse than before.

Therefore, if the job set is schedulable with the original priority and preemption threshold assignment, it will also be schedulable with our new assignment. We can keep using this method to break all priority ties until each job has unique priority, and still maintain schedulability.

$\square$

4.4 Optimizing the Implementation Model

In this section we address the optimality problem of finding a feasible implementation model with minimum number of threads for a given job set. The reason for us to set the optimization goal to be minimum number of threads can be seen by observing two facts: (1) a smaller number of threads implies less context switch between threads (2) a smaller number of threads implies less memory requirement to support multi-thread since jobs in the same thread share the stack space. As mentioned earlier, this problem may be viewed as a search over the space of feasible implementation models. Once again, the problem is a non-trivial combinatorial optimization problem. One difficulty in tackling this problem is how to search through the space of feasible implementation models. We again use a decomposition approach to tackle this problem.

From Theorem 3.4, we know that the schedulability of a job set is independent with thread assignment for any valid thread assignments. In other words, with fixed scheduling attributes (priority and preemption threshold), the schedulability of any valid thread assignment is the same as the nominal thread assignment. Based on this, we propose an optimal algorithm to generate a valid thread assignment with minimum number of threads from a nominal thread assignment while keeps the scheduling attributes unchanged.

The optimal thread assignment algorithm can be used in conjunction with the solution of the feasibility problem shown in last section in a straight-forward way. First, find a feasible set of scheduling attributes using the approach given in the previous section. Then, use the optimal thread assignment algorithm to minimize the number of threads. Of course, this does not solve the original problem optimally, and indeed may be much worse than the optimal solution, since the synthesized feasible scheduling attributes were not aimed at reducing the number of threads in the implementation. Therefore, we use a more intelligent approach. We first synthesize a feasible set of scheduling attributes as before. Then, we refine the scheduling attributes so as to eliminate any unnecessary preemptability, while maintaining schedulability. Reducing preemptability may be seen as a heuristic to reduce the number of threads. Finally, we use the refined, but feasible scheduling attributes for optimal thread assignment. Although this is still not an optimal solution, our simulation results shows that this approach gives satisfiable results.

4.4.1 Optimal Thread Assignment

We first begin with the situation when the scheduling attributes are already determined. We then try to group the jobs into the minimum number of threads, such that the thread assignment is valid as per definition 3.7. If the given scheduling attributes are feasible (assuming the nominal thread assignment), then from Theorem 3.4, we know that any implementation model with the same scheduling attributes, and a valid thread assignment is also feasible. Thus, the problem reduces to finding a valid thread assignment with the minimum number of threads, as stated below.

> *Given a set of jobs* $\mathcal{T} = \{\tau_i = \langle C_{i,}, D_i, T_i \rangle \mid 1 \leq i \leq N\}$, *and a feasible assignment of scheduling attributes* Π *and* Γ, *find a valid thread assignment* Ψ *such that the number of threads used is minimized.*

Recall that in a valid thread assignment, if two jobs are mapped to the same thread then they must be mutually non-preemptive. Since this is true for any pair of jobs, it must be true that the set of jobs mapped to the same thread must be pair-wise mutually non-preemptive. To capture the notion of what kind of jobs may be put in a thread, we formally define the concept of a non-preemptive group.

Definition 4.1 (Non-Preemptive Group) *A set of jobs $\mathcal{T} = \{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}\}$ forms a non-preemptive group if for every pair of jobs $\tau_j \in \mathcal{T}$ and $\tau_k \in \mathcal{T}$, τ_j and τ_k are mutually non-preemptive.*

Consider any valid thread assignment, then each thread in this assignment forms a non-preemptive group of jobs. Furthermore, each job is in exactly one such non-preemptive group. Thus, a thread assignment represents a partitioning of jobs into non-preemptive groups. Therefore, the problem can be re-stated as:

> *Given a set of jobs $\mathcal{T} = \{\tau_i = \langle C_i, D_i, T_i\rangle \mid 1 \leq i \leq N\}$, and a feasible assignment of scheduling attributes Π and Γ, find a partitioning of the jobs into non-preemptive groups $G_1, G_2, \ldots, G_M$, such that each job is in exactly one of the groups, and M, the number of groups is minimized.*

In Figure 4.4, we present Algorithm **OPT-Thread** that creates an optimal partitioning of jobs into non-preemptive groups. The algorithm begins by sorting the jobs in a non-decreasing order of their thresholds, with ties broken arbitrarily. Let the sorted list be denoted as L. We then remove the first job (τ_k) from this list and form a new group G. We will call τ_k as the representative of the group. Then, we look at every other job and add any job τ_j into G if $\pi_j \leq \gamma_k$, i.e., it is mutually non-preemptive with τ_k. Also τ_j is removed from L. Note that, since L was already sorted by preemption threshold, it must be the case that $\pi_k \leq \gamma_j$. Once all jobs have been examined, we have formed one non-preemptive group, with the remaining jobs in the list L. We reiterate this process of forming groups until no jobs remain in the list L. We now formally prove that the algorithm is correct (i.e., it produces a valid partitioning) and optimal (i.e., it creates the minimum number of groups).

Proposition 4.1 (Correctness of OPT-Thread Algorithm) *Algorithm* **OPT-Thread** *produces valid partitioning of the job set into non-preemptive groups.*

Proof: Clearly, the algorithm creates a partitioning, i.e., each job is placed into exactly one group. Therefore, we need to show that each group formed by the algorithm is a non-preemptive group. By definition, two jobs τ_i and τ_j are mutually non-preemptive if $\pi_i \leq \gamma_j$ and $\pi_j \leq \gamma_i$. Let us look at the representative member τ_k of a group G. Since the list of jobs is kept sorted by the threshold, and τ_k is the head of the list, it must be the case that

$(\gamma_k \leq \gamma_j)$ for any $\tau_j \in G$. Therefore, we have $\pi_k \leq \gamma_k \leq \gamma_j$. Also, if τ_j is added to G then $\pi_j \leq \gamma_k$. Thus, for any job $\tau_j \in G$, τ_j and τ_k are mutually non-preemptive. Now, consider any two jobs τ_i and τ_j in G. We know that $\pi_i \leq \gamma_k \leq \gamma_i$ and $\pi_j \leq \gamma_k \leq \gamma_j$. It follows that $\pi_i \leq \gamma_j$ and $\pi_j \leq \gamma_i$. □

Theorem 4.5 (Optimality of OPT-Thread algorithm) *Algorithm* **OPT-Thread** *is optimal.*

Proof: We need to show that no other partitioning into non-preemptive groups can be done with a smaller number of groups. Consider any two groups formed by the algorithm, and consider their representative members – say τ_j and τ_k. Then, due to the nature of the algorithm, it must be the case that τ_j and τ_k are not mutually non-preemptive. Therefore, they must be in separate non-preemptive groups in any partitioning of the job set into non-preemptive groups. Since this is true for each pair of representative group members, it is not possible to have a solution with fewer groups. □

Follow the same logic, if the algorithm starts from the highest priority job, selecting jobs with preemption threshold no less than the priority of the representative job to form non-preemptive groups, the algorithm can be proved to be correct and optimal in the same way as shown in Theorem 4.1 and Theorem 4.5. Note that the optimality of the algorithm depends on the ordering. If the non-preemptive groups are generated in other orders, the resulted number of threads may not be the minimum. A counter example can be given as follows. Given four actions A_1, A_2, A_3, and A_4 with scheduling attributes $[\pi_1, \gamma_1]$, $[\pi_2, \gamma_2]$, $[\pi_3, \gamma_3]$, and $[\pi_4, \gamma_4]$ respectively. Assume that we have $\gamma_1 > \gamma_2 > \pi_1 > \gamma_3 > \pi_2 > \gamma_4 > \pi_3 > \pi_4$. Use algorithm in Figure 4.4, it is easy to see that the minimum number of non-preemptive groups is 2. Now, if we first use A_2, A_3 to form a non-preemptive groups, the A_1 and A_3 can not be put into this group so that we will get 3 non-preemptive groups.

4.4.2 Preemption Threshold Assignment to Reduce Preemptions

After partitioning a job set with feasible scheduling attributes into the minimum number of threads, let us look at how to refine feasible scheduling attributes so as to reduce the number of threads created by Algorithm **OPT-Thread**. For this purpose, we use a simple heuristic strategy – we attempt to reduce any unnecessary preemptability that is introduced by the scheduling attributes. Since **OPT-Thread** partitions jobs into non-preemptive groups, less preemptability indirectly affects the number of threads needed.

```
Algorithm: Partitioning Jobs into Minimum Number of Non-Preemptive Groups
(1) ngroups := 0 ;
    /* Sort the jobs by $\gamma_i$, in non-decreasing order */
(2) L := SortJobsbyPreemptionThreshold(JobSet) ;
(3) while (L != NULL) do
    /* Find the job with the smallest value of $\gamma_i$ */
(4)    $\tau_k$ := Head(L); $G[ngroups] := \{\tau_k\}$ L := L - $\tau_k$ ;
(5)    foreach $\tau_j \in$ L do
(6)       if ($\pi_j \leq \gamma_k$) then $G[ngroups] = G[ngroups] + \{\tau_j\}$ ; L := L - $\tau_j$ ; endif
(7)    end
(8)    ngroups := ngroups + 1 ;
(9) end
```

Figure 4.4: An Optimal Algorithm for Job Partitioning with Minimum Number of Non-Preemptive Groups

Note that the set of higher priority jobs that can preempt a job τ_i is determined by the job's preemption threshold γ_i. By increasing the value of γ_i, we can reduce the number of jobs that can preempt τ_i. Suppose we increase the preemption threshold of τ_i from a to b ($a < b$), then this change may result in increased response times for any job τ_k, if $a < \pi_k \leq b$, since any such job may now incur a blocking from τ_i. We can safely increase the preemption threshold if the recomputed worst-case response times of these affected jobs are still no more than their deadlines.

Using the idea given above, we try to increase the preemption threshold of each job to the maximum value that will still keep the job set schedulable. Figure 4.5 gives the algorithm that attempts to assign larger preemption threshold values to jobs. The algorithm considers one job at a time, starting from the highest priority job, and tries to assign it the largest threshold value that will still keep the system schedulable. We do this one step at a time, and check the response time of the affected job to ensure that the system stays schedulable. By going from the highest to lowest priority job, we ensure that any change in the preemption threshold assignment in latter (lower priority) jobs cannot increase the assignment of a former (higher priority) job, and thus we only need to go through the list of jobs once.

Note that the algorithm in Figure 4.5 has a worst-case complexity of $O(n^2)$. However, if we combine it with the minimum preemption threshold assignment algorithm shown in Figure 4.1, then the total complexity is still $O(n^2)$ in the worst-case since this algorithm

```
Algorithm: Assign Maximum Preemption Thresholds
// Assumes that job priorities are fixed, and a set of feasible preemption thresholds are assigned
(1) for (i := n down to 1)
(2)     while (schedulable == TRUE) && ($\gamma_i < n$)
(3)         $\gamma_i$ += 1;    /* try a larger value */
(4)         Let $\tau_j$ be the job such that $\pi_j = \gamma_i$.
            /* Calculate the worst-case response time of job j and compare it with deadline */
(5)         $\mathcal{R}_j$ := WCRT($\tau_j$);
(6)         if ($\mathcal{R}_j > D_j$) then schedulable := FALSE ; $\gamma_i$ -= 1; endif
(7)     end
(8)     schedulable := TRUE
(9) end
```

Figure 4.5: Algorithm for Finding Maximum Preemption Threshold

continues in the same search space and only tries those not tried by the previous one. Therefore, we can say that this optimization is done with no extra cost.

4.5 Performance Evaluation

In previous sections, we made a claim that our new scheduling model can improve the schedulability, reduce scheduling cost, and reduce memory requirements of an implementation model. In this section we provide quantitative assessment to support our earlier claims. We designed and implemented a software tool to test the suitability of the algorithms and simulate the scheduling behavior of our new scheduling model. Instead of showing results from real systems, we choose to evaluate the performance over randomly generated job sets.

4.5.1 Simulation Design

The software tool we developed for the purpose of quantitative assessment can be divided into several parts: worst-case response time analysis, scheduling attributes assignment, system optimization, and run-time simulation. Thus, the benefit that can be obtained by applying our approach can be measured. These benefits include schedulability improvement, overhead reduction, and minimal requirement for total stack space in multi-threaded implementations.

We use randomly generated periodic job sets for our simulations. Each job is characterized by its computation time C_i and its period T_i. We varied two parameters in the generation of the job sets: (1) the number of jobs $nJobs$ and (2) the maximum period $maxPeriod$ for the jobs. For any given pair of $nJobs$ and $maxPeriod$, the job sets were generated as follows: For each job τ_i, we first randomly selected a period in the range $[1, maxPeriod]$ with a uniform probability distribution. Then, we assigned a utilization U_i in the range $[0.05, 0.5]$, again with a uniform probability distribution. The computation time of the job was then assigned as $C_i = T_i * U_i$, and the deadline was set to T_i.

4.5.2 Schedulability Improvement

Breakdown utilization serves as a measure of schedulability [LSD89]. For each randomly generated job set, we measure the breakdown utilization for (1) preemptive scheduling, (2) non-preemptive scheduling, and (3) scheduling with preemption threshold. For preemptive and non-preemptive scheduling, we use Audsley's algorithm for optimal priority assignment. For scheduling with preemption threshold, we provide the results by using simulated annealing[2]. The performance comparison between the greedy algorithm and simulated annealing is provided later.

We scale the computation time of all jobs in a job set to get different utilization, and then test whether the job set is schedulable. Since the randomly generated job sets may have utilization greater than 1 to begin with, we initially scale the utilization to 100%. We then do a binary search to find the maximum utilization at which the job set is schedulable under a particular scheduling algorithm.

