Mälardalen University Press Licentiate Theses
No. 160

RESOURCE AUGMENTATION FOR PERFORMANCE GUARANTEES IN EMBEDDED REAL-TIME SYSTEMS

Abhilash Thekkilakattil

2012

School of Innovation, Design and Engineering

ISBN 978-91-7485-086-4
ISSN 1651-9256
Printed by Mälardalen University, Västerås, Sweden

Abstract

Real-time scheduling policies have been widely studied, with many known schedulability and feasibility analysis techniques for different task models, that have advanced the state-of-the-art. Most of these techniques are typically derived under the assumption of negligible runtime overheads which may not be realistic for modern embedded real-time systems, and hence potentially compromises the guarantees on their correct behaviors. This calls for methods to reason about the functioning of the system under the presence of such overheads as well as to predictably control them. Controlling these overheads may place additional performance demands, consequently requiring more resources such as faster processors. At the same time, the need for energy efficiency in these class of systems further complicates the problem and necessitates a holistic approach.

In this thesis, we apply resource augmentation, viz., processor speed-up, to guarantee desired real-time properties even under the presence of runtime overheads. We specifically consider preemptions and faults that, at runtime, manifest as overheads in the system in various ways. Our aim is to provide specified non-preemption and fault tolerance feasibility guarantees in a real-time system. We first propose offline and online methods, that uses CPU frequency scaling, to control the number of preemptions in periodic and sporadic task systems, under a preemptive Fixed Priority Scheduling (FPS) policy. Furthermore, we derive the resource augmentation bound, specifically the upper-bound on the lowest processor speed, that guarantees the feasibility of a specified non-preemption behavior for any real-time task. We show that, for any task τ_i, the resource augmentation bound that guarantees a non-preemptive execution for a specified duration L_i, is given by $\frac{4L_i}{D_{min}}$, where D_{min} is the shortest deadline in the task set. Consequently, we show that the upper-bound on the lowest processor speed that guarantees the feasibility of a non-preemptive schedule for the task set is $\frac{4C_{max}}{D_{min}}$, where C_{max} is the largest execution time in the task set.

We then propose a method to guarantee specified upper-bounds on the preemption related overheads in the schedule. We first translate the requirements of meeting specified upper-bounds on the preemption related overheads to a set of non-preemption requirements for the task set. The resource augmentation bound in conjunction with a sensitivity analysis is used to calculate the optimal processor speed that guarantees the derived non-preemption requirements, achieving the specified bounds on the preemption related costs. Finally, we derive the resource augmentation bound that guarantees the fault tolerance feasibility of a set of real-time tasks under an error burst of known length. We show that if the error burst length is no longer than half the shortest deadline in the task set, the resource augmentation bound that guarantees fault tolerance feasibility is 6.

Our contribution bounds the extra resources, specifically the required processor speed-up, that provides specified non-preemption and fault tolerance feasibility guarantees in a real-time system. It allows us to quantify the 'goodness' of non-preemptive scheduling, referred to as its sub-optimality, as compared to an optimal uni-processor scheduling algorithm, in terms of the required processor speed-up that guarantees a non-preemptive schedule for any uni-processor feasible task set. We intend to extend this work to provide non-preemption and fault tolerance feasibility guarantees in multi-processor systems.

"Lokah Samastah Sukhino Bhavantu."

"May all beings everywhere be happy and free, and may the thoughts, words, and actions of my own life contribute in some way to that happiness and to that freedom for all."

This Sanskrit verse is an expression of the universal spirit found in the ancient Indian scriptures of Vedas.

Acknowledgements

It is hard to express in words my gratitude towards several people who have contributed directly or indirectly in enriching my life. However, today supposedly being an important day in my academic career, I take a moment to thank them for their kindness, love and affection.

Let me first thank my supervisors, Prof. Sasikumar Punnekkat and Dr. Radu Dobrin, for believing in me, and for their continuous support throughout my bachelors, masters and now the doctoral studies. Prof. Sasi has always supported me and motivated me to continue exploring my ways even when I was doubtful about them. Radu has been a great teacher - he taught me great many 'other' things, besides real-time systems, during the last four years, and we have had a wonderful time together during the many conference trips (I still remember the hunt for the hot peanuts in Hong Kong). Even though we rarely have a consensus regarding any matter, you have always been a role model to me. Let me also thank Vipin Sir for pushing me to pursue higher studies and for being a good teacher and friend, and for patiently listening to me whenever I have something to talk.

Next, I would like to thank my friends Vidhi and Deepthi for being partners in crime all these years and for the wonderful time that we have had together. Many thanks goes to Vidhi, Sebastian, Deepthi, Uma Chechi and Sajith for putting up with the several quick unplanned Amsterdam trips. I really miss you guys. Life would have been hard without the once in awhile 'chumma pingals' from Vidhi and the nice books and the long philosophical discussions from Sebastian. I thank Jezy chechi for motivating me to continue my higher studies at MDH and Paru for being a good friend and for cheering me up quite

A licentiate degree is a Swedish graduate degree halfway between MSc and PhD.

often.

My office-mates Adnan, Andreas J, Joe and Hus deserves special mention for the nice time that we have. Furthermore, many thanks goes to Gabriel, Jey, Yue, Andreas G, Fredrik E, Mikael Å, Jan C, Rafia, Saad, Guillermo, Svetlana, Moris, Leo, Aida, Hang, Thomas N, Irfan, Omar, Farhang, Nima, Eddy, Cristina, Dag and Barbara for the company during the lunch and/or conference trips. Special thanks to Adnan for helping me out with many practicalities and for being a good friend.

The learning process would have been incomplete without the nice Professors at IDT, especially Hans, Ivica, Damir, Gordana, Kristina and Bjorn. Heartfelt gratitude to Susanne, Carola and the other admin staff for the smooth and 'real-time' handling of the day-to-day practicalities. I also thank the other colleagues at IDT for lighting up the otherwise monotonous (winter) days.

I cannot leave out the football gang: Radu, Nikola, Adis, Murat, Fredrik A, Muhammed, Abdi, Nicho, Irfan, Omar, Alex, Simon, Pedram, Behnam, Federico, Fredrik E, Eddy, Dennis and the other irregulars, for making weekends so lively and fun- thanks to Radu and Nikola for organizing it every week. Also the badminton - pingpong gang: Svetlana, Saad, Gabriel, Dag, Severine, Irfan, Muhammed A, Andreas G., Bob and Leo, for making wednesdays seem like saturdays- thanks to Saad and Svetlana for organizing it.

A special word of thanks to Prof. Sasi, Sunitha Chechi and Kannan, for making me feel at home with the nice company and delicious food and also to Prakash Chettan, Mridula Chechi and little Mira for the good times.

Finally, I thank my parents and my sister for the love and affection and for the continuous support without which this thesis would have been impossible.

Abhi,
Västerås, November, 2012

The work presented in this thesis was partially supported by Vetenskapsrådet (The Swedish Research Council) under the project CONTESSE (2010-4276).

Publications

List of Papers Included in this Thesis[1]

Paper A: *Reducing the Number of Preemptions in Real-Time Systems Scheduling by CPU Frequency Scaling*, Abhilash Thekkilakattil, Anju S Pillai, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 18th International Conference on Real-Time and Network Systems, Toulouse, France, November, 2010

Paper B: *Probabilistic Preemption Control using Frequency Scaling for Sporadic Real-time Tasks*, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 7th International Symposium on Industrial Embedded Systems, IEEE, Karlsruhe, Germany, June, 2012

Paper C: *Quantifying the Sub-Optimality of Non-Preemptive Real-time Scheduling*, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, Technical Report, Mälardalen Real Time Research Centre, Mälardalen University, Västerås, Sweden, November, 2012

Paper D: *Resource Augmentation for Fault-Tolerance Feasibility of Real-time Tasks under Error Bursts*, Abhilash Thekkilakattil, Radu Dobrin, Sasikumar Punnekkat and Huseyin Aysan, In proceedings of the 20th International Conference on Real-Time and Network Systems, ACM, Pont á Mousson, France, November, 2012 (*Shortlisted for Best Student Paper Award*)

[1]The included articles are reformatted to comply with the licentiate thesis guidelines.

Other Relevant Papers

1. *Towards a Contract-based Fault-tolerant Scheduling Framework for Distributed Real-time Systems*, Abhilash Thekkilakattil, Huseyin Aysan and Sasikumar Punnekkat, In proceedings of the 1st International Workshop on Dependable and Secure Industrial and Embedded Systems, Västerås, Sweden, June, 2011

2. *Preemption Control using CPU Frequency Scaling in Real-time Systems*, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 18th International Conference on Control Systems and Computer Science, Bucharest, Romania, May, 2011

3. *Efficient Fault Tolerant Scheduling on Controller Area Network (CAN)*, Hseyin Aysan, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 15th International Conference on Emerging Technologies and Factory Automation, IEEE, Bilbao, Spain, September, 2010

4. *Towards Preemption Control Using CPU Frequency Scaling in Sporadic Task Systems*, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 6th International Symposium on Industrial Embedded Systems (WiP), IEEE, Västerås, Sweden, June, 2011

5. *Optimizing the Fault Tolerance Capabilities of Distributed Real-Time Systems*, Abhilash Thekkilakattil, Radu Dobrin, Sasikumar Punnekkat and Huseyin Aysan, In proceedings of the 14th International Conference on Emerging Technologies and Factory Automation (WiP), IEEE, Palma de Mallorca, Spain, September, 2009

List of Figures

1.1 Real-time task attributes 4
1.2 The dependability tree by Laprie et. al. [1] 7

2.1 (A) Preemptive real-time schedule (B) Non-preemptive real-time schedule 10
2.2 Fault tolerance related overheads for systems that employ temporal redundancy for fault tolerance 11

5.1 An offline detected *initial preemption* 39
5.2 An off-line detected *potential preemption* 39
5.3 Original RM schedule 45
5.4 RM schedule after eliminating one preemption 47
5.5 RM schedule after reducing preemptions using LOPF 47
5.6 Average number of *initial preemptions* after preemption elimination 49
5.7 Average number of *potential preemptions* after preemption elimination 49

6.1 Example probability mass function 71
6.2 A part of the original FPS schedule of the sporadic task set . . 71
6.3 The sporadic task schedule after preemption control 72
6.4 Average number of preemptions for various threshold probabilities 73
6.5 Average power consumption for various threshold probabilities 74

7.1 The feasibility bucket 88
7.2 Methodology overview 95

7.3 Non-preemption requirement to enable preemptions only at optimal preemption points 97
7.4 Non-preemption requirement to always enable critical section execution inside non-preemptive regions 98

8.1 The worst case error overhead due to error bursts on a single task 124
8.2 The maximum length of the burst error. 125
8.3 Error burst hitting multiple jobs 127
8.4 EDF schedule 136
8.5 EDF schedule under faults with $T_{length} = 4$ 136
8.6 EDF schedule under faults after a speed-up of 2.8 137

Contents

I Thesis **1**

1 Introduction **3**
- 1.1 Real-time Systems . . . 3
- 1.2 Real-time Scheduling . . . 4
- 1.3 Energy Awareness in Real-time Systems . . . 6
- 1.4 Dependability in Real-time Systems . . . 6
- 1.5 Resource Augmentation . . . 8

2 Motivation and Problem Description **9**
- 2.1 Motivation . . . 9
- 2.2 Problem Description . . . 12

3 Main Contributions **13**
- 3.1 Summary of Contributions . . . 13
  - 3.1.1 Paper A . . . 14
  - 3.1.2 Paper B . . . 14
  - 3.1.3 Paper C . . . 15
  - 3.1.4 Paper D . . . 17
- 3.2 Significance of the Contributions . . . 18

4 Conclusions and Future Work **21**
- 4.1 Conclusions . . . 21
- 4.2 Future Work . . . 22

Bibliography **25**

II Included Papers 29

5 Paper A:
Reducing the Number of Preemptions in Real-Time Systems Scheduling by CPU Frequency Scaling 31
5.1 Introduction . . . 33
5.2 Related Work . . . 34
5.3 System Model . . . 36
5.3.1 Task Model . . . 36
5.3.2 Energy Model . . . 37
5.3.3 Execution Time Model . . . 37
5.4 Methodology . . . 38
5.4.1 Preemption Identification . . . 40
5.4.2 Preemption Elimination . . . 40
5.5 Example . . . 44
5.6 Performance Evaluation . . . 47
5.6.1 Experiment 1 . . . 48
5.6.2 Experiment 2 . . . 48
5.6.3 Energy Consumption . . . 50
5.7 Discussion . . . 50
5.8 Conclusions and Future Work . . . 51
5.9 Acknowledgment . . . 52
Bibliography . . . 53

6 Paper B:
Probabilistic Preemption Control using Frequency Scaling for Sporadic Real-time Tasks 57
6.1 Introduction . . . 59
6.2 Related Work . . . 61
6.3 System Model . . . 63
6.3.1 Processor Model . . . 63
6.3.2 Task model . . . 63
6.3.3 Energy Model . . . 64
6.3.4 Execution Time Model . . . 64
6.4 Methodology . . . 65
6.4.1 Offline Phase . . . 65
6.4.2 Online Preemption Control Algorithm . . . 66
6.5 Example . . . 70
6.6 Evaluation . . . 72

6.7 Conclusions 73
Bibliography 75

7 Paper C:
Quantifying the Sub-Optimality of Non-Preemptive Real-time Scheduling 79
7.1 Introduction 81
7.2 System Model 83
7.2.1 Task model 83
7.2.2 Scheduling Model 84
7.2.3 Execution Time Model 85
7.3 Feasibility Analysis of Real-time Systems 85
7.4 Quantifying the Sub-Optimality of Non-Preemptive Scheduling 87
7.5 Guaranteeing a Specified Preemption Behavior using Processor Speed-up 94
7.5.1 Translating Preemption Cost Control Requirements to Non-Preemption Requirements 96
7.5.2 Sensitivity Analysis for Preemption Control 99
7.5.3 Example 101
7.6 Discussions 102
7.6.1 Relaxing the Assumption of Linear Speed-up 102
7.6.2 Relaxing the Assumption of Negligible Preemption Related Overheads at Optimal Preemption Points 105
7.7 Related Work 105
7.8 Conclusions 108
Bibliography 111

8 Paper D:
Resource Augmentation for Fault-Tolerance Feasibility of Real-time Tasks under Error Bursts 115
8.1 Introduction 117
8.2 Related Work 119
8.3 System Model 121
8.3.1 Task model 121
8.3.2 Scheduling Model and Fault Tolerance Strategy 122
8.3.3 Execution Time Model 122
8.4 Problem Description 123

8.5 Fault Tolerance Feasibility Analysis 123
8.6 Resource Augmentation for FT-Feasibility 132
8.7 Example 135
8.8 Conclusions 138
Bibliography 139

I

Thesis

Chapter 1

Introduction

1.1 Real-time Systems

Real-time computing paradigm is being ubiquitously deployed in many areas and is increasingly becoming the backbone of most mission and safety critical systems. A real-time system is a computer system where the correctness of the system depends not only on the functional characteristics of the computations performed, but also on its temporal characteristics [2]. The temporal characteristics of the computations are derived from the temporal properties of the events occurring in the environment that the real-time system interacts with. The events occurring in the environment, that may be periodic, sporadic or aperiodic, are mapped to a set of real-time tasks to perform a desired function within a bounded time interval. The real-time tasks are specified using a set of task attributes, that reflect the timing requirements of the corresponding events. In this thesis, we focus on periodic and sporadic real-time tasks that are characterized by a release time, an exact or a minimum inter-arrival time, a Worst Case Execution Time (WCET) and a relative deadline with respect to its release time, as shown in figure 1.1. Each task generates an infinite sequence of jobs where any two consecutive jobs are separated by the exact/minimum inter-arrival time. All the jobs generated by the real-time tasks have to complete their execution before their respective deadlines.

Depending on the consequences of a deadline miss in the system, real-time systems can be classified as hard real-time systems and soft real-time systems. In hard real-time systems, e.g., aircraft control systems, a deadline miss can potentially cause catastrophic consequences such as loss to life and

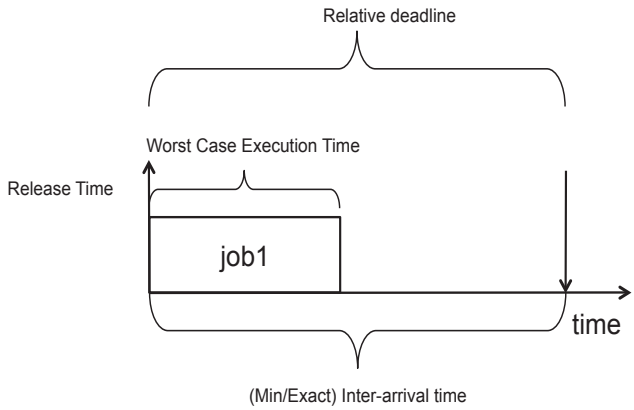


Figure 1.1: Real-time task attributes

property. On the other hand, a deadline miss in soft real-time systems, e.g., telecommunication systems, merely leads to a degradation of the service level.