We did the simulations for $nJobs \in \{5, 10, 15, 20, 25, 30, 35, 40, 50\}$ and $maxPeriod \in \{10, 20, 50, 100, 500, 1000\}$. In Figures 4.6 and 4.7, we show the schedulability improvement as the number of jobs varies. The results are shown for $maxPeriod =$ 10 and 100; the results are similar for other values. In each case, we plot the average and maximum increase in breakdown utilization when using preemption thresholds.

Figure 4.6 shows schedulability improvement as compared to pure preemptive priority scheduling. As the plot shows, when looking at average improvement, there is a modest improvement in schedulability (2%-8%), depending on the number of jobs. As the number of jobs increases, the improvement tends to decrease. Perhaps, more interesting is the plot

[2]The optimal algorithm is too time consuming to provide results for job sets with more than 20 jobs.

for maximum increase, which shows that the schedulability improvement can be as high as 21% in breakdown utilization for selected job sets, although once again the improvement decreases as the number of jobs is increased.

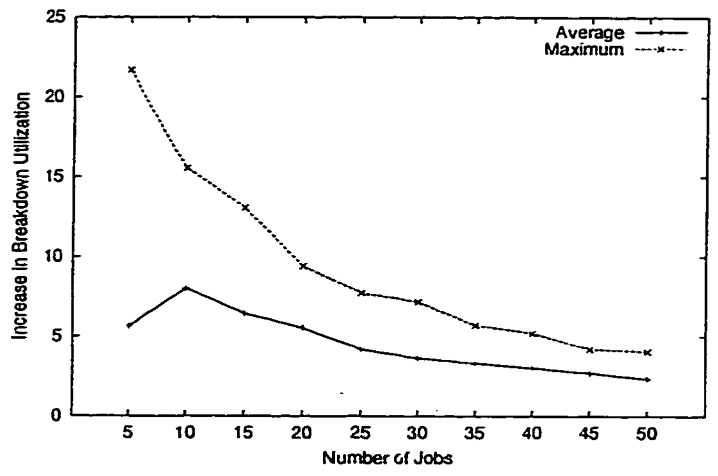
The results showing the schedulability improvement with non-preemptive scheduling are more varied. First, for most ranges of the parameters, the schedulability improvement is much more than the preemptive case (which also means that preemptive scheduling gives higher breakdown utilization, as compared to non-preemptive scheduling). The result should not be surprising since non-preemptive scheduling performs very badly even if one job has a tight deadline, and any other job has a large computation time. In such cases, the breakdown utilization can be arbitrarily low, as can be seen in Figure 4.7(b).

While non-preemptive scheduling performs poorly in general, there are selected cases when it performs better than preemptive scheduling, and better than our simulated annealing algorithm for preemption threshold. Note, however, that we assume that our simulated annealing algorithm is augmented with the schedulability check using non-preemptive scheduler as well, and so in the plots those cases simply show up as zero percent improvement. These results can be seen in Figure 4.7(a), where non-preemptive scheduling outperforms the others when $nJobs \geq 20$; similar results are obtained for other smaller values of $maxPeriod$, but this effect goes away when $maxPeriod = 100$ or more. The reason for this is that with large number of jobs, and a small value of $maxPeriod$, the computation times for all jobs are small. This means that any blocking caused by non-preemption has little effect on schedulability, which gives rise to higher breakdown utilization.

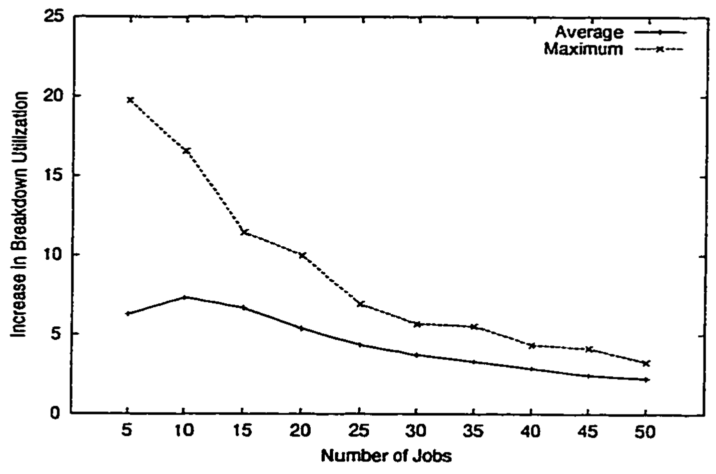
We summarize the observation from the simulation results as follows:

(1) Our approach can never perform worse than either pure preemptive scheduling or pure non-preemptive scheduling; this follows directly from the fact that our model includes both as special cases.

(2) When the number of jobs is relatively small, e.g., $5 - 15$, we observe that in many cases, a significant improvement in schedulability is possible with our approach. For example with $nJobs = 10$, we are able to improve breakdown utilization by as much as 15%. To illustrate the improvement in breakdown utilization, we look at the percentage (we used 100 randomly generated job sets) of job sets for which the schedulability improvement was significant (say more than 5%). In Table 4.2, we present the results for $nJobs = 10$, with $maxPeriod = 10$ and 100 for illustration purposes. We show the number (percentage) of jobs for which greedy algorithm and simulated



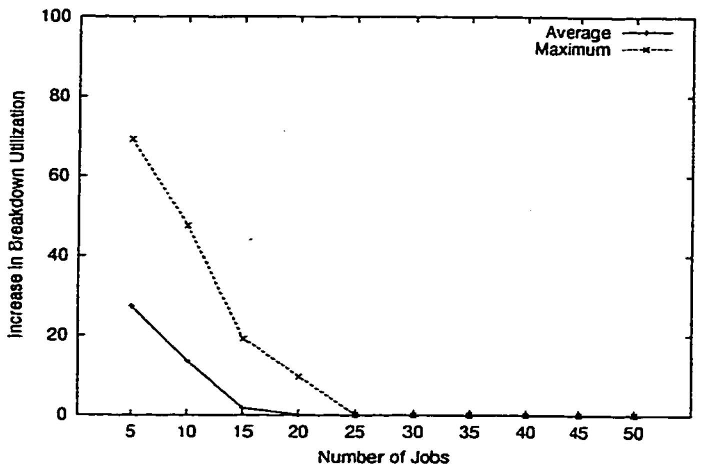
(a) Maximum Period = 10



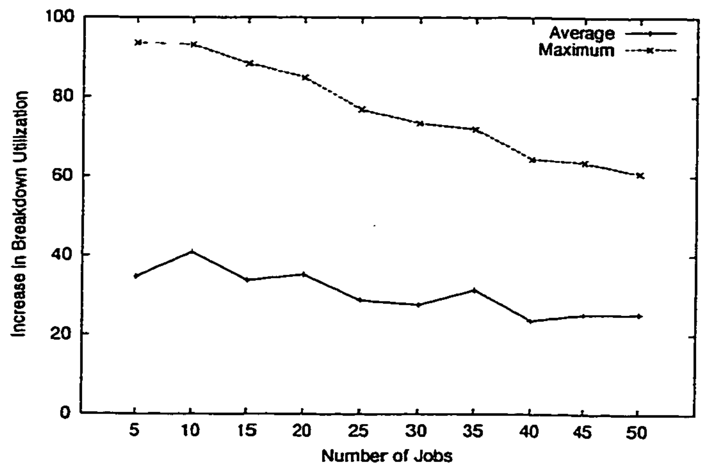
(b) Maximum Period = 100

Figure 4.6: Schedulability Improvement with Preemption Threshold as compared to Preemptive Scheduling.

annealing showed an improvement of more than 5 and 10% of breakdown utilization as compared to the best of preemptive and non-preemptive scheduling. We can see that with our greedy algorithm, we can get a modest schedulability improvement (5 – 10%) in a significant percentage of job sets. With simulated annealing we get modest schedulability improvement in almost half the job sets, and get significant schedulability improvement in a modest percentage of job sets.



(a) Maximum Period = 10



(b) Maximum Period = 100

Figure 4.7: Schedulability Improvement with Preemption Threshold as compared to Non-Preemptive Scheduling.

(3) The schedulability improvement tends to decrease as the number of jobs is increased, such that with about 50 jobs, the schedulability improvement is marginal in most cases (less than 2%). Since our algorithms are not optimal, and with large value of *nJobs*, an optimal solution is not available, it is hard to say whether this lack of improvement is a limitation of our algorithms or whether we have reached the limitation of the scheduling model. In any case, this may be a moot point since in most cases (with larger number of jobs) we observe the breakdown utilization to be

	Greedy Algorithm		Simulated Annealing	
$maxPeriod$	$> 5\%$	$> 10\%$	$> 5\%$	$> 10\%$
10	17	1	45	16
100	28	3	47	14

Table 4.2: Percentage of job sets showing significant schedulability improvements $nJobs = 10$ and $maxPeriod = 10$ and 100.

quite high – 90% or more.

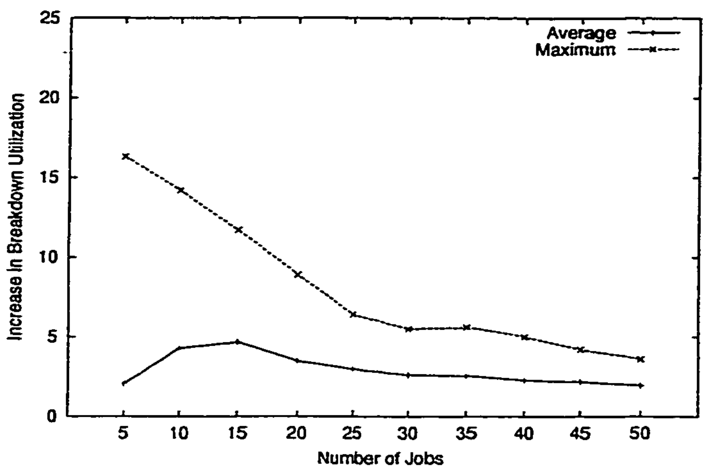
4.5.3 Greedy Algorithm versus Simulated Annealing

We tried both greedy algorithm and simulated annealing for the same job set to find the breakdown utilization. With very rare exceptions, simulated annealing dominates over our greedy algorithm. However, the search space of our greedy algorithm grows linearly to the number of jobs in the job set, while simulated annealing is in the order of $O(nlogn)$. In Figure 4.8, we give the average and maximum improvement of breakdown utilization by using simulated annealing rather than greedy algorithm. Table 4.2 also provides a hint about how often simulated annealing outperforms our greedy algorithm.

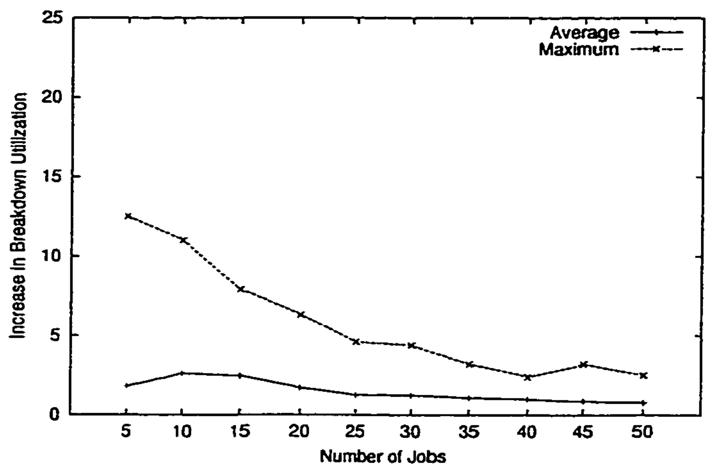
Furthermore, the simulated annealing approach provides a good approximation to the optimal solution. For small job sets, $njobs \in \{5, 10, 15\}$, simulated annealing provides the same breakdown utilization as the optimal algorithm. For larger job sets, a comparison of the solution found by simulated annealing and the optimal solution is missing due to the fact that to get the optimal solution is so time consuming that it may take years before the solution is found.

4.5.4 Overhead Reduction

To evaluate the performance of overhead reduction, we again use randomly generated job sets. We simulate the execution of a job set for 100000 time units, and track the number of preemptions. With $maxPeriod = 1000$, this gives at least 1000 instances of each job in a simulation run. We want to see the savings in preemptions when using preemption thresholds as compared to pure preemptive scheduling. Accordingly, we use percentage reduction in the number of preemptions as the metric, which is defined as:



(a) Maximum Period = 10



(b) Maximum Period = 100

Figure 4.8: Schedulability Improvement with Simulated Annealing as compared to Greedy Algorithm.

$$\frac{NumPreemptions_p - NumPreemptions_{pt}}{NumPreemptions_{pt}} * 100$$

where $NumPreemptions_p$ and $NumPreemptions_{pt}$ are the number of preemptions encountered in a particular simulation run with preemptive scheduling and preemption threshold scheduling respectively.

We did one simulation run for each job set generated in the simulations of previous

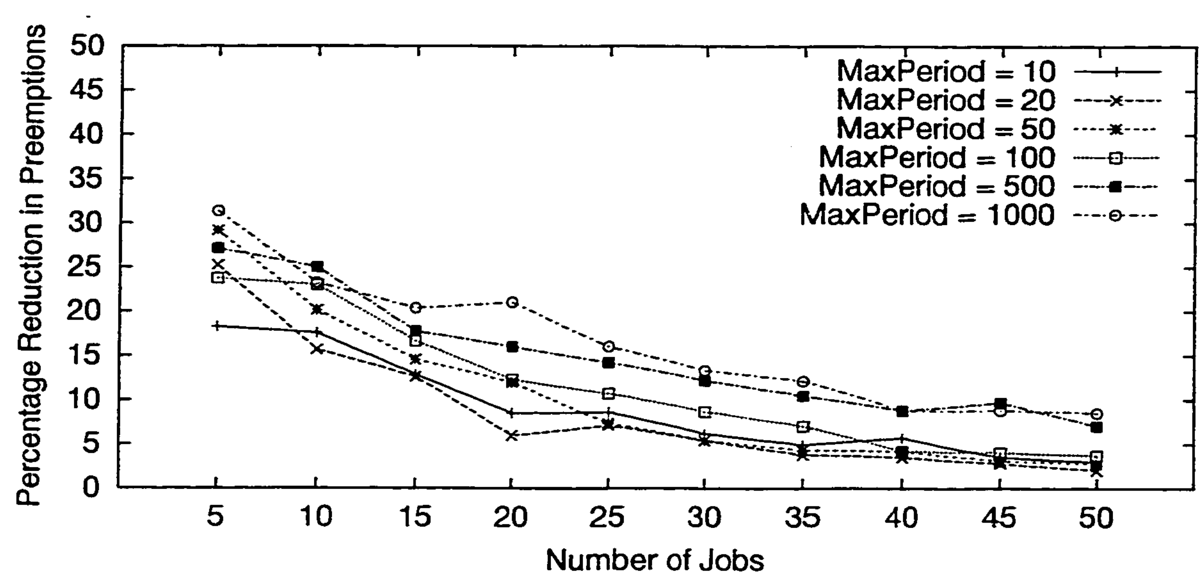


Figure 4.9: Average Percentage Reduction in Number of Preemptions for Preemption Threshold Scheduling as compared to Pure Preemptive Scheduling

sections. The computation times for the jobs were chosen by scaling them to the largest value at which the job set was schedulable under preemptive scheduling (i.e. the breakdown utilization). We assigned priorities to these job sets using the rate-monotonic (optimal) algorithm. We use the same priorities for the preemption threshold case, but additionally assigned preemption thresholds to the jobs as described before. That is, we first generated a feasible assignment of preemption thresholds using the algorithm shown in Figure 4.1. Then, we used the algorithm in Figure 4.5 to optimize the setting of preemption thresholds for preemption reduction. During the simulation, we randomly assigned the initial arrival time of each job τ_i in the range $[0, T_i]$.

We plot the average percentage reduction in the number of preemptions for preemption threshold scheduling as compared to preemptive scheduling. The results are shown in Figure 4.9. As can be seen in the figure, there is a significant reduction in preemptions for small number of jobs, but it tapers down to less than 5% as the number of jobs is increased. Also, for any given number of jobs, the number of reductions is larger for larger values of $maxPeriod$, i.e., when the period range is larger.

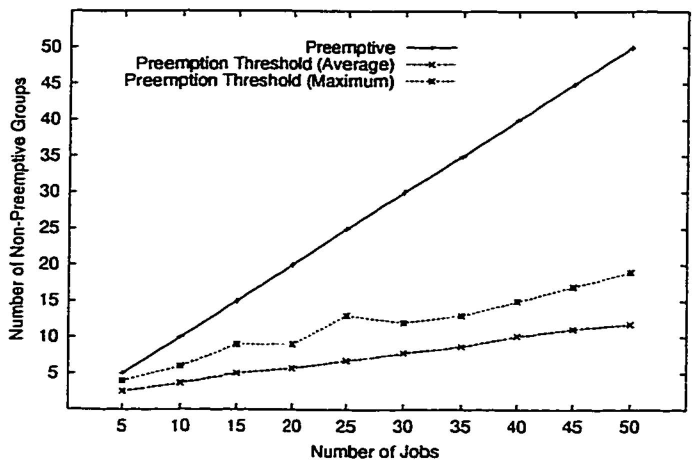


Figure 4.10: Number of Threads as a Function of Number of Jobs, with $maxPeriod = 100$

4.5.5 Reducing the Number of Threads

As we mentioned before, jobs that are non-preemptive to each other may share the same stack space to reduce memory requirements for supporting multi-threaded real-time systems. This enables the jobs in the same thread to share stack space. Reducing the number of threads will reduce the total memory cost of an implementation model. In this section, we quantitatively measure the reduction in number of threads.

Our results show that in many cases, we can reduce the number of threads significantly as compared to the preemptive scheduling case, where each job is put into its own thread. In Figure 4.10, we show the number of threads generated by our approach when the number of jobs is varied from 10 to 100, with $maxPeriod = 100$. To do this, we first found the breakdown utilization with preemptive scheduling. We plot both the average and the maximum number of threads produced by our approach; for comparison, the straight line shows the number of threads with a purely preemptive approach. Similar results are also achieved with other parameters.

As the plots show, the number of threads increase much more slowly than the number of jobs (which would be the case for preemptive scheduling), indicating that as the number of jobs increase, there can be substantial reduction in run-time overheads. For example, with $nJobs = 100$, we have less than 30 threads in all cases, and on an average only 14.3 threads.

Chapter 5

Extension for Event-Driven Complex Real-Time Systems

In this chapter, we consider the extension of our approach to a more general design model that allows shared resources between jobs and inter-job communications (e.g. asynchronously and synchronously triggered jobs). The schedulability analysis and design methodology proposed in this chapter is under the assumption of uni-processor hard real-time system and serves as an important part of the overall auto-synthesis approach presented in Chapter 1. The model is motivated by UML-RT and established by extracting real-time aspects of models described by UML-RT. Therefore, it enables incorporating our auto-synthesis approach with the UML-RT standard for generating event-driven complex real-time systems. As we did with the simplified model, we first study the scheduling theories with the general model, then address the feasibility problem and the optimization problem.

5.1 A General Model Motivated by UML-RT

The simplified model discussed in Chapter 3 and 4 assumes independent jobs, i.e. the execution of one job does not depend on other jobs and there is no communication and resource sharing between jobs. However, this model is too simple to describe complex real-time systems. As the embedded real-time systems encountered in applications such as telecommunications, aerospace, defense, and automatic control tend to be large and extremely complex, there is an urgent need for software designed with sound architecture.

ObjecTime and the Rational Corporation are collaborating to extend the Unified Modeling Language (UML) standard for modeling complex real-time systems. This extension is called UML for Real-Time, or UML-RT for short. It combines the powerful modeling constructs originally developed by ObjecTime for the modeling of complex real-time systems in the Real-Time Object-Oriented Modeling (ROOM) language with UML. We establish our analysis model by extracting the real-time aspects of the design models described by UML-RT and extend our approach to this model.

5.1.1 UML-RT

UML-RT combines UML, role modeling and ROOM concepts to deliver a complete solution for modeling complex, event-driven, and potentially, distributed real-time systems. It uses UML's in-built extensibility mechanism of stereotypes to capture the field-proven ROOM concepts in UML. The modeling construct used for complex real-time systems can be partitioned into two major groups: constructs for modeling structure and constructs for modeling behavior.

Structure Modeling

UML provides two complementary diagrams, namely the class diagram and the collaboration diagram, to capture the logical structure of the system, i.e., the entities in the system and the relationship between them. The basic architectural entity in UML-RT is the *capsule*. Capsules correspond to the ROOM concept of *actors*, which are objects representing independent, concurrently active logical machines. Capsules interact with each other by sending and receiving messages through interface objects called *ports*. *Connectors* represent communication channels interconnecting ports and capture the *communication relationship* between capsules. A capsule may have an internal structure, and collaboration diagrams are used to describe the structural decomposition of capsules [Lyo98, SR98]. Figure 5.1 shows an example of a system structure for a cruise-control system, consisting of several capsules, and inter-connections between capsules through ports and connectors.

Behavior Modeling

The behavior of a capsule is represented by an extended finite state machine using state diagrams. A capsule remains dormant until a message is received by the capsule. Incoming

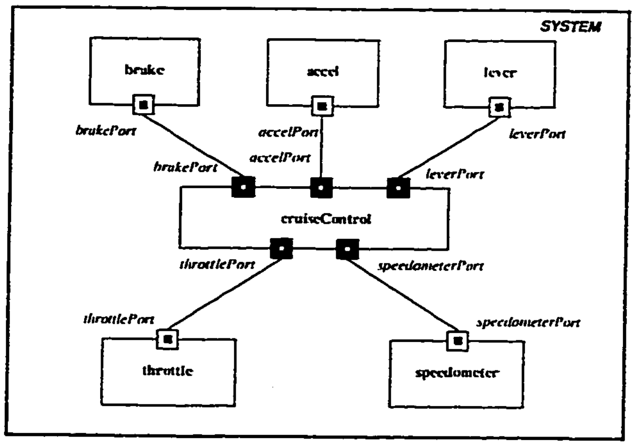


Figure 5.1: Collaboration Diagram for Cruise-Control System

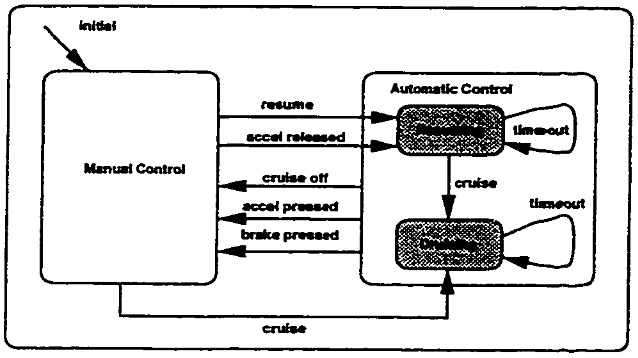
messages trigger transitions associated with the capsule's finite state machine. Actions may be associated with transitions as well as entry and exit points of a state. The sending of messages to other capsules is initiated by an action. Finite state machines can be hierarchically specified, such that a state can be decomposed into a finite state machine. Figure 5.2(a) gives an example of a finite state machine for a cruise-control system. Note the decomposition of the `Automatic Control` state into the `Cruising` and `Resuming` (we have not shown the decomposition of the `Manual Control` state, for clarity.).

Conceptually, each active object has its own thread of control. The finite state machine behavioral model imposes that only one transition at a time can be executed by a capsule. Thus, a *run-to-completion* paradigm applies to state transitions. Figure 5.2(b) depicts the behavioral life-cycle of a capsule using a flow-chart of its conceptual thread of control. A capsule behaves as a message handler, processing incoming requests (sent as messages).

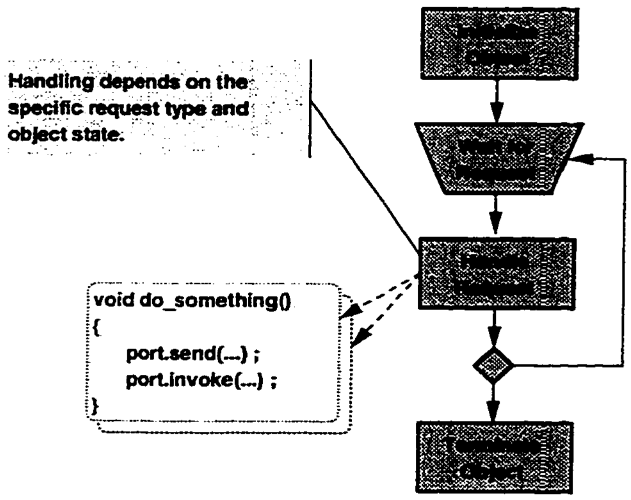
5.1.2 Motivation and Overview of Our General Model

Since UML-RT represents the leading technology in real-time object-oriented modeling, we establish our analysis model by extracting the real-time aspects from the UML-RT models. In this way, we incorporate our approach with the industry standard to automatically translate the design model described using UML-RT into an implementation for a desired target platform.

While finite state machine behavioral models of objects are useful for code-generation,



(a) Finite State Machine


Handling depends on the specific request type and object state.
void do_something()
{
port.send(...) ;
port.invoke(...) ;
}

(b) Capsule Behavior Life Cycle

Figure 5.2: Finite State Machine for Modeling Capsule Behavior

they are not very conducive for reasoning about end-to-end behaviors, or scenarios. UML-RT uses sequence diagrams for this purpose. However, sequence diagrams are weak in expressing a detailed specification of end-to-end behaviors, which is necessary for schedulability analysis. To express our ideas, we extend the sequence diagram notation to capture detailed end-to-end behaviors.

We use the term *transaction* to refer to the entire causal set of actions executed as a result of an external event, i.e., an event coming from an external source. In the rest of the thesis, we use *event* to indicate the arrival of a message at the receiver. Transactions are useful to capture end-to-end system behaviors, and are used in our model for specification

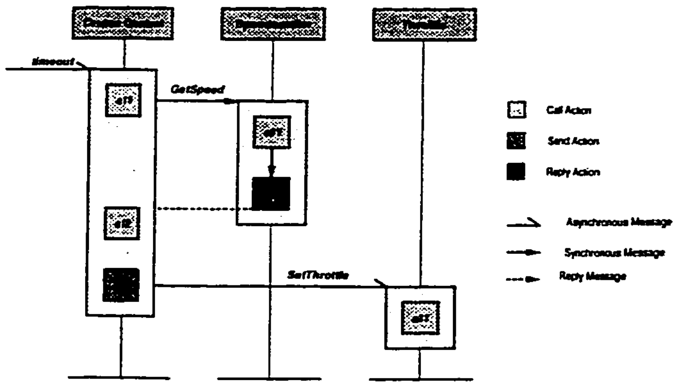


Figure 5.3: Extended Sequence Diagram Representation of a Transaction

of timing constraints and for response time analysis. Since ultimately all processing is initiated by some external event, these transactions can capture all the computations in the design model.

In the rest of the thesis, we will use the term *action* to refer to the entire run-to-completion processing for an event. Each action is, in general, a composite action, and composed from primitive *sub-actions*. These primitive sub-actions include send, call, and return actions, which generate internal events through sending messages to other capsules [RJB99, BRJ99]. Resource sharing is also depicted at the sub-action level using lock and unlock sub-actions.