1.2 Real-time Scheduling

Real-time scheduling can be classified as offline scheduling and online scheduling. In offline scheduling, the schedule is computed offline and is stored, typically in a table, and the tasks are executed according to this order. The main advantage of using offline scheduling algorithms is that much of the complexity involved in generating a valid schedule is handled offline and it has a simple runtime mechanism. However, the main disadvantage is that the schedule needs to be regenerated every time a new task is added to the system. On the other hand in online scheduling, the scheduling decision is taken online, based on a suitable criteria e.g., Earliest Deadline First (EDF) or highest priority first. The main advantage of using an online scheduling algorithm is the flexibility it provides to the scheduler in accounting for dynamically changing factors e.g., new task arrivals. Online scheduling, which is typically priority driven, can be further classified as static priority and dynamic priority scheduling. In static priority scheduling, also known as Fixed Priority Scheduling (FPS), the scheduling decisions are based on the priorities that are determined offline and the

task priorities does not change online e.g., Rate-Monotonic (RM) scheduling and Deadline Monotonic (DM) scheduling. In dynamic priority scheduling, the scheduling decisions depend on the priorities that change dynamically e.g., Earliest Deadline First (EDF) scheduling and Least Laxity First (LLF) scheduling.

One of the main goals with respect to the design of real-time systems is to provide temporal guarantees to the set of real-time tasks. A *schedulability analysis* is a technique designed for a particular scheduler that determines whether or not there will be a deadline miss in the schedule generated by that scheduler for any given task set. There exists several utilization based [3], response time based [4][5] and demand bound based [6][7] schedulability tests for various scheduling algorithms such as FPS or EDF. A *feasibility analysis* on the other hand determines the existence of a real-time schedule for a given task set, independent of the scheduler. However, many of the well known feasibility analysis techniques [6][7] build on the optimality of scheduling algorithms such as EDF [8] to determine the existence of a real-time schedule.

Real-time scheduling can also be classified as preemptive and non-preemptive scheduling. In preemptively scheduled systems, a lower priority task can be preempted in favor of a higher priority task. This enables preemptive schedulers to achieve significantly high processor utilization and preemptive scheduling is known to strictly dominate non-preemptive scheduling with respect to feasibility [9]. On the other hand in a non-preemptively scheduled system, the lower priority task is allowed to complete its execution before a higher priority task is scheduled. Non-preemptive scheduling schemes increase blocking times on higher priority tasks, which may have shorter deadlines, leading to a significantly under utilized system [9]. It can also be shown that, in general, non-preemptive scheduling can be infeasible even at arbitrarily low utilizations [10]. One of the main advantage of using a non-preemptive scheduler is the extend of determinism it provides to the system, due to the absence of preemption related overheads, that are typical of a preemptive scheduler.

Even though preemptive scheduling strictly dominates non-preemptive scheduling with respect to feasibility [9], it suffers from preemption related overheads that may translate as temporal overheads, causing deadline misses in the schedule. These preemption related overheads e.g., cache related preemption delays, may invalidate the arguments in many schedulability analysis schemes e.g., worst case response times at critical instant for FPS [11]. One of the ways to overcome this problem is to control the number of preemptions as well as the points at which preemptions occur, thereby bounding the associated overheads.

In chapter 2, we discuss in detail about the preemption related issues in

real-time systems, the pros and cons of using a non-preemptive scheduler and motivate the need to control preemptions.

1.3 Energy Awareness in Real-time Systems

The increasing use of more powerful processors coupled with the increase in the mobile nature of most real-time applications, necessitated sound methods to conserve energy in the usually processor intensive real-time applications. Also, the advances in battery technology did not keep its pace with the advances in processor technology requiring efficient utilization of the energy resources due to the increase in need for energy. Various methods like switching off the unused devices, selectively switching off certain circuits in the processor and Dynamic Voltage Scaling (DVS), were proposed for managing the power consumption of the system. Many of these methods were adopted fairly quickly by the industry with publication of standards like ACPI [12] which established an open standard for power management. DVS has been applied to energy constrained real-time systems to prolong its operational lifetime, while meeting the real-time requirements. Increasing the processor frequency reduces the task execution times in many applications, however, requiring higher operating voltages. The general strategy adopted for implementing DVS for reducing power consumption is to utilize the slack in the schedule to slow down task executions, by lowering the processor frequency and the applied voltage. This has been shown to provide significant reduction in the processor power consumption, which quadratically decreases with decrease in applied voltage and linearly with the frequency. While applying DVS techniques in real-time systems to save energy, one must ensure that the slowing down of task executions does not cause any deadline misses in the schedule.

The possibility of CPU frequency scaling provides for the ability to control task execution rates in a real-time system to influence the real-time schedule, making it one of the most important research area in real-time computing.

1.4 Dependability in Real-time Systems

Ubiquitous deployment of real-time systems in safety and mission critical applications necessitates high levels of dependability due to the catastrophic consequences of a failure. Dependability, as defined by Laprie et. al. [1], is the ability of a system to deliver a justifiably trusted service. The service delivered by the system is defined as the perceived behavior of the system by a user,

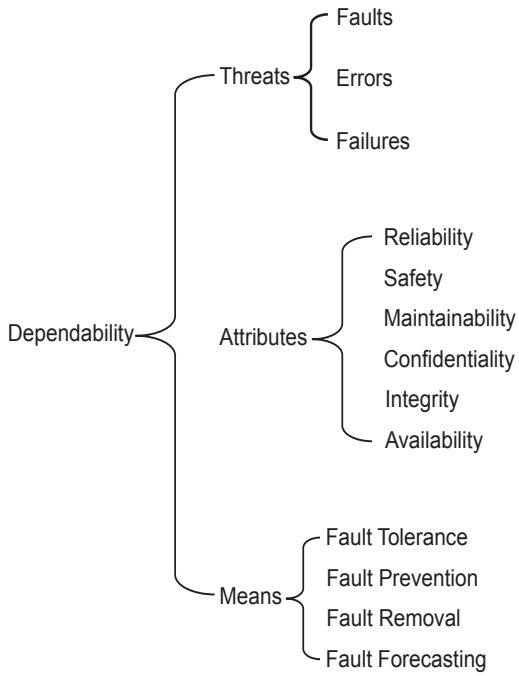


Figure 1.2: The dependability tree by Laprie et. al. [1]

which can be a human or another system, at the system interface. Laprie et. al. [1] identified the threats to dependability, the attributes of dependability and the means to achieve dependability in their famous dependability tree (refer figure 1.2). Availability, reliability, safety, integrity, confidentiality and maintainability together constitutes the attributes of dependability. The threats that affect the dependability of a system include faults, which is the hypothesized cause of an error, that may eventually lead to a failure of the system. An error in a subsystem can be considered as a fault in the system. Hence, the usage of faults and errors depends on the level of abstraction. Previous research on dependable real-time systems [13][14][15] focused on single errors per task/job, which may not be realistic [16]. A more realistic error model may be to consider error bursts e.g., a vehicle passing through a region of electromagnetic interference that makes any useful task execution impossible during that du-

ration [16]. One of the means of achieving dependability is to employ fault tolerance mechanisms, that can mask subsystem failures from the system.

Temporal redundancy is one of the most commonly employed fault tolerance strategy that can be used to tolerate transient and intermittent faults in the system by a simple execution of a recovery computation, which may be a re-execution of the failed computation or the execution of an alternate computation. The failed computations represent a temporal overhead in the system, potentially causing deadline misses, and can be seen as fault tolerance related overhead. In real-time systems, a task is typically considered as an independent unit of failure. Real-time fault tolerance aims at re-executing the failed task or executing an alternate task before a predefined deadline, which is usually the original deadline of the failed task, requiring sufficient slack in the schedule for fault tolerance. The main challenge here is to control these fault tolerance related overheads so that the feasibility of the task set can be guaranteed even under the presence of faults.

1.5 Resource Augmentation

Augmenting the scheduler with extra resource, e.g., a faster processor, can be beneficial for controlling the behavior of the system to guarantee specified requirements e.g., preemption control. While augmenting the scheduler with extra resources to achieve a specified behavior, the amount of extra resources required should be bounded by a reasonable value for it to be useful in practice. Resource augmentation analysis aims at finding the effects of having additional resources in the system and to find upper-bounds on the extra resources, to achieve a certain specified system behavior. Additionally, resource augmentation provides insights into the parameters that affect the satisfaction of the specified goal, giving greater flexibility to the system designer while designing the system. Kalyanasundaram et. al. [17] first introduced resource augmentation and proved that augmenting a non-clairvoyant online scheduler with a faster processor can effectively buy the power of clairvoyance.

In this thesis, we derive resource augmentation bounds that guarantee the feasibility a set of real-time tasks under preemption and fault tolerance related overheads and propose methods to control these overheads.