We depict transactions using extended sequence diagrams, which capture the details of the processing associated with an event. Figure 5.3 depicts the transaction *Feedback Control* for a cruise-control system. The transaction is driven by a timeout message. As can be seen, the cruise control object obtains the speed from the speedometer object using a synchronous call action. It then does the control law calculations and generates a throttle output which is sent asynchronously to the throttle object. The throttle object then sends a command to the actuator.

Currently, these sequence diagrams for transactions must be manually extracted from the design models, although we believe that this process can be automated. One hurdle is that actions are specified in detailed level language (e.g. C++) making it difficult to extract out the necessary information. Note also that there are many "pre-specified" actions that are automatically generated with the code. These actions also include calls to the real-time execution framework. An automatic generation process can easily include these actions as well.

The sequence diagrams are also useful to capture timing constraints [RJB99, BRJ99]. For the purposes of this thesis, we are concerned with (1) arrival rates of external events, and (2) end-to-end deadlines. The end-to-end deadlines can be specified on any action in a transaction; the deadlines are end-to-end in the sense that they are relative to the arrival of the transaction (external event).

To capture the mapping of the design to the implementation, we also model *resources*, which may include hardware resources (CPUs) and software or logical resources. We will restrict our attention to single CPU models. Logical resources include mutex resources, that are used to ensure mutual exclusion. We model both capsules and threads as specialized mutex resources, since only one action at a time may be active within either a capsule or a thread.

5.1.3 Notations

External Events and Transactions

External events are originated from external sources such as input devices (sensors) and will interrupt the CPU from running embedded software when they occur. We also include timed events generated by periodic or one-shot timers as external events.

Since all processing within an event-driven system is ultimately initiated by some external event, we use a collection of transactions to capture all the possible computation in the design model. A *transaction* is defined as a single end-to-end computation triggered by an external event. Each external event stream E_i corresponds to a transaction $\mathcal{T}_i$.

Internal Events and Actions

A transaction is decomposed into a set of actions and internal events that trigger them. A tree structure best represents the internal structure of a transaction, where each node represents an action and each edge connecting actions represents an internal event. Therefore, except the root (the action triggered directly by the external event that triggers the whole transaction), all actions are triggered by an internal event and each action will also generate zero or more internal events that in turn triggers other actions in this particular transaction. Thus internal events (in the form of message passing) represents a precedence between actions in a transaction.

Recall that in our design model, each event is processed within some capsules by the capsule's finite state machine. The processing within a capsule is done in a run-to-completion manner. We use the notation *action* to capture the entire run-to-completion processing within an object for the event. Each action A_i is associated with an event E_i. The internal events are represented by the communication relationships that we will discuss later.

Sub-Actions

While the execution of an action is atomic within the context of the capsule where it is executed, its effects may be visible outside the capsule even when it has only partially executed. This happens when an action communicates with others (e.g. generates internal events), or accesses common resources shared with others as part of its execution. To include these effects in our analysis, we allow an action to be composed of *sub-actions*. Since our interest lies in the communication relationship and resource sharing, we define five kinds of sub-actions: *send*, *call*, *reply*, *lock*, and *unlock*. The first three are defined to describe communication relationships while the last two are defined to describe resource sharing behavior.

A *send* sub-action generates an internal event and asynchronously sends it to the recipient capsule while a *call* sub-action generates an internal event, sends it to the recipient capsule, then blocks waiting for a reply, which is generated by the called action using a *reply* sub-action. To simplify our analysis, we assume that if an action is triggered by a synchronous message, that action must have a single reply sub-action, which should be the last sub-action of that action. A *lock* sub-action requests a specific resource and if no one else is using the resource, it will lock the resource to prevent others to access the same resource before the *unlock* sub-action explicitly releases the resource. Other sub-actions may be defined in the model as long as they have bounded execution times and have no externally visible side-effects.

Sub-actions are also useful to capture conditional behavior within an action, as may happen when the action may execute different steps depending on the state of object or the data associated with the event. Using the notion of sub-action, alternative paths are enabled within an action and within a transaction. However, there is a restriction that simultaneous paths are forbidden in an action. In other words, sub-actions inside an action run in a sequential order where no sub-action in an action can simultaneously branch into more than one sub-action and have them all executed. This restriction is due to the nature of thread as an event handler and the thread that handles this event must execute the sub-actions in sequential order.

Thus an action A_i is decomposed into a sequence of sub-actions $A_i = \langle a_{i,1}, a_{i,2}, \ldots, a_{i,n_i} \rangle$, where each $a_{i,j}$ denotes a primitive action characterized by a worst-case execution time C_{ij}. The computation time of A_i can be presented as the summary of worst-case execution times of its sub-actions on the longest path.

Timing Properties

In our analysis model, we are mainly concerned with two timeliness properties: (a) arrival rates of external events Ψ_i, and (b) end-to-end deadlines D_i. The end-to-end deadlines can be set for transactions as the response time requirements for the external event that triggers the transaction. The deadline can also be specified for any action in a transaction if necessary, however, it is the deadline relative to the arrival of the external event associated with the transaction. To enable our worst-case response time analysis, we assume bounded rate for the arrival of external events. The arrival pattern of internal events depends on the system behavior since they are generated during the process of an external event.

5.1.4 Communication Relationship

We use binary relations to represent the communication relationships between actions, which indicates the precedence of actions in a transaction. There are two kinds of communication relationships, namely asynchronous and synchronous, as defined below:

Definition 5.1 (Asynchronous and Synchronous Relations) *An asynchronous relation $A_i \dashrightarrow A_j$ exists between action A_i and A_j if A_i generates an asynchronous event E_j (using a send sub-action) that triggers the execution of action A_j. Likewise, a synchronous relation $A_i \rightleftharpoons A_j$ exists between action A_i and A_j if A_i generates a synchronous event E_j (using a call sub-action) that triggers the execution of action A_j.*

The synchronous/asynchronous relationship is one-to-many. A single action may synchronously or asynchronously trigger 0 or more actions. While these relations are defined between actions, we present them at the sub-action level to provide better insight for our analysis. To be more specific, a send or call sub-action generates exactly one event that triggers another action and thus establishes the relation between the two actions. For example, $a_{i,p}\ \mathcal{R}\ A_j$, where $\mathcal{R} \in \{\rightarrow, \rightleftharpoons\}$ indicates that sub-action $a_{i,p}$ of action A_i generates E_j that triggers the execution of action A_j.

In addition to the synchronous/asynchronous relationship defined above, it is also useful to define a *causes* relationship, denoted by the symbol $\rightsquigarrow$, which captures the causal relationship between actions. A causal relationship exists between two actions in the same transaction, whenever one of the actions directly or indirectly causes the execution of the other. A formal definition is presented as follows.

Definition 5.2 (Causes Relation)

$$A_i \rightsquigarrow A_j \overset{\text{def}}{=} (A_i \rightarrow A_j)\ \vee\ (A_i \rightleftharpoons A_j)\ \vee\ ((\exists k)((A_i \rightsquigarrow A_k) \wedge (A_k \rightsquigarrow A_j)))$$

In other words, A_i causes A_j if either it directly triggers the execution of A_j (by generating event E_j), or it indirectly causes the execution of A_j through another action A_k. Furthermore, if a causal relation $A_i \rightsquigarrow A_j$ exists, we call A_j a *successor* of A_i and we call A_i an *ancestor* of A_j.

A synchronous relationship between two actions have a significant effect on the scheduling behavior of a real-time system since the call sub-action will block the calling action and wait for a reply from the called action. To simplify the discussion, we define *synchronous set* to identify a continuous sequence of actions linked by synchronous relationships.

Definition 5.3 (Synchronous Set) *The synchronous set of A_i, denoted $\Upsilon(A_i)$, is a set of actions that can be built starting from action A_i and adding all actions that are called synchronously from it. The process is recursively repeated for each action in the set until no more actions can be added to the set.*

In our analysis, we treat synchronously-triggered actions as a natural extension of the calling action. Thus, it is useful to define the concept of *complete synchronous set* to identify a sequence of actions linked by synchronous relationships that starts with an asynchronously triggered action and ends up with actions that do not have synchronous calling sub-action. A complete synchronous set is one thread of control, executed in a run-to-completion manner and scheduled as one action in our model. This implies one priority and preemption threshold for all actions in the same complete synchronous set.

Definition 5.4 (Complete Synchronous Set) *The complete synchronous set of A_i, denoted as $\bar{\Upsilon}(A_i)$, is a set of actions that can be built starting from the synchronous set of A_i by recursively adding actions synchronously calling the set (i.e., all its synchronous ancestors) and all actions that are called synchronously from the actions in the set until no more actions can be added to the set.*

In this way, each complete synchronous set starts with an action triggered by an asynchronous send sub-action or an external event. We call this action as the root of the complete synchronous set, and denote it as $\xi(\bar{\Upsilon}(A_i))$. The root will serve as a unique identifier of a complete synchronous set in our analysis later.

Since the scheduling behavior of a complete synchronous set is the same as if it is an action, for simplicity, it is replaced by an action in our model. Therefore, in our analysis, we only consider asynchronous relationships between actions. The computation time for the action that replaces a complete synchronous set should be the cumulative computation time of all actions in the complete synchronous set.

$$C(\bar{\Upsilon}(A_i)) = \sum_{\forall j, A_j \in \bar{\Upsilon}(A_i)} C(A_j) \tag{16}$$

5.1.5 Resource Sharing Between Actions

Mutual exclusive access to shared resources has been well studied in real-time system design [Mok83, SRL90, Aud91b, RSLR88, CL90, Bak90, KRP+93, BMS90]. Resource sharing protocols are introduced to prevent deadlock and bound the *priority inversion* caused by resource sharing in a priority driven preemptive scheduling system. In our general model, resource sharing may happen between actions of a specific capsule, or actions in a specific thread. In these cases, mutual exclusive access to the shared resource is guaranteed implicitly, due to the non-preemptive scheduling nature within a capsule or a thread. In a more general case, common resource can be shared by actions that neither belongs to the same capsule nor the same thread. Therefore, we need special locking mechanisms to guarantee the mutual exclusion. In our model, we assume that for each resource, there is only one instance to be shared. This requires that any two actions that share the same resource can not access it simultaneously.

We introduce two sub-actions: *lock* and *unlock* to provide mutual exclusive access to commonly shared resource. An action should use lock before it enters the critical section and unlock when it leaves the critical section. The lock sub-action will lock the resource if it is unlocked or enter the action to a waiting queue for this resource if the resource is locked by some other action. The unlock sub-action will unlock the resource and put all actions in the waiting queue for this resource back to the ready queue. Thus, the mutual exclusive accessing of commonly shared resource is guaranteed.

To minimize priority inversion we use the Highest Locker Protocol (HLP) [KRP+93], which is reviewed in Chapter 2. HLP is a refinement of the Priority Ceiling Protocol (PCP). While keeping the merits of PCP, i.e. bounded blocking time equals the longest critical section of lower priority actions, HLP reduces context switches and reduces the complexity at runtime. Using HLP, the lock sub-action raises the priority of the action to the highest priority of the set of actions that share the resource. The unlock sub-action restores the previous running priority of the action. (The previous running priority may be the preemption threshold of the action or ceiling priority of other resources, whichever is higher, depends on the status of the action before the lock sub-action.) The lock and unlock sub-actions implement a mutex. A mutex is created for each resource, and a ceiling priority is assigned to each mutex has the highest priority of all actions that share the resource. HLP has a desirable feature that it can avoid deadlock and chained blocking. However, changing running priority brings complexity in our response time analysis.

5.1.6 Dual Level Scheduling

While extending our approach for event-driven complex real-time systems, we assume a uni-processor multi-threaded architecture with preemptively scheduled threads, where each thread is implemented as an event handler. Traditional schedulability analysis assumes one event per thread, and each thread has a defined priority and timing constraints. Previous attempts of integrating schedulability analysis with object-oriented design models also restrict analysis to a set of threads each handle a single event (see, for example [BW94]). However, as we explained in Chapter 1, a thread usually handles several events in implementation of real systems.

This gives rise to a dual level scheduling: threads are scheduled using priority based preemptive scheduling policy while events queued in a thread are scheduled in a priority based non-preemptive manner. In earlier chapters, we demonstrate that our new scheduling model with preemption threshold can nicely abstract this dual level scheduling behavior.

For our generalized model, we extend these proposed scheduling theories to include communication relationship and resource sharing. In the rest of the thesis, we use action to refer to both the action and the event triggers it.

5.1.7 A Formal Description of the General Model

Based on previous discussions, a formal description for our general model used for schedulability analysis and implementation model generation can be stated as follows:

An event-driven complex real-time system can be described as a model $\mathcal{M} = \{\langle E_i, \mathcal{T}_i \rangle | (E_i \in \mathcal{E}_{ext}) \wedge (\mathcal{T}_i \in \mathcal{T}), 1 \leq i \leq n\}$, *where each external event* $E_i \in \mathcal{E}_{ext}$ *is featured with an arrival function* Ψ_i *and triggers a transaction* $\mathcal{T}_i$ *that is featured with a deadline* D_i. *A transaction* $\mathcal{T}_i$ *is in turn represented as* $\mathcal{T}_i = \{A_j \to A_k | A_j, A_k \in A\}$, *where A is the action set of the model.*

Furthermore, $\forall A_j \in A, A_j = \langle \pi_j, \gamma_j, (a_{j,1}, a_{j,2}, \ldots, a_{j,n_j}) \rangle$, *where each sub-action* $a_{j,l}$ *has a bounded computation time* C_{jl}, *and may be of type call, send, reply, lock, or unlock.*

5.1.8 Restrictions

The schedulability analysis presented in this chapter is conducted on the above model. We impose a few restrictions on it to simplify the analysis. Most of these restrictions are reasonable, and do not impose serious limitations on the application of the model.

1. A synchronously triggered action has a single reply sub-action that is the last sub-action,

2. Any reply action within an asynchronously triggered action is treated as a send action, i.e., it generates an asynchronous event,

3. We treat the synchronously-triggered actions to be natural extensions of the calling action. This implies that a complete synchronous set is in one thread and scheduled as one action, which means the running priority is maintained when calling another action.

4. We assume that the assignment of priorities to actions follows the rule that any successor action A_j with respect to action A_i must have a priority of equal or less importance than action A_i. Thus, we say:

$$(A_i \rightsquigarrow A_j) \Rightarrow (\pi(A_i) \geq \pi(A_j)) \tag{17}$$

5. We assume non-preemptive scheduling within a thread or an active object (capsule). The preemption threshold assignment is limited according to this assumption. Therefore, an implementation model is valid if the following constraint is true:

$$((\psi(A_i) = \psi(A_j)) \vee (\mathcal{O}(A_i) = \mathcal{O}(A_j))) \Rightarrow (\gamma(A_i) \geq \pi(A_j)) \ \wedge \ (\gamma(A_j) \geq \pi(A_i)) \tag{18}$$

5.2 Schedulability Analysis

Similar to the approach we used in the simplified model, we use worst-case response time calculations to determine the schedulability of our general model. However, the response time of an action A_i is derived relative to the arrival of the external event that triggers the transaction that A_i belongs to, instead of the arrival of the event directly triggers A_i. We still use the pessimistic calculation with the assumption that all other actions with higher or equal priority should finish before the current action can start. We made a minor modification to make it more realistic. We eliminate the actions that are triggered by the execution of the current action (they may have equal priority as the current action) in the response time calculation.

We extend the level-i busy period analysis further to calculate the worst-case response time for our general model. Our definition of level-i busy period in Definition 3.6 is still valid in our general model with "job" replaced by "action". Same as in Chapter 3, we need to identify the critical instant that leads to the worst-case scenario.

5.2.1 Problem Statement

In this section, we analyze the schedulability of our general model with given scheduling attributes (including priority, preemption threshold, communication relationships, and ceiling priorities for shared resources) through the computation of action response times. We define the schedulability of a model as following:

A model is said to be schedulable if all response times obtained for each transaction do not surpass the respective deadlines.

Each transaction has an explicitly assigned deadline. Some work has proposed approaches for decomposing these end-to-end deadlines into deadlines for each action [GHS94, BB99]. However, assigning deadlines is beyond the scope of this thesis and deadline assignment is irrelevant to our schedulability analysis. In this thesis, we use a simple approach by assuming all actions in the same transaction share the same deadline, which is the deadline of the transaction. Since we are using HLP (Highest Locker Protocol) to avoid unbounded priority inversion due to resource sharing, the ceiling priority is determined by the priority setting of actions and is no longer treated as an independent set of variables. The effect of communication relationships is partly reflected by the restriction on priority setting of actions, and will be analyzed later. Thus the problem of schedulability analysis can be stated as follows:

Given a model $\mathcal{M} = \{\langle E_i, \mathcal{T}_i \rangle | (E_i \in \mathcal{E}_{ext}) \wedge (\mathcal{T}_i \in \mathcal{T}), 1 \leq i \leq n\}$, where each external event E_i is featured with an arrival function Ψ_i and triggers a transaction $\mathcal{T}_i$ that is featured with a deadline D_i. A transaction $\mathcal{T}_i$ is in turn represented as $\mathcal{T}_i = \{A_j \rightarrow A_k | (A_j, A_k \in A)\}$. For each action in A, $A_j = \langle \pi(A_j), \gamma(A_j), C(A_j) \rangle$, find whether the model $\mathcal{M}$ is schedulable with the given configuration.

As we did for the simplified model before, we assume nominal thread assignment, where each action has its own thread, while calculating the worst-case response time analysis. Later in this chapter, we prove that for all valid thread assignments, the worst-case response time is identical.

5.2.2 Scheduling Behavior Analysis

Our generalized model abstract event-driven complex real-time system, in which multiple threads serve as event handler and scheduled by their priority. Since each thread handles several events that has different priority, the thread priority is not fixed. In this model, action (events) has fixed priority and preemption threshold while each critical section has a ceiling priority.

Before an action starts execution (i.e. in the 'ready' state in which event is queue in the event queue of the thread), it is scheduled by its nominal priority. When it starts execution, the running priority is raised to its preemption threshold before it finishes. These are the same as in the simplified model. However, the resource sharing brings more complication. When the execution of an action is entering a critical section, if the ceiling priority of the resource is higher than the preemption threshold, the running priority is raised to the ceiling priority and maintained before leaving the critical section. We extend the definition of *effective priority* to include this new scenario.

Definition 5.5 (Effective Priority) *The effective priority of a job equals its (nominal) priority if it is in the state 'ready,' equals its preemption threshold if it is not accessing any shared resources and in the 'preempted' or 'running' state, and equals the higher one of its preemption threshold or the ceiling priority of the shared resources that it is accessing if it is in the 'preempted' or 'running' state.*

With the definition of the effective priority, dynamical management of thread priorities can be described in following rules:

- When an action is queued at a thread, then the thread's priority is set to the maximum of its current priority and the priority of the action being queued,
- When a thread removes an action for processing, the thread priority is set to the action's preemption threshold,
- When the processing of an action enters a critical section to access a shared resource, the thread's priority is set to the maximum of its current priority and the ceiling priority of the resource.
- When a thread finishes processing an action, it changes its priority to the highest priority pending action in its queue.

As with our simplified model, we define a *nominal thread assignment* with which each action assigned with its own thread and a *valid thread assignment* as a thread assignment that any two actions mapped into the same thread are mutually non-preemptive. For simplicity, our worst-case response time equations in this chapter are based on nominal thread assignment while we prove that they are applicable to any valid thread assignment.

The communication relationship and resource sharing makes it complicated to analyze the scheduling behavior of our model. However, using the concept of preemption threshold and ceiling priority for shared resources, we are able to describe the complicate scheduling behavior through the change of running priority. For simplicity in our discussion, we divide the lifetime of an action into two parts: from the arrival to the start of execution and from the start to the end of the execution.

During the period between the arrival and the start of execution, an action is in the 'ready' state and there are two factors affecting the length of this period:

1. Interference

   Just as with a normal priority based scheduler, all actions that have higher or equal priority than the current action will be scheduled before the current action can start. The general model did not bring much change to this case. The only restriction we have is that the ancestors of the current action should be finished before the current action can start and none of its successors will be able to start before the current action finishes. Since we made the assumption that an action will have priority less than or equal to its ancestor, we only need to concern ourselves with successors.

2. Blocking introduced by high effective priority

   In our model, there are two cases where we may have the running priority of an action higher than its nominal priority. First, an action will start with running on its preemption threshold. Second, an action may run on the ceiling priority of a resource while it is in the critical section and the ceiling priority is higher than its preemption threshold. In either case, the effective priority, which is higher than its nominal priority, may block the start of an action that has a higher nominal priority than the current running action but no higher than the effective priority of the current action. In Chapter 3, we have discussed the blocking caused by preemption threshold. The introduction of resource sharing using HLP (Highest Locker Protocol) did not change the nature of the analysis. Nonetheless, it brings new considerations for the maximum blocking calculation.

   While having two sources of blocking: preemption threshold and ceiling priority for shared resources, it is necessary to study the relationship between these two. We redefine the *blocking range* for the general model as follows:

Definition 5.6 (Blocking Range) *The blocking range of an action A_i is defined as the range of priorities given by its nominal priority and its current running priority.*

Proposition 5.1 *An action A_i can be blocked by at most one lower priority action A_j. Furthermore, it must be the case that its running priority is higher than or equal to $\pi(A_i)$.*

Proof: The proof is trivial following the same logic shown in the proof for Lemma 3.2. □

After the start and till the end of execution, an action is in either 'running' or 'preempted' state and there are another two factors affecting the length of this period:

1. Interference from actions with higher priority

   The interference only comes from actions with higher priority than the effective priority of the current action. The effective priority is the maximum of the preemption threshold or the ceiling priority of the resource if in the critical section. An action with a priority higher than the preemption threshold of current action but lower than the ceiling priority of a specific critical section may be blocked if it arrives while the action is in this critical section. However, if this critical section is not at the end of the action, the blocked higher priority action will still finish before the current action finishes execution.

2. Effect of resource sharing with higher or lower priority actions

   Since we are using the HLP, Theorem 5.1 proves that the current action will not be blocked while trying to enter a critical section. Theorem 5.2 further proves that sharing resources will not affect the response time of the actions interfering with the current action unless they are sharing the same resource. We will discuss in detail the case when higher priority actions share resources with the current actions later in this chapter.

Theorem 5.1 *There will be no blocking caused by sharing resources with lower priority actions during the execution of an action once it gets started.*

Proof: Assume actions A_i and A_j have priorities and preemption thresholds $\langle\pi(A_i), \gamma(A_i)\rangle$ and $\langle\pi(A_j), \gamma(A_j)\rangle$ respectively. We have $\pi(A_i) > \pi(A_j)$. A_i and A_j share the same resource R, which has a ceiling priority $\pi_{\mathcal{R}}$. $\pi_{\mathcal{R}} \geq \pi(A_i) > \pi(A_j)$.
We consider the following possible situation:

(1) A_j is in the critical section when A_i arrives.

A_j has a running priority of $\pi_{\mathcal{R}} \geq \pi(A_i)$. Therefore, A_i can not start before A_j leaves its critical section. This will cause blocking for A_i before it starts.

(2) A_j has started but has not entered the critical section when A_i arrives.

A_j is running on priority $\gamma(A_j)$. If $\gamma(A_j) < \pi(A_i)$, then A_j will not be able to continue executing before A_i finishes. Of course, this will not block A_i from accessing its critical section. If $\gamma(A_j) \geq \pi(A_i)$, A_j will finish before A_i can start. As in (a) this will cause blocking before A_i starts.

(3) A_j has not started when A_i arrives.

This includes 2 situations: A_j arrives before A_i but has not started when A_i arrives, and A_j arrives after A_i. In either situation, since $\pi(A_i) > \pi(A_j)$, A_j will not get started before A_i finishes.

In summary, A_i will not be blocked by A_j due to resource sharing. □

Following the same reasoning, we can easily come up with the following theorem:

Theorem 5.2 *Any action that interferes between the start and the end of execution of the current action will not suffer blocking due to resource sharing from actions with lower priority than the current action. However it may be blocked by the current action if they share the same resource.*

5.2.3 Blocking Analysis

As discussed above, blocking only happens before the current action starts and there are two sources of blocking: preemption threshold and resource sharing. Actually, these two sources of blocking have the same nature: a lower priority and a higher running priority than the nominal priority of current action. This leads to an overlap between these two kinds of blocking.

Assume two actions A_i and A_j, where $\pi(A_i) > \pi(A_j)$. A_j has a preemption threshold $\gamma(A_j)$, and shares resource $\mathcal{R}$ with some actions (may includes A_i). The resource $\mathcal{R}$ has a ceiling priority $\pi(\mathcal{R})$ and the length of critical section for $\mathcal{R}$ in A_j is $C(\mathcal{R})$. Let us look at the following situations:

1. $(\gamma(A_j) < \pi(A_i)) \wedge (\pi(\mathcal{R}) < \pi(A_i))$

   In this situation, A_j will not block A_i.

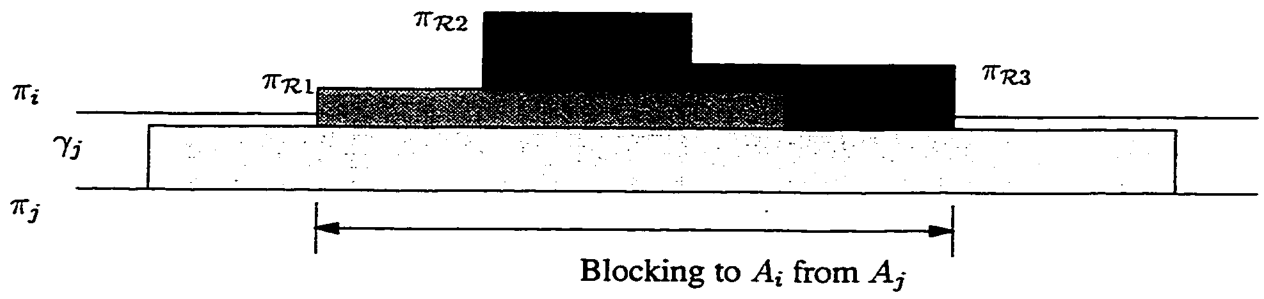


Figure 5.4: Blocking caused by overlapped critical section

2. $(\gamma(A_j) \geq \pi(A_i)) \wedge (\pi(\mathcal{R}) < \pi(A_i))$

   In this case, the preemption threshold of A_j may cause blocking and the maximum blocking time from A_j is $C(A_j)$.

3. $(\gamma(A_j) < \pi(A_i)) \wedge (\pi(\mathcal{R}) \geq \pi(A_i))$

   In this case, the ceiling priority of the critical section of A_j may cause blocking and the maximum blocking time from A_j is $C(\mathcal{R})$. Note that A_j may share more than one resource with other actions. If the critical sections do not overlap, then the maximum blocking time caused by A_j will be the longest critical section within A_j that satisfied the above condition. Otherwise, the maximum blocking time will be the union of a sequence of overlapped critical sections of A_j where their ceiling priority are all higher than $\pi(A_i)$. Figure 5.4 depicts the situation of overlapping critical section.

4. $(\gamma(A_j) \geq \pi(A_i)) \wedge (\pi(\mathcal{R}) \geq \pi(A_i))$

   In this case, it is hard to say which one causes the blocking. However, the maximum blocking time from A_j is $C(A_j)$ again. Therefore, we can merge it with the second situation.