Chapter 2

Motivation and Problem Description

2.1 Motivation

The unpredictable and costly run time overheads in real-time systems, e.g., preemption related overheads and fault tolerance related overheads, can be controlled by changing the task attributes [18]. In this thesis, we control the task worst case execution times (WCET) and consequently the real-time schedule, by controlling the processor speed. Preemption related overheads can be controlled by controlling the number of preemptions and the points at which these preemptions occur. This can be achieved by controlling the task execution times such that preemptions occur only at specified points in the task code e.g., speed up the task such that it completes before a preemption. Fault tolerance feasibility under a given fault hypothesis can be achieved by using a faster processor which ensures that sufficient slack exists in the schedule to successfully schedule the recoveries of the failed tasks under an error burst of known length without causing deadline misses.

The detrimental impact of preemptions on task schedulability has received considerable attention from the research community [11] [19] and the need for preemption control is widely recognized. The most commonly considered preemption related overheads are:

1. **Context Switch Related Overheads:** Whenever a lower priority task is preempted by a higher priority task, the context of the preempted task

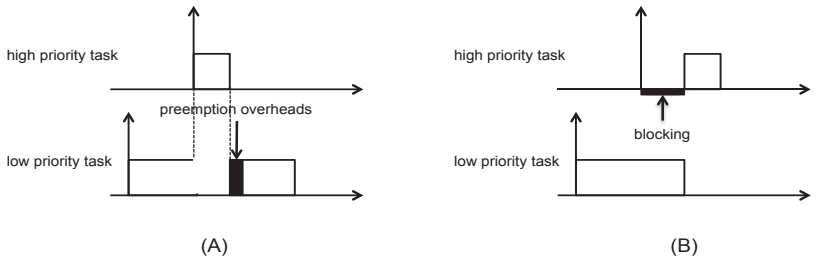


Figure 2.1: (A) Preemptive real-time schedule (B) Non-preemptive real-time schedule

is saved to enable its execution from the same point when it resumes its execution at a later time. The time taken to perform this context switch manifests as a temporal overhead in the schedule, potentially causing deadline misses.

2. **Cache Related Preemption Delays:** Preemptions can also displace data from the cache memory, which has to be reloaded when the preempted task resumes its execution. This translates to a temporal overhead, which can vary with the point of preemption [11]. The cache related preemption delays can be of the order of hundreds of micro seconds for a single preemption [20]. Consequently, the total temporal overhead on a task, due to preemptions, can be very high depending on the number of preemptions and the points at which the preemptions occur. Hence, bounding the number preemptions, as well as restricting them to only certain points in the task code is highly desirable.

3. **Pipeline Related Overheads:** Preemption on a lower priority task flushes the instruction pipelines, to load the instructions of the higher priority task. When the preempted task resumes its execution, the pipeline has to be refilled. This flushing and refilling of the instruction pipeline manifest as a temporal overhead in the system, leading to potential deadline misses.

Non-preemptive scheduling on the other hand increases blocking times on higher priority tasks leading to a significantly underutilized system. Consequently, non-preemptive scheduling can be prohibitively expensive for most

applications due to space and cost constraints. Hence, controlling preemptions in a real-time schedule can be an attractive option, i.e., enable non-preemptive execution of tasks as long as necessary, to reduce the indeterminism in the task execution times. As seen earlier, the preemption related overheads in a schedule depend on the number of preemptions in the schedule and the points at which the preemptions occur. Therefore, by controlling the number of preemptions and the points at which these preemptions occur, the associated overheads can be controlled.

Figure 2.1 shows examples of a preemptive and a non-preemptive schedule along with the associated temporal overheads.

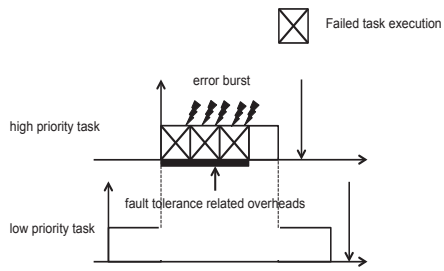


Figure 2.2: Fault tolerance related overheads for systems that employ temporal redundancy for fault tolerance

Implementing temporal redundancy requires that, in any time interval under a given fault hypothesis, there exists enough slack in the schedule to achieve one successful execution of all the tasks, that have their release times and deadlines within that interval. If enough slack does not exist during some time interval, there is a possibility of a deadline miss during that time interval. As mentioned earlier, the real-time tasks must respond to events occurring in the environment in a timely manner and hence, most of the real-time task attributes e.g., task periods and deadlines, are derived from the characteristics of these events, and are therefore strict. Consequently, one way of ensuring the existence of enough slack in the schedule to guarantee fault tolerance by temporal redundancy, is to use a faster processor. Finding such a processor speed can be done by performing a simple search among the set of available processor speeds. However, before performing a search, it may be desirable to verify if there exists a bound on the lowest processor speed that can guarantee fault tolerance. Hence, we need to find an upper-bound on the lowest processor speed

that guarantees fault tolerance feasibility using temporal redundancy under a given fault assumption. Figure 2.2 shows an example of the fault tolerance related overheads under an error burst, in systems that employ temporal redundancy.

Hence, in this thesis, we examine the use of resource augmentation to control the preemption and fault tolerance related overheads, to guarantee a desired preemption behavior and fault tolerance feasibility.

2.2 Problem Description

Our aim is to use resource augmentation, specifically a faster processor, to achieve preemption control and fault tolerance feasibility. Specifically, we consider the following questions:

Q1 How can we control preemptions in a periodic task system using CPU frequency scaling?

Q2 How can we control preemptions in a sporadic task system using CPU frequency scaling?

Q3 What are the resource augmentation bounds that guarantees the feasibility of a specified non-preemption behavior?

Q4 What are the resource augmentation bounds that guarantees fault tolerance feasibility of a set of real-time tasks under an error burst of known length?

We present methods to provide non-preemption and fault tolerance feasibility guarantees using resource augmentation. We use analytical and experimental approaches to evaluate our proposed methods.

Chapter 3

Main Contributions

Our contributions are presented in four papers which address various aspects of the problem. We first present an offline method to control preemptions in periodic task systems using CPU frequency scaling. We also present a combined offline-online approach to perform preemption control in sporadic task systems. We then derive the resource augmentation bound that guarantees the feasibility of a certain *specified preemption behavior* for periodic and sporadic task systems. Consequently, we derive the resource augmentation bound that guarantees the feasibility of a *fully non-preemptive* schedule. We also present a method to translate the requirements of meeting a specified upper-bound on the preemption related costs to a set of non-preemption requirements on the tasks. Later, we use the resource augmentation bound in a sensitivity analysis to calculate the exact processor speed that guarantees the feasibility of the derived non-preemption requirements for any task set. Finally, we apply resource augmentation to find upper-bounds on the processor speed-up that guarantees fault tolerance feasibility of a set of real-time tasks under an error burst of known length.

3.1 Summary of Contributions

In this section, summarize our solutions to the research questions raised in the previous chapter.

3.1.1 Paper A

Reducing the Number of Preemptions in Real-Time Systems Scheduling by CPU Frequency Scaling, Abhilash Thekkilakattil, Anju S Pillai, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 18th International Conference on Real-Time and Network Systems[1], Toulouse, France, November, 2010

Summary: Controlling the number of preemptions in real-time systems is highly desirable in order to achieve an efficient system design in multiple contexts. For example, the delays due to context switches account for high preemption overheads which detrimentally impact the system schedulability. Preemption avoidance can also be potentially used for the efficient control of critical section behaviors in multi-threaded applications. At the same time, modern processor architectures provide for the ability to selectively choose operating frequencies, primarily targeting energy efficiency as well as system performance. In this paper, we propose the use of CPU Frequency Scaling for controlling the preemptive behavior of real-time tasks. We present a framework for selectively eliminating preemptions, that does not require modifications to the task attributes or to the underlying scheduler. We evaluate the proposed approach by four different heuristics through extensive simulation studies.

My contributions: Main author of the paper and performed simulations. Anju S Pillai implemented the task generator and helped with the literature survey. All the co-authors contributed by participating in the discussions and in reviewing the paper.

Relation to the research questions: This paper, which proposes an offline preemption control method for periodic task systems, addresses research question Q1.

3.1.2 Paper B

Probabilistic Preemption Control using Frequency Scaling for Sporadic Real-time Tasks, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, In proceedings of the 7th International Symposium on Industrial Embedded Systems, IEEE, Karlsruhe, Germany, June, 2012

[1]This paper is also published in IEEE as ”Preemption Control Using Frequency Scaling in Fixed Priority Scheduling” in the IEEE/IFIP Embedded and Ubiquitous Computing Conference since RTNS-2010 had no copyright.