According to Proposition 5.1, the maximum blocking of an action A_i comes from either of two sources: (a) action A_j, $\pi(A_j) < \pi(A_i)$ and $\gamma(A_j) \geq \pi(A_i)$, or (b) action A_k, $\pi(A_k) < \pi(A_i)$ and $\gamma(A_k) < \pi(A_i)$, and it shares resource $\mathcal{R}$ and $\pi(\mathcal{R}) \geq \pi(A_i)$. For (a), it is simply the longest computation time of actions. For (b), the situation is a little complicated. We know that a resource can be shared by several actions and one action may access several shared resources during its processing. However, for a specific resource, there will be only one action in critical section at a time. Therefore, the maximum blocking from (b) will be the longest critical section if critical sections do not overlap, and the largest union of overlapping critical section sets otherwise, as shown in Figure 5.4.

The maximum blocking time that action A_i may suffer is given by Equation 19. To simplify the expression, we assume critical sections do not overlap. (The equation can be easily extended to include overlapping.)

$$\begin{aligned} B(A_i) &= MAX(\max_j\{C(A_j) :: \gamma(A_j) \geq \pi(A_i) > \pi(A_j)\}, \\ &\quad \max_{\mathcal{R}_k}\{C(A_m, \mathcal{R}_k) :: (\gamma(A_m) < \pi(A_i)) \wedge (\pi(\mathcal{R}_k) \geq \pi(A_i))\}) \end{aligned} \tag{19}$$

where $C(A_m, \mathcal{R}_k)$ denotes the computation time for the critical section in which action A_m accesses $\mathcal{R}_k$.

5.2.4 Identifying Critical Instant

A critical instant is an instant that leads to the worst-case scenario of the response time of an action. In our extended level-i busy period analysis, the worst-case response time is found in a level-i busy period starting from a critical instant.

Based on our blocking analysis above, we propose the following lemma:

Lemma 5.1 *In a level-i busy period, only the first instance of action A_j whose priority $\pi_j = i$ will suffer blocking from actions with lower priority. Any instance after the first instance will not suffer from blocking. Furthermore, at most only one instance of an action A_k with $\pi_k < \pi_j$ and $\gamma_k \geq \pi_j$ or an action has a critical section with a ceiling priority higher or equal to i may contribute to blocking.*

Proof: It is easy to see that the start of processing first instance of action A_j ensures the finish of all active instances of actions with lower priority and a higher effective priority. Since in a level-i busy period, there is always at least one pending instance of action A_j, no action with lower priority can start. Therefore, after the start of processing first instance of action A_j, the following instance of A_j will not suffer from blocking.

Furthermore, based on our blocking analysis, we can easily see that at most only one instance of a job τ_k with $\pi_k < \pi_j$ and $\gamma_k \geq \pi_j$ or an action has a critical section with a ceiling priority higher or equal to i may contribute to blocking. □

We give the definition of the critical instant of an action as follows:

Definition 5.7 *The critical instant of an action A_i occurs when (1) an instance of all external events with higher or equal priority comes at the same time, or in other words, all transactions whose start action has higher or equal priority comes at the same time. (2) the action that contributes the maximum blocking time has just entered its section that cause the blocking.*

With this definition, we can prove the following theorem which serves as the foundation of our worst-case response time calculation. Note that the arrival time we used to measure the response time for an action is the arrival time of the external event that triggers the transaction holding the action.

Theorem 5.3 *The worst-case response time for an action A_j ($\pi_{j,=}i$ occurs during a level-i busy period initiated by a critical instant.*

Proof: The proof is trivial following the same logic used to prove Theorem 3.1. □

Let $Arr_{A_i}(q)$ denote the arrival time of instance 'q' of the external event that triggers the transaction containing A_i. Let $\mathcal{S}_{A_i}(q)$ and $\mathcal{F}_{A_i}(q)$ denote the q^{th} start time and q^{th} finish time of action A_i respectively. Assume external events arrive at their maximum rate. We iteratively compute the results of $\mathcal{S}_{A_i}(q)$ and $\mathcal{F}_{A_i}(q)$ for $q = 1, 2, 3, \ldots$ until $q = m$ such that $\mathcal{F}_{A_i}(m) \leq Arr_{A_i}(m+1)$. Then the worst-case response time of action A_i is given by:

$$\mathcal{R}_{A_i} = \max_{q \in [1,\ldots,m]} \mathcal{F}_{A_i}(q) - Arr_{A_i}(q) \qquad (20)$$

5.2.5 Computing q^{th} Start Time

As we discussed above, to calculate the q^{th} start time of A_i, we need to consider both blocking and interference. The maximum blocking is given in Equation 19. We need to calculate the interference. Assume that A_i is in transaction $\mathcal{T}(A_i)$. Interferences may come from higher or equal priority actions from other transactions as well as inside transaction $\mathcal{T}(A_i)$. However, for those actions that are successors of A_i (i.e. $\forall A_j \in \mathcal{T}(A_i), A_i \rightsquigarrow A_j$), the q^{th} arrival will not come before the q^{th} A_i finishes. Therefore, for successors of A_i that have equal priority as A_i, only their instances from 1 to q-1 will be considered for interference.

As we know, in our model, each external event that triggers a unique transaction has a bounded rate. Therefore, we use their maximum arriving rate for the worst-case response time analysis. This simplifies the arrival function for the external event. However, the arrival patterns of the internal events are quite complex. Fortunately, we do not need the details of these arrival patterns to compute the worst-case response time of a specific action. We can derive the number of arrived instances without knowing the exact pattern. This claim is based on the following reasoning. As mentioned earlier, our model has the priority restriction shown in Equation 17, which imposes that an action should have priority equal to or less than its ancestor. This implies that if an external event arrives before the start of the q^{th} instance of the current action, then all higher or equal priority actions in the same transaction triggered by that external event will arrive before the start of the q^{th} instance of the current action. It is easy to see that all the ancestors of the current action will have the same number of arrivals as the external event that triggers the whole transaction while all its successors with equal priority as the current action will have one arrival less than the current action. Therefore, it is safe to use the number of external event arrivals as the number of arrivals for the ancestors of the current action and those higher or equal priority actions in other transactions.

Equation 21 below shows how the q^{th} start time of action A_i can be iteratively computed.

$$\begin{aligned}
\mathcal{S}_{A_i}(q) \quad = \quad & B(A_i) \\
& + \sum_{(\mathcal{T}(A_j) \neq \mathcal{T}(A_i)) \wedge (\pi(A_j) \geq \pi(A_i))} \left(\left(1 + \left\lfloor \frac{\mathcal{S}_{A_i}(q)}{\mathcal{T}(A_j)} \right\rfloor \right) C(A_j) \right) \\
& + \sum_{(\mathcal{T}(A_j) = \mathcal{T}(A_i)) \wedge (\neg(A_i \rightsquigarrow A_j)) \wedge (\pi(A_j) \geq \pi(A_i))} \left(\left(1 + \left\lfloor \frac{\mathcal{S}_{A_i}(q)}{\mathcal{T}(A_i)} \right\rfloor \right) C(A_j) \right) \\
& + (q-1) \sum_{(\mathcal{T}(A_j) = \mathcal{T}(A_i)) \wedge (A_i \rightsquigarrow A_j) \wedge (\pi(A_j) \geq \pi(A_i))} C(A_j)
\end{aligned} \qquad (21)$$

5.2.6 Computing q^{th} Finish Time

Theorem 5.1 and 5.2 suggest that between the start and the end of the execution of the current action, there is no blocking caused by resource sharing from lower priority actions. Nonetheless, HLP makes the calculation for interference more complicated.

Action A_i may access resources shared with some higher priority actions. Therefore, there may exist a resource $\mathcal{R}$ whose ceiling priority $\pi(\mathcal{R}) > \pi(A_i)$. If $\pi(\mathcal{R}) \leq \gamma(A_i)$ this ceiling priority will not affect the scheduling behavior of A_i between the start and the end of its execution. If $\pi(\mathcal{R}) > \gamma(A_i)$, it requires the current action to increase its running priority to $\pi(\mathcal{R})$ at the entry of the critical section thus preventing interference from action A_j, $\pi(\mathcal{R}) \geq \pi(A_j) > \gamma(A_j)$. However, if this critical section is not at the end of action A_i, the running priority will recover to $\gamma(A_i)$ at the exit of the critical section, allowing A_j to preempt and finish before A_i finishes. In this case, the q^{th} finish time $\mathcal{F}_{A_i}(q)$ can be iteratively computed using the equation given below:

$$\begin{aligned}\mathcal{F}_{A_i}(q) &= \mathcal{S}_{A_i}(q) + C(A_i) \\ &+ \sum_{\pi(A_j)>\gamma(A_i)} \left(\left\lceil \frac{\mathcal{F}_{A_i}(q)}{T(A_j)} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_{A_i}(q)}{T(A_j)} \right\rfloor \right) \right) \cdot C(A_j)\end{aligned} \tag{22}$$

However, when the critical section with a ceiling priority $\pi(\mathcal{R}) > \gamma(A_i)$ is at the end of A_i, we have to divide this period into two parts to compute the q^{th} finish time. The first part is the period between the start of A_i and the entry to the last critical section. The second part is the period between the entry to the last critical section and the finish of the whole transaction. Assuming the computation time of the critical section to be $C(cs)$, we use $CS_{A_i}(q)$ to denote the entry time of the critical section, which can be iteratively computed using the equation given below:

$$\begin{aligned}CS_{A_i}(q) &= \mathcal{S}_{A_i}(q) + C(A_i) - C(cs) \\ &+ \sum_{\pi(A_j)>\gamma(A_i)} \left(\left\lceil \frac{CS_{A_i}(q)}{T(A_j)} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_{A_i}(q)}{T(A_j)} \right\rfloor \right) \right) \cdot C(A_j)\end{aligned} \tag{23}$$

In this case, the q^{th} finish time can be iteratively computed using the equation given below:

$$\begin{aligned}\mathcal{F}_{A_i}(q) &= CS_{A_i}(q) + C(cs) \\ &+ \sum_{\pi(A_j)>\pi(\mathcal{R})} \left(\left\lceil \frac{\mathcal{F}_{A_i}(q)}{T(A_j)} \right\rceil - \left(1 + \left\lfloor \frac{CS_{A_i}(q)}{T(A_j)} \right\rfloor \right) \right) \cdot C(A_j)\end{aligned} \tag{24}$$

5.3 Generating A Feasible Implementation Model

Our approach to automatic synthesis of real-time systems requires automatic assignment of scheduling attributes such as priorities, preemption thresholds, and ceiling priority of shared resources. Same as in Chapter 4, we are facing the feasibility problem and optimization problem. In the previous section, we solved the problem of feasibility test for an implementation model with nominal thread assignment whose scheduling attributes are assigned. In this section, we focus on the solution to the feasibility problem. That is, designing a systematic approach to find an assignment of scheduling attributes that makes the implementation model with nominal thread assignment schedulable, if one exists.

5.3.1 Problem Statement

The feasibility problem becomes more complicated since the communication relationship and resource sharing introduces more variables to the design. Fortunately, our analysis shows that these variables are more or less dependent of the priority and preemption threshold. For a given priority assignment, the ceiling priority of each resource is fixed as well as their effect on worst-case response time of each action. Therefore, we actually need to assign only two sets of variables: the priorities and the preemption thresholds.

In the practice of real-time system design, usually there is an end-to-end timing constraint for each transaction. However, there is no explicit timing constraint for each action in the transaction, Here we choose a simple approach by assuming all actions in the same transaction share the same deadline, which is the end-to-end deadline of the transaction. Thus the problem of generating a feasible implementation model can be stated as follows:

> *Given a model $\mathcal{M} = \{\langle E_i, \mathcal{T}_i\rangle | (E_i \in \mathcal{E}_{ext}) \wedge (\mathcal{T}_i \in \mathcal{T}), 1 \leq i \leq n\}$, where each external event E_i is featured with an arrival function Ψ_i and triggers a transaction $\mathcal{T}_i$ that is featured with a deadline D_i. A transaction $\mathcal{T}_i$ is in turn represented as $\mathcal{T}_i = \{A_j \rightarrow A_k | A_j, A_k \in A\}$. For each action in A, A_j is featured with a worst-case execution time $C(A_j)$. Find whether there exists an assignment of scheduling attributes Π and Γ such that $\mathcal{M}$ is schedulable.*

5.3.2 Timing Properties of the Scheduling Model

Before we can start discussing the scheduling attribute assignment, it is necessary for us to have a better understanding about the scheduling behavior of the model and its effect on the response time of actions. In this section, we summarize several important timing properties that may serve as guidelines for assigning scheduling attributes to the model.

Lemma 5.2 *Raising the preemption threshold of action A_i from γ_1 to γ_2 may only reduce the worst-case response time of action A_i while increase the worst-case response time of those actions with priority between γ_1 and γ_2.*

Proof: As we have shown in the last section, the calculation of response times can be divided into 3 parts: blocking time, start time, and finish time. We will examine the effect of changing preemption threshold one by one.

From Equation 19, we can see that the maximum blocking time of A_i is determined by the preemption threshold of lower priority actions and the ceiling priority of their critical sections. The calculation in Equation 21 shows that the start time of A_i depends on the priority of A_i and has nothing to do with its preemption threshold. Therefore, the q^{th} start time of A_i won't be affected by raising its preemption threshold.

A raised preemption threshold will reduce the interference from higher priority actions shown in Equation 22 while keeping other terms unchanged. Of course this will result in a decreased finish time as well as worst-case response time.

At the same time, raising the preemption threshold of A_i from γ_1 to γ_2 will enable A_i to block actions with priority between γ_1 and γ_2. If A_i happens to be the action that contributes to the maximum blocking time for these actions, they will suffer an increase in their worst-case response times.

It is easy to see that other actions with priority lower than γ_1 (surely including those lower than $\pi(A_i)$), or higher than γ_2 will not be affected since no term in their equation will change. □

As a direct derivative, we get a very useful property of our model as shown in Corollary 5.1. As we mentioned before, there are two variables to be assigned to each action in the model. Both of them will affect the worst-case response time of the action. Their interaction complicates the search for a feasible assignment. The importance of Corollary 5.1 lies in the fact that it decouples the effect of these two variables to some extent thus simplifying the situation.

Corollary 5.1 *The worst-case response time of action A_i will not be affected by the preemption threshold assignment of any action A_j with $\pi(A_j) > \pi(A_i)$.*

Another important derivative of Lemma 5.2 is presented as Corollary 5.2. This corollary gives a sufficient condition for claiming a model to be unschedulable with a given priority assignment.

Corollary 5.2 *For any given priority assignment, if there exist an action A_i, such that setting the preemption thresholds of actions with lower priorities to the minimum schedulable* *value and setting γ_i equal to the highest priority in the system can not make the specific action A_i schedulable, then the model is unschedulable with this priority assignment.*

Theorem 5.4 *Suppose that a model is schedulable with given priorities and preemption thresholds. Then, if changing only the preemption threshold of A_i from γ_i to γ_i',($\gamma_i' < \gamma_i$) can still make A_i schedulable, then the whole model is also schedulable by setting γ_i' as the preemption threshold of A_i.*

Proof: From Equation 19, 21, and 22, we can see that when the preemption threshold of A_i changes from γ_i to γ_i' ($\gamma_i > \gamma_i'$), the worst-case response time of any action A_j with $\pi(A_j) < \pi(A_i)$ or $\pi(A_j) > \gamma(A_i)$ will not change. The worst-case response time of any action A_j with priority $\gamma_i' \geq \pi(A_j) > \pi(A_i)$ will also remain the same. Furthermore, the action A_j whose priority $\gamma_i' < \pi(A_j) \leq \gamma_i$ will have no worse worst-case response time with γ_i' than with γ_i. Moreover, we already know that A_i is schedulable with γ_i'. Therefore, if the model is schedulable with γ_i, it is also schedulable with γ_i'. □

5.3.3 Preemption Threshold Assignment with Pre-Assigned Priority

The problem of feasible assignment of scheduling attributes includes three sets of variables to be set: priority of each action, preemption threshold of each action, and ceiling priority of each critical section. However, we found that the ceiling priority for any critical section is determined by the priority assignment. Hence, with any given priority setting, there is only one ceiling priority assignment for each critical section while we are using Highest Locker Protocol (HLP). Therefore, we have only two sets of variables to assign.

As we can see from Chapter 3 and 4, the assignment of feasible priority and preemption threshold are closely related. A feasible assignment of either one can not be determined without the complete information about the assignment of the other. To some extent, we can say that the preemption threshold assignment depends on the priority assignment. As we did in the last chapter, we will first discuss the case of a system with given priority assignment. The problem seems trivial since we have already proved in the last subsection that the theories about our scheduling model with preemption threshold still hold with the introduction of communication relationship and resource sharing.

The properties shown above provide guidelines for preemption threshold assignment while priority setting is fixed. Corollary 5.1 suggests that the preemption threshold assignment process starts from the lowest priority action. Theorem 5.4 indicates that the optimal preemption threshold setting for the current action is the minimum value that makes the action schedulable. Furthermore, Corollary 5.2 provides a sufficient condition to claim a model to be unschedulable. To deal with the ceiling priority of each critical section, we just need to set it to the highest priority of the actions sharing the resource.

With these results, we found that the algorithm we proposed in Figure 4.1 is still valid for our general model with pre-defined priority. Of course, the worst-case response time calculation in the algorithm should be replaced by the equations we proposed in Section 5.2. The algorithm still has a complexity of $O(n^2)$, where n is the number of actions. Due to the run-to-completion scheduling within an active object (capsule), we have a restriction that the preemption threshold of actions in an active object should be no less than the highest priority among all actions in the same object. Therefore, we should start searching for the feasible preemption threshold from the ceiling priority of the object that an action belongs to. Thus, the search space is reduced.

5.3.4 Feasible Assignment of Scheduling Attributes

We face the problem of assigning both priorities and preemption thresholds. An exhaustive search is always a natural approach to solve a problem like this. However, the search space is exponential to the number of actions thus makes the exhaustive search not scalable and not practical in large systems. Using some properties of the model, we systematically construct a more efficient algorithm for finding the optimal solution.

In our general model, there are two restrictions imposed by the real world practice. First, the priority of an action is lower than or equal to its ancestor. Second, the preemption threshold of actions in an active object should be no less than the highest priority among all actions in the same object. Both restrictions reduce the search space, thus improving the performance of our algorithm. Again, we follow the basic logic used in Audsley's algorithm. As shown in Figure 5.5, we view our general model as a "forest", in which each transaction is a "tree". The nodes represent the actions and the edges represents the communication relationships. When we consider the candidates for the lowest priority, we only need to consider the leaf actions in our model. The term *leaf action* denotes the action that does not trigger the execution of any other action. Or in other words, the lowest priority action should be one of the ending actions of a transaction. For any specific priority, the candidates are actions with no successors or actions whose successors have all been assigned a priority. Figure 5.5 shows the situation while considering priority π_5. All

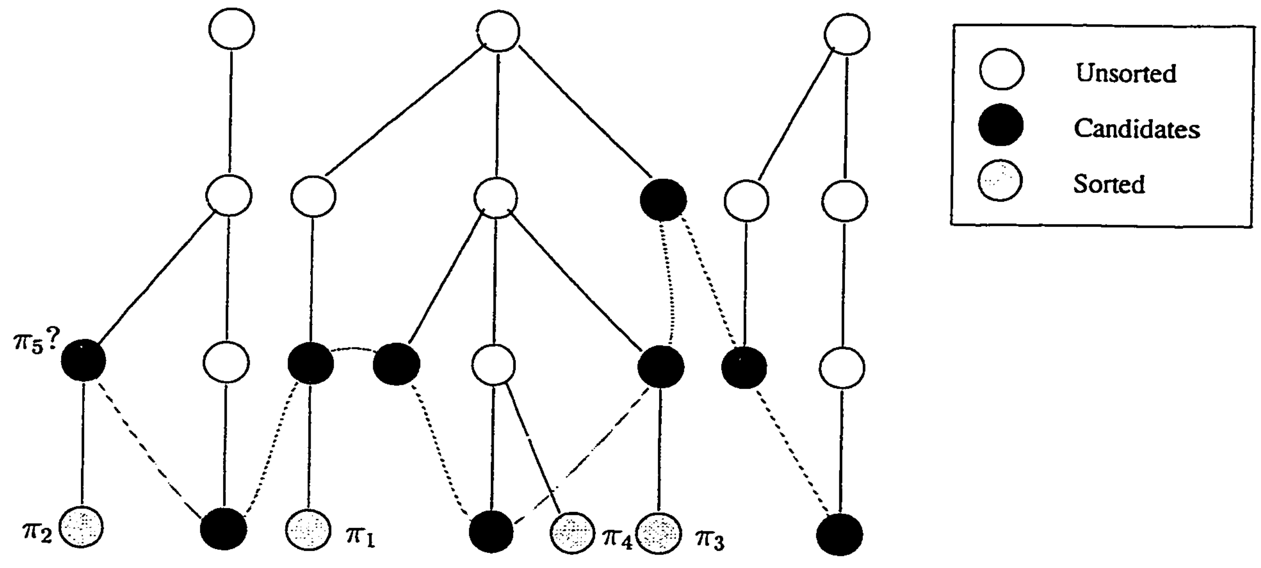


Figure 5.5: An Example of Candidates

the actions shaded the darkest (or in red if colored) are candidates.

Our algorithm is a two-stage algorithm. The first stage is trying to give a good priority assignment. We sort the candidates based on heuristic values hoping to reach the optimal solution quicker. If we only try the "best" candidate, then the algorithm becomes a greedy algorithm which can not guarantee optimality but will have a search space of $O(n)$. The logic we discussed in Section 4.2 for choosing the greedy algorithm is still valid so that we do not need to repeat it again. The second stage takes advantage of the efficient minimum preemption threshold assignment we discussed above for predefined priorities, which has a search space of $O(n^2)$.

5.3.5 Simulated Annealing

As we have already shown earlier, the simulated annealing approach has the power of providing a good approximation in a reasonable amount of time. To avoid the exponential search space of our algorithm, we again apply simulated annealing as a good approximate solution for the problem of feasible assignment of priority and preemption threshold.

To accommodate the general model, we need to modify our algorithm for simulated annealing shown in Figure 4.3. While the algorithm will remain the same for most parts, the starting priority setting and the structure of neighbourhoods should be changed. Since the performance of simulated annealing does not depend on the initial point, we can give an arbitrary priority assignment that satisfies the restriction that an action should have priority no higher than its ancestor. We can manually choose a priority ordering using the following easy approach. Starting from a randomly chosen transaction, we assign the leaf actions with the lowest priorities. We continue by recursively moving to their direct ancestor until we reach the root of the transaction. In this way, each action will be assigned a unique priority while satisfying the restrictions. Following the same process, we continue for each transaction until all actions are assigned a unique priority. Another approach may use the breadth first search to go through each tree and assign priorities so that it ensures priorities in a transaction are non-increasing from the root to the leaves. The second one may perform better since its neighbour is relatively larger than the first one at the initial stage. We use the same energy function defined before: $Max(0, \mathcal{R}(A_i) - D(A_i))$, where $\mathcal{R}(A_i)$ stands for the worst-case response time of A_i and $D(A_i)$ stands for the deadline of A_i.

The most difficult part of applying the simulated annealing approach to our general model is the design of a neighbourhood structure. We still want to generate neighbours by randomly swapping priorities of actions. However, a real random swap may conflict with the precedence restriction of the general model. Generating a valid neighbour is non-trivial in this case. The existing precedence in a transaction prevents any priority swapping that causes an action to have either higher priority than its ancestors or lower priority than its successors. We design the process of generating a valid neighbour as shown in Figure 5.7. The idea of the algorithm is quite simple, just trying to avoid the illegal swap of priority, which would undo the precedence defined within transactions. To some extent, the neighbourhood structure is similar to the candidate list shown in Figure 5.5.

To get the best performance from simulated annealing, the algorithm should have a fine tune-up on the initial temperature, equilibrium condition, and temperature cooling speed. This can be done through simulation of some examples.

5.4 Optimizing the Implementation Model

In the last section, we provide the solution for the feasibility problem. In this section, we focus on optimizing the model by reducing the number of preemptions, the number of threads, and the per-thread costs (especially the memory space associated with the stack of each thread). Ultimately, minimizing the number of threads gives rise to the other reductions. Just the same as with the simplified model, we use a decomposition approach to tackle this problem. First, we extend the algorithm for minimizing the number of threads in a valid thread assignment with given scheduling attributes. Then we show how to extend the sub-optimal approach we proposed in Chapter 4 to incorporate it into our general model. Since the "run-to-completion" manner of an object is already considered in the feasibility problem, the definition of a valid thread assignment in our general model remains the same as in the simplified model. Most importantly, the scheduling equivalence between a nominal thread assignment and a valid thread assignment shown in Theorem 3.4 still holds for our general model and thus provides a ground for the discussion in this section. Re-proving it in the context of our general model is trivial since it follows the same logic as in the simplified model.

5.4.1 Optimal Thread Assignment

The scheduling equivalence between a nominal thread assignment and any valid thread assignment enables us to keep the optimization process separate from the feasibility test. We merge the threads in a feasible implementation model with nominal thread assignment to minimize the number of threads in the implementation model while maintaining its schedulability. The advantages of reduced number of threads is prominent: minimized total memory space for stacks and reduced number of context switches.

Recall that in our simplified model, with a valid thread assignment, the jobs mapped to the same thread must be pair-wise mutually non-preemptive. The same condition is also applicable to the general model. This is done through proper setting of preemption threshold. To capture the notion, we redefined the non-preemptive group for the general model as follows:

Definition 5.8 (Non-Preemptive Group) *A set of actions $\mathcal{A} = \{A_{i_1}, A_{i_2}, \ldots, A_{i_m}\}$ forms a non-preemptive group if for every pair of actions $A_j \in \mathcal{A}$ and $A_k \in \mathcal{A}$, A_j and A_k are mutually non-preemptive.*

Minimum number of non-preemptive groups can still be formed using the **OPT-Thread** algorithm shown in Figure 4.4 by merging the actions based on their priority and preemption threshold settings. Although the introduction of communication relationships and resource sharing changed the appearance of the worst-case response time equations, it does not affect the correctness and optimality of the algorithm. It is trivial to prove the theorems again for the general model. Actions in the same non-preemptive group will be partitioned into one thread and share the same stack.

5.4.2 Compare to the Approach Following ROOM Convention

Capsules in UML correspond to the ROOM concept of actors, which are objects representing independent, concurrently active logical machines. In ObjecTime Developer, a tool-set supporting ROOM, threads are allocated to actors. Although our approach does not require all actions of the same active object to be allocated into the same thread, we have to explore how this tradition affects the thread assignment.

As we have discussed in preemption threshold assignment, the algorithm starts from the ceiling priority of the object that an action belongs to. This guaranteed that the actions in the same object naturally form a non-preemptive group. Following the convention of ObjecTime Developer, all actions of a single object should go to the same thread. Therefore, to reduce the number of threads, we have try to see whether it is possible for several objects to share the same thread. The problem becomes how to merge these non-preemptive groups, each representing an object, to reduce the number of threads required in the implementation. To solve this problem, we defined a *merging range* for each non-preemptive group as follows:

Definition 5.9 (Merging Range) *The merging range of a non-preemptive group G is defined as:* $[\max_{\forall A_i \in G}(\pi(A_i)), \min_{\forall A_i \in G}(\gamma(A_i))]$.