Summary: Preemption related costs are major sources of unpredictability in the task execution times in a real-time system. We examine the possibility of using CPU frequency scaling to control the preemption behavior of real-time sporadic tasks scheduled using a preemptive Fixed Priority Scheduling (FPS) policy. Our combined offline-online method provides probabilistic preemption control guarantees by making use of the release time probabilities of the sporadic tasks. The offline phase derives the probability related deviation from the minimum inter-arrival time of tasks. The online algorithm uses this information to calculate appropriate CPU frequencies that guarantees non-preemptive task executions while preserving the overall system schedulability. The online algorithm has a linear complexity and does not lead to significant implementation overheads. Our evaluations demonstrate the effectiveness of the method as well as the possibility of energy-preemption trade offs. Even though we have considered FPS, our method can easily be extended to dynamic priority scheduling schemes.

My contributions: Main author of the paper and performed simulations. All the co-authors contributed by participating in the discussions and in reviewing the paper.

Relation to the research questions: This paper, which proposes a combined offline-online preemption control method for sporadic task systems, addresses research question Q2.

3.1.3 Paper C

Quantifying the Sub-Optimality of Non-Preemptive Real-time Scheduling, Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat, Technical Report, Mälardalen Real Time Research Centre, Mälardalen University, Västerås, Sweden, November, 2012

Summary: Many preemptive real-time scheduling algorithms, such as the Earliest Deadline First (EDF), are known to be optimal on a uni-processor. However, no such algorithms exist under the non-idling non-preemptive scheduling paradigm. Hence preemptive schemes strictly dominate non-preemptive schemes with respect to feasibility. However, the 'goodness' of non-preemptive schemes in successfully scheduling feasible task sets when compared to uni-processor optimal preemptive scheduling schemes such as the EDF, which can

also be referred to as its sub-optimality, is unknown. In this paper, we apply resource augmentation, specifically processor speed-up, to quantify the sub-optimality of non-preemptive scheduling with respect to an optimal uniprocessor scheduling scheme such as the EDF. We also present a method to guarantee user specified upper-bounds on the preemption related costs in the schedule.

We prove that the speed-up required to guarantee the feasibility of a non-preemptive execution of any task τ_i, for a duration of L_i, is upper-bounded by $\frac{4L_i}{D_{min}}$, where D_{min} is the smallest relative deadline in the task set. Consequently, we show that the upper-bound on the processor speed that guarantees the feasibility of a non-preemptive schedule for the task set is $\frac{4C_{max}}{D_{min}}$, where C_{max} is the largest execution time in the task set. The derived upper-bound is used in a sensitivity analysis based method to calculate the optimal processor speed that guarantees a specified upper-bound on the preemption related costs in the schedule. For this, we first present a method to translate the system-level requirements of meeting specified upper bounds on the preemption related costs to task level non-preemption requirements. We then use sensitivity analysis technique to calculate the optimal processor speed that guarantees the feasibility of the derived task level non-preemption requirements, which in its turn guarantees the desired bounds on the preemption related overheads.

Our contribution quantifies the sub-optimality of non-preemptive scheduling in terms of the processor speed-up required to successfully schedule all the uniprocessor feasible task sets. It also enables a system designer to use a faster processor to guarantee specified upper-bounds on the preemption related overheads.

My contributions: Initiator and main author of the paper. All the co-authors contributed by participating in the discussions and in reviewing the paper.

Relation to the research questions: This paper derives upper-bounds on the lowest processor speed that guarantees the feasibility of a specified non-preemption behavior of a set of real-time tasks. Hence, we can obtain the upper-bound on the processor speed that can guarantee the feasibility of a fully non-preemptive schedule. Using these upper-bounds in a sensitivity analysis, exact processor speed-up that guarantees the feasibility of a specified non-preemption behavior is derived. The paper addresses research question Q3, while also providing a solution for questions Q1 and Q2.

3.1.4 Paper D

Resource Augmentation for Fault-Tolerance Feasibility of Real-time Tasks under Error Bursts, Abhilash Thekkilakattil, Radu Dobrin, Sasikumar Punnekkat and Huseyin Aysan, In proceedings of the 20th International Conference on Real-Time and Network Systems, ACM, Pont á Mousson, France, November, 2012 (*Shortlisted for Best Student Paper Award*)

Summary: Dependability is a vital system requirement, particularly in safety critical and mission critical real-time systems, due to the potentially catastrophic consequences of failures. In most critical applications different fault tolerance mechanisms using redundancy are employed to prevent possible failures. In the case of real-time systems the system designer must ensure that the task set is feasible even under faults, which we refer to as fault tolerance feasibility. Due to cost considerations, often temporal redundancy has been prevalently used to meet this objective. In this paper we focus on guaranteeing fault-tolerance feasibility under error bursts on uni-processor systems by the usage of resource augmentation, specifically through processor speed-up. Firstly, we derive a processor demand bound based sufficient condition for a set of real-time tasks to be fault tolerance feasible under an assumption that no more than one error burst occurs during the hyper-period of the task set. Subsequently, we derive the necessary resource augmentation bounds (i.e., the processor speed-up), that guarantees the fault tolerance feasibility, if the sufficient test fails. Finally, we prove that, if the error burst length is no more than half the shortest relative deadline of the task set, the processor speed-up required to guarantee fault tolerance feasibility is upper-bounded by 6.

My contributions: Initiator and main author of the paper. All the co-authors contributed by participating in the discussions and in reviewing the paper. The motivation of the work builds on Huseyin Aysan's PhD thesis [16].

Relation to the research questions: This paper derives upper-bounds on the processor speed-up required, that can guarantee fault tolerance feasibility of a set of real-time tasks under an error burst of known length. It addresses the research question Q4.

3.2 Significance of the Contributions

The optimality of preemptive and non-idling non-preemptive Earliest Deadline First (EDF) scheduling, under the respective assumptions, is well known [8] [21]. However, not all the task sets schedulable by preemptive EDF is schedulable by non-idling non-preemptive EDF, as preemptive scheduling strictly dominates non-preemptive scheduling [9]. Despite the domination of preemptive scheduling over non-preemptive scheduling, due to preemption related overheads, a preemptive schedule may not always be feasible and limiting preemptions [9] [22] may be necessary to guarantee feasibility. However, the limited preemption model [9] [22] may not always guarantee a *specified preemption behavior* e.g., guarantee a *specified upper-bound* on the number of preemptions or on the preemption related costs, because the largest non-preemptive region per task in any time interval is bounded by the processor demand in that interval. The dependence of the largest non-preemptive region of a task on the processor demand enables us to use a faster processor to control its length. Our upper-bound on the processor speed-up, guarantees a specified non-preemption behavior per task (ranging from preemptive to fully non-preemptive), achieving a certain generality. Hence, it opens up the possibility of augmenting the scheduler with a faster processor to guarantee the feasibility of the set of all uni-processor feasible task sets by a non-idling non-preemptive EDF scheduler. Consequently, it provides significant insights into developing a utilization based test for non-preemptive feasibility of periodic and sporadic real-time tasks.

A yet another interesting contribution of this thesis is the quantification of the sub-optimality of the non-preemptive scheduling scheme with respect to a uni-processor optimal preemptive scheduling scheme. As mentioned earlier in this section, preemptive scheduling strictly dominates non-preemptive scheduling with respect to feasibility. However, the 'goodness' of non-preemptive scheduling, when compared to a uni-processor optimal preemptive scheduling scheme, which is referred to as its sub-optimality, is unknown. In this thesis, we derive the resource augmentation bound that guarantees the feasibility of a non-preemptive schedule, for any set of real-time tasks that is uni-processor feasible. This allows us to quantify the sub-optimality of non-preemptive scheduling in terms of the processor speed-up required to guarantee the non-preemptive feasibility of any uni-processor feasible task set.

Bounding the processor speed-up that guarantees the fault tolerance feasibility of real-time tasks under an error burst of known length simplifies many design decisions, while building safety and mission critical real-time systems.

In cases where the original task set was feasible upon a uni-processor under an error free scenario, but is not feasible under an error burst, a system designer can opt for a faster processor to achieve fault tolerance feasibility under the error burst. We have shown that if the error burst length is no longer than half the shortest deadline of the task set, the lowest processor speed-up required to guarantee fault tolerance feasibility is upper-bounded by a constant 6. Hence, in this case, by increasing the processor speed by only a constant factor, the fault tolerance feasibility of a real-time task set can be guaranteed.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

In this thesis, we present methods to control preemption related and fault tolerance related overheads in a real-time system, using resource augmentation. We first present an offline preemption control method for periodic task systems, that derive individual task instance frequencies which reduces the number of preemptions in the schedule. Such an offline method is however not possible in a sporadic task system, when the task release times are not known offline. We hence propose a combined offline-online approach to control preemptions in a sporadic task system where the probabilities of the task releases are known. We first find the probability related deviation from the minimum inter-arrival times of the task. We then use this information in an online preemption control algorithm that, at any time instant, determines the maximum time for which the outstanding computations can execute non-preemptively and performs CPU frequency scaling to avoid a preemption. We hence demonstrate the feasibility of using resource augmentation to control preemptions in periodic and sporadic systems.