Using the concept of merging range, we define a merging policy for these non-preemptive groups as follows: if the merging ranges of two non-preemptive groups overlap, then we merge the two groups into one. It is easy to see that the actions in the new group are still pair-wise mutually non-preemptive. However, this merging policy defined to be consistent with the convention of ObjecTime Developer may not be optimal, i.e., it may results in a larger number of threads while compared with the **OPT-Thread** algorithm shown in Figure 4.4.

The following example indicates a situation where these two approaches result in different number of threads. Assume four actions A_1, A_2, A_3, and A_4 have blocking ranges of $[\pi_1, \gamma_1]$, $[\pi_2, \gamma_2]$, $[\pi_3, \gamma_3]$, and $[\pi_4, \gamma_4]$ respectively. Assume that we have $\gamma_1 > \gamma_2 > \pi_1 > \gamma_3 > \pi_2 > \gamma_4 > \pi_3 > \pi_4$. It is easy to see that the minimum number of non-preemptive groups is two. Now, if we know that $A_1 \in \mathcal{O}_1, A_2, A_3 \in \mathcal{O}_2, A_4 \in \mathcal{O}_3$ and we start merging objects, we will get 3 non-preemptive groups.

Although merging the objects may not achieve the minimum number of threads, there can be an option for the user. Since the minimum number of threads can be easily achieved by applying the **OPT-Thread** algorithm shown in Figure 4.4, it can be used as a reference for comparison with the number of threads results from object-based merging. In this way, the users are free to choose from either one according to their preferences.

5.4.3 Preemption Threshold assignment to Reduce Preemptions

As in the independent job model, there might be more than one feasible implementation models. We are facing a non-trivial combinatorial optimization problem: find the one with the minimum number of threads. The difficulty of searching through the space of feasible implementation models that we addressed in the simplified model still holds in the general model. Fortunately, we found the approach we used in the simplified model is also applicable to the general model. That is: first synthesize a feasible set of scheduling attributes as shown in the earlier section; then refine the scheduling attributes to eliminate any unnecessary preemptability while maintaining feasibility; and finally use the refined feasible scheduling attributes for minimum number of threads assignment.

Although we extended the model to incorporate transactions, communication relationships, and resource sharing, we found that most major properties of the simplified model still hold in our general model. This has been proved in previous sections of this chapter. These properties enable us to use the same algorithm shown in Figure 4.5 to increase the preemption threshold of each action to the maximum value while maintaining the feasibility of the implementation model. However, for the worst-case response time calculation in the algorithm, the equations we proposed in this chapter should be applied. The efficiency of the algorithm still holds, thus we only need to go through the actions (complete synchronous sets) in the implementation model once. As we can see, high preemption threshold may result in reduced number of preemptions and reduced number of threads in the implementation model and thus improves system performance.

```
Algorithm: AssignSchedAttributes($\mathcal{M}$, $\pi$)

(1)  $Candidate$ := GenerateCandidate($\mathcal{M}$);
     /* Terminating Condition; assign preemption thresholds */
(2)  if ($Candidate$ == {}) then
       /* Use algorithm in Figure 4.1 for preemption threshold assignment */
(3)    return AssignThresholds()
(4)  endif

     /* Heuristically generate a priority assignment */
     /* Assign Heuristic Value to Each Task */
(5)  L := {} ;
(6)  foreach $A_k \in Candidate$ do
(7)    $\pi_k := \pi$;   $\gamma_k := n$;   $\mathcal{R}_k$ := WCRT($A_k$);
(8)    if $\mathcal{R}_k > D_k$ then Continue ; /* prune */
(9)    $\gamma_k := \pi_k$ ;   $\mathcal{R}_k$ := WCRT($A_k$);
(10)   if $\mathcal{R}_k \leq D_k$ then
(11)     $H_k$ := GetBlockingLimit($A_k$);   /* positive value */
(12)   else
(13)     $H_k := \mathcal{D}_k - \mathcal{R}_k$;   /* negative value */
(14)   endif
(15)   L := L + $A_k$;
(16)   $\pi_k := n$ ;   /* reset */
(17) end

     /* Recursively perform depth first search */
(18) while (L != {}) do
       /* Select the job with the largest heuristic value next */
(19)   $A_k$ := GetNext(L) ;
(20)   $\pi_k := \pi$;
(21)   if AssignSchedAttributes($\mathcal{M}$, $\pi$+1) == SUCCESS then
(22)     return SUCCESS ;
(23)   endif
(24)   L := L - $A_k$;
(25)   $\pi_k := n$ ;   /* reset */
(26) end
(27) return FAIL
```

Figure 5.6: Optimal Assignment of Priority and Preemption Threshold for Extended Model

```
Algorithm: GenerateValidNeighbour($\mathcal{M}$)
(1)  randomly choose an action $A_i$ from model $\mathcal{M}$
(2)  randomly select another action $A_j$ that $\neg(A_i \leadsto A_j) \wedge \neg(A_j \leadsto A_i)$
(3)  if $(\forall A_k, A_k \leadsto A_i \Rightarrow \pi(A_j) \leq \pi(A_k)) \wedge (\forall A_k, A_k \leadsto A_j \Rightarrow \pi(A_i) \leq \pi(A_k))$
         $\wedge(\forall A_l, A_i \leadsto A_l \Rightarrow \pi(A_j) \geq \pi(A_l)) \wedge (\forall A_l, A_j \leadsto A_l \Rightarrow \pi(A_i) \geq \pi(A_l))$ then
(4)      swap $\pi(A_i)$, $\pi(A_j)$
(5)      return;
(6)  else
(7)      Goto (2);
```

Figure 5.7: Algorithm for Generating a Valid Neighbour

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

The demand for more complex real-time systems challenges the low-level and unscalable design methodology used by real-time community in history. Combining the state-of-the-art technologies in object-oriented software modeling, real-time scheduling, and automatic code generating provides a promising approach for efficient automatic synthesis of real-time software. This approach allows early consideration of timing issue in the real-time software design, reduces the ad-hoc decision making in the design process, and generates software that predictably meet the timing constraints.

To address the timing issues in this approach, we introduce a new scheduling model, namely fixed priority scheduling with preemption threshold, which subsumes both preemptive and non-preemptive scheduling models. This new scheduling model abstracts the dual-level scheduling behavior of real-time software systems running on multi-threaded real-time operating systems in industrial practice. With this model, we are able to assess feasibility of a system at design stage, automatically assign scheduling attributes to generate a feasible implementation model, as well as optimize the implementation model to reduce runtime overhead and memory requirement.

This thesis provides a profound analysis of the scheduling behavior of our new model emphasizing on two critical issues in our automatic synthesis approach, namely feasibility and optimization. We extend the busy period analysis in the literature by redefining the critical instant and level-i busy period. The equations for calculating worst-case response times and the theories summarizing the scheduling behavior of the model are presented. Using these theories as guidelines, we provide solutions for automatic synthesis and optimization of implementation models. Other benefits for designers using this new model include better schedulability compared with both preemptive and non-preemptive scheduling models and reduced run-time costs.

The automatic synthesis approach is illustrated both on a simplified model with independent jobs and a general model that features with end-to-end transactions, communication relationships, and resource sharing. While the computation time of the branch-and-bound algorithm for feasible implementation model generation grows exponentially with the number of tasks, simulated annealing provides a good approximation to the optimal solution in a reasonable amount of time.

A simulation tool is developed to implement the algorithm proposed for automatic synthesis and to simulate the run time behavior of the implementation generated by the automatic synthesis process. Using this tool, we provide quantitative assessment of the merits of our approach, including schedulability improvement, preemption reduction, and number of threads reduction. The performance evaluation shows significant improvements over the traditional preemptive and non-preemptive scheduling model.

The use of our automatic synthesis approach allows combination of the full benefit of code-generation technology and object-oriented modeling. Automatic synthesis largely releases a designer from the burden of choosing between various implementation artifacts (such as event priorities and mapping of events to threads), in much the same way that automatic code-generation releases a designer from the burden of deciding how to implement the modeling abstractions.

We believe the work presented in this thesis makes an important move in real-time system design. It will influence both future research in the area as well as the commercial world of real-time object-oriented CASE tools. Our implementations will not be immediately applicable to the product lines of the tool manufacturers for real-time systems, however, interests lie in the long-term objectives, when competition for tools that integrate real-time and object-orientation will be in high demand.

6.2 Future Work

While refining our approach of object-oriented software design for hard real-time uniprocessor systems, a number of extensions of this work are also under exploration. Case studies will surely provide examples for applying our approach in the real world and probably give hints for further improvements. While our current approach is based on a uniprocessor architecture, we are exploring the possibility of extending it to multi-processor and distributed system architectures. Since the main idea behind our approach is efficient automation, we are exploring the way to apply it online to handle changing workloads or timing requirements in a dynamic and adaptive real-time system.

Bibliography

[AKZ96] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach using OMT and Fusion.* Prentice Hall, 1996.

[AL77] A.E.Ritchie and L.S.Tuomenoksa. No.4 ESS: System Objective and Organization. *The Bell System Technical Journal*, 56(7):1017–1027, September 1977.

[Aud91a] N. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times. Technical Report YCS 164, Department of Computer Science, University of York, England, December 1991.

[Aud91b] N. Audsley. Resource Control for Hard Real-Time Systems: A Review. Technical Report YCS159, Department of Computer Science, University of York, August 1991.

[Bak90] T. Baker. A Stack-Based Resource Allocation Policy for Real-Time Processes. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 191–200. IEEE Computer Society Press, December 1990.

[BB99] I. Bate and A. Burns. An Approach to Task Attribute Assignment for Uniprocessor Systems. In *11th Euromicro Workshop on Real-Time Systems*, pages 46–53, June 1999.

[Bel98] R. Bell. Code Generation from Object Models. *Embedded Systems Programming*, 11(3), March 1998.

[BMS90] O. Babaoglu, K. Marzullo, and F.B. Schneider. Priority Inversion and its Prevention in Real-Time Systems. Technical Report TR-90-1088, Department of Computer Science, Cornell University, March 1990.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BS88] T. P. Baker and A. Shaw. The Cyclic Executive Model and Ada. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 120–129, December 1988.

[BW94] A. Burns and A. J. Wellings. HRT-HOOD: A Design Method for Hard Real-Time. *Real-Time Systems*, 6(1):73–114, 1994.

[Car84] G.D. Carlow. Architecture of the Space Shuttle Primary Avionics Software System. *Communication of the ACM*, 27(9):926–936, September 1984.

[Cen] Honeywell Technology Center. Metah user's manual. Available at http://www.htc.honeywell.com/metah.

[CL90] M.I. Chen and K.J. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. *The Journal of Real-Time Systems*, 2(4):325–346, November 1990.

[GHS94] R. Gerber, S. Hong, and M. Saksena. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 192–203, December 1994.

[Gom93] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Publishing Company, 1993.

[Gom00] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley Publishing Company, 2000.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. Technical Report N^o 2966, INRIA, France, sep 1996.

[Har87] D. Harel. Statecharts: A Visual Approach to Complex Systems. *Science of Computer Programming*, 1987.

[HG86] P. Hood and V. Grover. Designing Real Time Systems in Ada. Technical Report 1123-1, SofTech, Inc., January 1986.

[HKL91] M. Harbour, M. Klein, and J. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

[JJ82] J.Leung and J.Whitehead. On the Complexity of Fixed-priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *Computer Journal*, 29(5):390–395, 1986.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KRP$^+$93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.

[Lam] William Lamie. Preemption-threshold. White Paper, Express Logic Inc. Available at http://www.threadx.com/preemption.html.

[Leh90] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society Press, dec 1990.

[LL73] C. Liu and J. Layland. Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[Loc92] C. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4(1):37–53, March 1992.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[LW82] J. Leung and J. Whitehead. On the Complexity of Fixed-priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[Lyo98] Andrew Lyons. UML for Real-Time Overview. White Paper, Published by ObjecTime, and available from www.objectime.com, April 1998.

[Mok83] A.K. Mok. *Fundamental Design Problems for the Hard Real-Time Environments*. PhD thesis, MIT, 1983.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RSL89] R. Rajkumar, L. Sha, and J.P. Lehoczky. An Experimental Investigation of Synchronization Protocols. In *Proceedings of IEEE Workshop on Real-Time Operating Systems an d Software*, pages 11–17, May 1989.

[RSLR88] R. Rajkumar, L. Sha, J.P. Lehoczky, and K. Ramamithram. An Optimal Priority Inheritance Protocol for Real-Time Synchronization. Technical Report 88-98, Department of Computer and Information Science, University of Massachusetts, October 1988.

[SD88] S.R.Faulk and D.L.Parnas. On Synchronization in Hard-Real-Time Systems. *Communication of the ACM*, 31(3):274–287, March 1988.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[Sim90] H. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. In *IEE Proceedings Part E*, volume 137, pages 17–30, January 1990.

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In *Proceedings, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000.

[SM] S. Shlaer and S. Mellor. The Shlaer-Mellor Method. Technical Paper, Project Technology, Inc. 1996. Available from http://www.projtech.com.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. White Paper, Published by ObjecTime, and available from www.objectime.com, March 1998.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39:1175–1185, September 1990.

[TBW94] K. Tindell, A. Burns, and A. Wellings. An Extendible Approach For Analysing Fixed Priority Hard Real-Time Tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[VB93] S. Vestal and P. Binns. Scheduling and Communication in MetaH. In *Proceedings, IEEE Real-Time Systems Symposium*, 1993.