We derive the upper-bound on the processor speed-up required that guarantees the feasibility of a specified preemption (or alternately non-preemption) behavior of a set of real-time tasks. We show that the upper-bound on the processor speed-up that guarantees a non-preemptive region of length L_i for

any task τ_i is given by $\frac{4L_i}{D_{min}}$, where D_{min} is the smallest deadline in the task set. Consequently, we show that the upper-bound on the processor speed that guarantees the feasibility of a *fully non-preemptive* schedule is given by $\frac{4C_{max}}{D_{min}}$, where C_{max} is the largest execution time in the task set. However, changing the processor speed changes the required length of the non-preemptive region as well as its largest possible length. We finally use the derived bound to guarantee a user specified upper-bound on the preemption related costs. In this approach, we first present a method to derive a set of non-preemption requirements per task τ_i, which is the required length L_i of the non-preemptive region for τ_i, that guarantees the specified upper-bounds on the preemption related costs. Then, using a sensitivity analysis based approach, we derive the exact processor speed-up that guarantees the feasibility of the derived non-preemption requirement L_i for any such task τ_i.

Finally, we apply resource augmentation to guarantee the fault tolerance feasibility (FT-feasibility) of a set of real-time tasks under an error burst of known length during the hyper-period (LCM of time periods) of the task set. To calculate the bound, we first derive a sufficient condition that guarantees the feasibility of a set of real-time tasks under the error burst. We then derive the necessary bounds on the lowest processor speed-up required that guarantees FT-feasibility if the sufficient condition fails. We then show that if the error burst length is no longer than half the shortest deadline in the schedule, the required lowest processor speed-up is upper-bounded by a constant 6. Thus, by only a constant times increase in the processor speed, the FT-feasibility of a real-time task set can be guaranteed when the error burst length is no longer than half the shortest deadline.

4.2 Future Work

Some possible future work include:

- **Extensions to multi-processor scheduling:** Derive resource augmentation bounds that guarantees a specified non-preemption behavior for multi-processor systems.
- **Utilization based tests:** Develop utilization based tests for non-preemptive scheduling, and to determine the FT-feasibility of a real-time task set.
- **Redefining the notion of feasibility of real-time tasks:** Augmenting the notion of feasibility of real-time task sets with the extend of processor

speed-up required to guarantee the existence of a schedule even under the presence of overheads.

- **Contracts for preemption control and fault tolerance:** Incorporating the possibility of processor speed-up to derive online/offline contracts to achieve preemption control and fault tolerance feasibility in component based real-time systems.

Bibliography

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Securr Computing*, January 2004.

[2] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, October 1988.

[3] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *The Journal of ACM*, 1973.

[4] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 1986.

[5] N.C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[6] Sanjoy K. Baruah, Louis E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 1990.

[7] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *The 11th Real-Time Systems Symposium*, 1990.

[8] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, 1974.

[9] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *The 17th Euromicro Conference on Real-Time Systems*, 2005.

[10] Gang Yao, G. Buttazzo, and M. Bertogna. Comparitive evaluation of limited preemptive methods. In *The 15th International Conference on Emerging Technologies and Factory Automation*, 2010.

[11] H Ramaprasad and F Mueller. Tightening the bounds on feasible preemptions. In *The ACM Transactions on Embedded Computing Systems*, 2008.

[12] Advanced Configuration and Power Interface (ACPI): http://www.acpi.info/spec.htm [Last accessed:- 4 Jul 2012] .

[13] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *The 8th Euromicro Workshop on Real-Time Systems*, June 1996.

[14] H. Aydin. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Transactions on Computers*, October 2007.

[15] R.M. Pathan and J. Jonsson. Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks. In *The16th International Conference on Embedded and Real-Time Computing Systems and Applications*, April 2010.

[16] Huseyin Aysan. Fault-tolerance strategies and probabilistic guarantees for real-time systems. In *PhD thesis, Malardalen University*, June 2012.

[17] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 2000.

[18] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *The 16th Euromicro Conference on Real-time Systems*, 2004.

[19] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1995.

[20] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[21] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *The 12th IEEE International Real-time Systems Symposium*, 1991.

[22] M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *IEEE Transactions on Industrial Informatics*, November 2010.

II

Included Papers

Chapter 5

Paper A: Reducing the Number of Preemptions in Real-Time Systems Scheduling by CPU Frequency Scaling

Abhilash Thekkilakattil, Anju S Pillai, Radu Dobrin and Sasikumar Punnekkat
In proceedings of the 18th International Conference on Real-Time and Network Systems[1], Toulouse, France, November, 2010

[1]This paper is also published in IEEE as "Preemption Control Using Frequency Scaling in Fixed Priority Scheduling" in the IEEE/IFIP Embedded and Ubiquitous Computing Conference since RTNS-2010 had no copyright.

Abstract

Controlling the number of preemptions in real-time systems is highly desirable in order to achieve an efficient system design in multiple contexts. For example, the delays due to context switches account for high preemption overheads which detrimentally impact the system schedulability. Preemption avoidance can also be potentially used for the efficient control of critical section behaviors in multi-threaded applications. At the same time, modern processor architectures provide for the ability to selectively choose operating frequencies, primarily targeting energy efficiency as well as system performance. In this paper, we propose the use of CPU Frequency Scaling for controlling the preemptive behavior of real-time tasks. We present a framework for selectively eliminating preemptions, that does not require modifications to the task attributes or to the underlying scheduler. We evaluate the proposed approach by four different heuristics through extensive simulation studies.

5.1 Introduction

Preemptions in real-time scheduling may cause undesired high processor utilization, high energy consumption and, in some cases, even infeasibility. The preemption cost includes the direct costs to perform the context switches [1] and to manipulate the task queues [2, 1], as well as the indirect cost of cache-related preemption delays [3, 4]. The pessimism in many schedulability analysis methods could be reduced if an efficient control of the critical section behaviors can be established and preemption eliminations are ideally suited for achieving this.

Preemptive Fixed Priority Scheduling (FPS) has been extensively analyzed since the work of Liu and Layland [5], and is used in a large number of applications, mostly due to its flexibility and simple run-time overhead. In practice, however, preemptive FPS may imply large preemption related overheads and the need for preemption control is well recognized [6, 2, 7]. Buttazzo [8] showed that the rate monotonic algorithm (RM) introduces a higher number of preemptions than earliest deadline first algorithm (EDF).

Many techniques towards eliminating/minimizing preemptions have been proposed in literature [9, 10, 11, 12, 13, 14]. Most of the work focuses on reassigning task attributes like release times, deadlines, priorities etc and cannot be applied to those real-time tasks for which the task attributes such as priority, release times and deadlines reflect strict timing constraints. Alternative choices have to be developed for such systems where task attributes cannot be modified due to the inherent nature of the application involved. Butazzo has identified Quality of Service (QoS) [15] as one of the important research areas in future real time computer systems. Current methods for preemption elimination may reduce the quality of service as they change task attributes, which can result in an increased jitter or reduced levels of service due to late execution of tasks. Thus, removing preemptions without affecting QoS and without any modifications to the task attributes would be the ideal one from a system designer's perspective. At the same time, reducing the number of preemptions can also be beneficial from an energy point of view in systems with demands on low power consumption. When a task is preempted, there is a great probability that its contents in the cache will be lost. When the execution of the task is again resumed it will cause a lot of energy consuming accesses to off-chip memory. An access to off-chip memory is typically 10-100 times more expensive than an on-chip cache access in terms of energy consumption. Reducing the number of preemptions will reduce these additional expensive memory accesses due to reduced cache pollution.

Traditionally, Dynamic Voltage and Frequency Scaling techniques have been used for reducing energy consumption by slowing down tasks' executions [16, 17, 18, 19]. This is effective in reducing the energy consumption according to the relation $P = CV^2F$, where P is the power consumed by the processor, V is the applied voltage, C is the effective capacitance and F is the operating frequency. This means that the power dissipation increases/decreases linearly with frequency and quadratically with the applied voltage.

In this paper we apply CPU Frequency Scaling theory to control the preemption behavior in real-time system scheduling. We propose an offline method that identifies the maximum number of preemptions in a given schedule, and provides specific frequencies at which task instances need to be executed such that the preemptions are avioded. Our method is capable of guaranteeing a significantly lower number of preemptions without altering the original task attributes or modifying the underlaying scheduler. While executing tasks at a higher processor frequency may result in an increased energy demand, our method is capable of providing trade-offs between the number of preemptions and the overall energy consumption. While the methodology can be easily applied to any existing scheduling policy, in this paper we present an instantiation to FPS.

The main contributions of this paper consist of a) a formal analytical model to detect and eliminate preemptions using CPU Frequency Scaling by first detecting a preemption and then finding the minimum sufficient frequency that guarantees the completion of the preempted task before the release of the higher priority tasks, as well as b) a framework to study the effect of change in frequency at which task instances execute, over the rest of the schedule.

The rest of the paper is organized as follows: Section 5.2 describes the related work and in section 5.3 we give an overview of our system model. In Section 5.4 we describe our methodology illustrated by a simple example in Section 5.5. In Section 5.6 we present the experimental evaluation results and in Section 5.7 we discuss some important issues related to this paper. Finally, in Section 5.8 we present the conclusions and future work.

5.2 Related Work

Several methods have been proposed in the past to reduce the number of preemptions in real-time scheduling. Preemption Threshold Scheduling (PTS) for FPS was proposed by Wang and Saksena [11, 12], showing that this method improves schedulability as well as reduces the number of preemptions and the

number of threads in the system. In [11] the authors describe an optimal algorithm to assign preemption threshold by iterating over the solution and attempting to assign the largest feasible preemption threshold values to tasks such that the task set remains schedulable. The results show that large threshold values reduce the probability of preemptions and therefore should result in less preemptions. However, this approach results in a dual priority system which may not be directly suitable for, e.g., legacy systems, where scheduler modifications may not be possible.

The integration of real time synchronization schemes into PTS was proposed [20], where the authors integrate priority inheritance protocol and priority ceiling protocol into PTS. The authors have proposed two integrated schemes- in the first approach, instead of priority, the preemption threshold of a blocked task is inherited when blocking occurs; in the second approach, the priority ceiling is used instead of preemption threshold. The results show that the integrated schemes can minimize worst-case context switches and are appropriate for the implementation of real-time object-oriented design models.

Gai et al. [21] extend this scheduling model to EDF priority assignment and show that it can reduce the memory requirements of the system. In [22], the authors have presented an approach to combine PTS with Dynamic Voltage Scaling (DVS) to enable energy efficient scheduling. PTS decreases the number of context switches among tasks as well as the memory requirement in the system. Furthermore, the authors describe a dynamic slack reclamation technique, in conjunction with PTS, that yields energy gains depending on the available slack.

A method to integrate preemption threshold to FPS under DVS scheduling algorithms, was proposed in [13], where two preemption-aware algorithms, ccFPPT and FPPT-WD, are studied. ccFPPT is a cycle conserving fixed priority preemption scheduling, which slows down every task instance in its cycle or working range by the same amount. All the slack times are used to slowdown the processor speed. FPPT-WDA is the FPPT- Work Demand Analysis which is more complex compared to ccFPPT. The key feature of an online WDA DVS method is to postpone the release of the tasks as much as possible. Here, most of the slack time will be used for the first several tasks that discover these times leaving very tight, or even no scale down at all, for other tasks that arrive later.

In [14], the authors present two techniques that can reduce the increased number of preemptions introduced by using DVS algorithms. The first method is an accelerated completion based technique, where the main idea is to shorten the completion time of a low priority task before the arrival of a high priority task by accelerating its execution. The second approach is a delayed preemp-

tion based control technique, in which the activation point of a high priority task is delayed so that a scheduled low priority task can complete its execution without the preemption.

In an earlier work [9], we have proposed a method that analyzes offline a set of periodic tasks scheduled by FPS, and identifies the maximum number of preemptions that can occur at run time. It then reassigns task attributes, such as the task priority, period and offsets, without affecting the schedulability of the task set, while attaining a significantly lower number of preemptions. This is achieved at the cost of increased number of tasks and/or reduced task execution flexibility.

While the existing approaches have substantially advanced the state of the art in the field, all have either introduced potential infeasible costs or have focused on energy conservation when applying DVS. In this paper we propose the use of CPU frequency scaling to control the preemptive behavior in real-time scheduling without requiring modifications of the existing task attributes or to the underlaying scheduler.

5.3 System Model

5.3.1 Task Model

We assume a uniprocessor system implementing a preemptive fixed priority scheduling policy. We consider a periodic task set $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$ where task τ_i has a period T_i, a priority P_i, and a relative deadline D_i. The tasks are sorted in decreasing priority order, i.e., P_1 is the highest priority and P_n is the lowest. The hyperperiod of the tasks is defined by LCM representing the least common multiple of the task periods. Each task instance $\tau_{i,l}$ is characterized by a worst case execution requirement $C_{i,l}$, $i \in [1, n]$ and $l \in [1, \frac{LCM}{T_i}]$, at a discrete CPU frequency $F_p \in [F_{min}, F_{max}]$, where F_{min} and F_{max} are the minimum and maximum frequency respectively, as imposed by the hardware constraints. We assume that the tasks are initially executed at a default frequency supported by the hardware.

Additionally, we denote the release time of the l^{th} instance of task τ_i by $rel_{i,l}$, its corresponding actual start time by $start_{i,l}$, and its finishing time by $finish_{i,l}$. In the description of our method, we assume that the offsets are zero and the deadlines are equal to the periods. However, this restriction can be easily extended for non-zero offsets and deadlines shorter than periods. Finally, we assume that the tasks do not suspend themselves.

5.3.2 Energy Model

We consider a power-aware processor which can operate in a set of discrete operating modes identified by $M = \{m_1, m_2, m_3, ...m_p\}$, where each m_i is characterized by $m_q = (F_q, w_q)$, where F_q is processor frequency and w_q is the power (in watts) consumed by the processor in mode m_q [17]. We assume a negligible frequency-switch overhead, which may occur only in conjunction with a scheduling decision.

The total energy consumed by the system over the period of LCM can be represented as:

$$E_{LCM} = \sum_{i=1}^{n} \sum_{l=1}^{\frac{LCM}{T_i}} C_{i,l} \times w_{i,l}^{q} \tag{5.1}$$

where $w_{i,l}^{q}$ is the power consumed by the processor while executing the task instance $\tau_{i,l}$ in mode m_q at frequency F_q.

5.3.3 Execution Time Model

The execution time of a task instance is inversely proportional to the clock frequency at which the instance is executed, and can be represented as:

$$C_{i,j}^{1} = \frac{C_{i,j}^{max}}{F_1} \times F_{max}$$

where F_1 is the frequency which gives an execution time of $C_{i,j}^{1}$ and $C_{i,j}^{max}$ is the execution time obtained at F_{max}. This implies that,

$$F_1 = \frac{C_{i,j}^{max}}{C_{i,j}^{1}} \times F_{max} \tag{5.2}$$

Similarly to obtain an execution time of $C_{i,j}^{2}$ we require a frequency of:

$$F_2 = \frac{C_{i,j}^{max}}{C_{i,j}^{2}} \times F_{max} \tag{5.3}$$

Dividing the equation 5.2 by 5.3, we get:

$$\frac{F_1}{F_2} = \frac{C_{i,j}^{2}}{C_{i,j}^{1}}$$

which gives,

$$F_2 = \frac{C^1_{i,j}}{C^2_{i,j}} \times F_1 \tag{5.4}$$

This equation gives the frequency required for scaling $C^1_{i,j}$ to $C^2_{i,j}$. We have used this equation to derive the maximum frequency necessary to ensure a required worst case execution time for a particular task instance. This model is derived from the model presented in [18].

5.4 Methodology

In this paper we apply CPU Frequency Scaling theory to control the preemption behavior in fixed priority schedules. We propose an offline method that identifies the maximum number of preemptions in a given schedule, and provides specific frequencies at which task instances need to be executed such that the number of preemptions is reduced.

A preemption typically occurs when a higher priority task instance is released during the execution of a lower priority task instance. One way to avoid the preemption is to make sure that the preempted task instance completes its execution before the release of the higher priority one. As CPU Frequency Scaling can be used to speed up or slow down task execution times within a specified range, our method attempts to provide the minimum sufficient frequencies per task instance that guarantees preemption elimination.

In our offline preemption analysis we assume that tasks execute for their WCET. However, at run-time, tasks will most likely execute for less than WCET, implying a different number of preemptions compared to the ones detected by our off-line method. Hence, we divide the preemptions in two major categories:

Initial preemptions – are detected in the off-line analysis assuming task executions equal to their WCET, i.e., a high priority task instance is *initially preempting* a low priority task instance (Figure 5.1).

Potential preemptions – that occur at run-time due to task executions less than WCET. In Figure 5.2 a) we can see that if tasks execute for WCET, no preemption will occur. However, in this situation we consider task A *potentially* preempting task B since, if task C, that delays the execution of B, is executing for less than its WCET, then B can start executing earlier, i.e., before the release time of A, and will actually be preempted by A (Figure 5.2 b)).

In this paper we focus on the offline part of the methodology and, thus, we do not explicitly address the elimination of potential preemptions that would

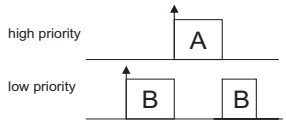


Figure 5.1: An offline detected *initial preemption*

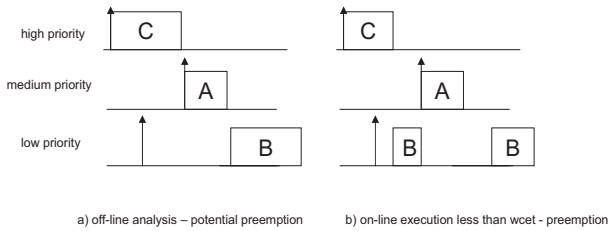


Figure 5.2: An off-line detected *potential preemption*

mostly benefit of the use of online mechanisms, and is the aim for future work. However, as later illustrated by the evaluation results, a large number of potential preemptions are automatically eliminated in the process of eliminating initial preemptions. At the same time, future work will address the use of online mechanisms for slowing down tasks at runtime, to ensure that the remaining potential preemptions are not converted to actual preemptions, as well as to compensate for the increase in energy for removing initial preemptions.

Our approach to eliminate a particular preemption is performed in two steps: preemption identification followed by the calculation of the minimum sufficient frequency at which the preempted task instance needs to execute in order to guarantee the preemption elimination. Obviously, the such frequency needs to be available, i.e., if the required frequency may not exceed the maximum available one, i.e., F_{max}, otherwise the preemption cannot be eliminated.

As the problem of finding the set of individual task instance frequencies to minimize the number of preemption for a given set of tasks is NP-hard, the significance of offline analysis lies in the fact that complex algorithms can be used to remove preemptions, which can complement the efforts to remove initial preemptions online.

In this paper we investigate and compare four different heuristics with respect to the order in which the preemptions are eliminated. We have examined the following four possibilities:

1. HPF – highest priority preempted task first
2. LPF – lowest priority preempted task first
3. FOPF – first occurring preemption first (under LCM)
4. LOPF – last occurring preemption first (under LCM)

In each of the approaches we attempt to eliminate the preemptions recursively until all preemptions are eliminated, or no feasible frequencies can be found for the remaining ones. Note that a preemption that cannot be eliminated at a particular stage, may be eliminated at a later iteration point in the algorithm, due to earlier completion of interfering tasks in the schedule.

5.4.1 Preemption Identification

We say that a task instance $\tau_{i,l}$ *initially* preempts another task instance $\tau_{j,k}$ if four conditions hold simultaneously [9]:

1. $\tau_{i,l}$ has a higher priority than $\tau_{j,k}$,
2. $\tau_{i,l}$ is released after $\tau_{j,k}$,
3. $\tau_{i,l}$ starts executing after the start time of $\tau_{j,k}$
4. $\tau_{j,k}$ finishes its execution after the release time of $\tau_{i,l}$

In case of nested preemptions we consider only the cases where a context switch occurs.

5.4.2 Preemption Elimination

To eliminate a single preemption, e.g., $\tau_{i,l}$ preempts $\tau_{j,k}$, we identify the new frequency at which the preempted instance must execute, by calculating the execution reduction that guarantees its completion before the release of the higher priority instance, i.e,:

$$finish_{j,k}^{new} \leq rel_{i,l}$$

Where, $finish_{j,k}^{new}$ is the new finishing time of the preempted instance after preemption elimination.

Theorem 5.4.1. *Given a preemption where $\tau_{i,l}$ preempts $\tau_{j,k}$, the worst case execution time of $\tau_{j,k}$ that guarantees the preemption avoidance is given by the relation:*

$$C_{j,k}^{new} = finish_{j,k}^{new} - start_{j,k} - I_{j,k} \tag{5.5}$$

The interference $I_{j,k}$ is given by:

$$I_{j,k} = \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x}$$

where

$$\Psi(j,k,l,x) = \begin{cases} 1, & if\ start_{j,k} < start_{l,x} < finish_{j,k}^{new} \\ 0, & otherwise \end{cases} \tag{5.6}$$

Proof. The $finish_{j,k}$ for a task instance $\tau_{j,k}$ is obtained by adding its execution time and the interference caused due to preemptions by higher priority tasks to its start time:

$$finish_{j,k} = start_{j,k} + C_{j,k} + \sum_{\forall \tau_{l,x} \in \Gamma_2'} C_{l,x}$$

Where Γ_2' is the set of all higher priority task instances released between the start time and finish time of $\tau_{j,k}$. Γ_2' can be found by a recursively checking whether any higher priority task instances start between the start time and the latest computed finish time of $\tau_{j,k}$ with the $finish_{j,k}$ initially set to $(start_{j,k} + C_{j,k})$ .Thus, we rewrite the above equation as:

$$\begin{aligned} finish_{j,k} &= start_{j,k} + C_{j,k} + \\ &\quad + \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x} \end{aligned} \tag{5.7}$$

where

$$\Psi(j,k,l,x) = \begin{cases} 1, & if\ start_{j,k} < start_{l,x} < finish_{j,k} \\ 0, & otherwise \end{cases}$$

with $finish_{j,k}$ set to $start_{j,k} + C_{j,k}$ initially.

We rewrite (5.7) as:

$$finish_{j,k} = start_{j,k} + C_{j,k} + I_{j,k}$$

Where $I_{j,k}$ is given by:

$$I_{j,k} = \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x}$$

and $\Psi(j,k,l,x)$ is given by the earlier equation. Now rearranging the terms we get:

$$C_{j,k} = finish_{j,k} - start_{j,k} - I_{j,k}$$

Here we substitute the new required finish time $finish_{j,k}^{new}$ such that the preemption on $\tau_{j,k}$ by $\tau_{i,l}$ is eliminated:

$$C_{j,k} = finish_{j,k}^{new} - start_{j,k} - I_{j,k}$$

□

After calculating the new execution time required to eliminate a single preemption, we check whether it is possible to speed up the execution of the task instance to guarantee this execution time by checking whether the corresponding frequency range is within the CPU permitted range. This is done by first calculating the required CPU frequency, denoted by F_r, using the formula:

$$F_r = \frac{C_{j,k}^{cur}}{C_{j,k}^{new}} \times F_q$$

where $C_{j,k}^{new}$ is the execution time of k^{th} instance of task τ_j to finish before it is preempted by a higher priority task, and $C_{j,k}^{cur}$ is its execution time before removing the preemption, when executing at a frequency F_q. The calculated F_r is approximated to the nearest discrete value among the values which the processor can attain, and the old frequency is retained if $F_r \notin [F_{min}, F_{max}]$.

Finally, we need to investigate the impact of the preemption elimination on the rest of the schedule by recalculating the start times and finish times of all lower priority task instances, according to the equations 5.8 and 5.11, when

$$F_r \in [F_{min}, F_{max}]$$

Theorem 5.4.2. *The start time of any task instance $\tau_{j,k}$ is given by,*

$$start_{j,k} = max(f_{hp}(j,k), rel_{j,k}) \tag{5.8}$$

where,

$$f_{hp}(j,k) = max_{\forall l \in hp(j)}(\, finish_{l,\lceil \frac{f_{hp}(j,k)+1}{T_l} \rceil}) \tag{5.9}$$

and, initially,

$$f_{hp}(j,k) = rel_{j,k} \tag{5.10}$$

Proof. According to our assumption, a task instance $\tau_{j,k}$ starts its execution if it is released, and after all high priority tasks in the ready queue have finished execution. This has two cases,

Case 1 : The ready queue is empty at $rel_{j,k}$ and no higher priority tasks are released simultaneously or are currently executing

Case 2 : There exists at least one high priority task instance the is released, or is currently executing at $rel_{j,k}$

The value computed by $\frac{f_{hp}(j,k)+1}{T_l}$ will give the latest instance number of all higher priority tasks τ_l. Using this instance number, equation 5.9 will return the maximum of the finish times of the corresponding high priority task instances. This is done recursively until a single value is obtained.

Consider Case 1, where no high priority tasks are executing/released or in the ready queue at $rel_{j,k}$. The value computed by 5.9 will be less than $rel_{j,k}$, since the latest of the higher priority task instances would have already completed. So equation 5.8 will return $rel_{j,k}$ as the start time of $\tau_{j,k}$. Hence, the equation 5.8 holds for Case 1.

Consider Case 2, where there exists at least one higher priority task that is currently executing at the time when $\tau_{j,k}$ is released. The value computed by 5.9 will be greater than $rel_{j,k}$, since 5.9 computes the latest of the finish times of all high priority tasks that are released in the busy period before $\tau_{j,k}$ starts executing. This finish time is the start time of $\tau_{j,k}$ as we have assumed that no task can suspend itself. Hence the equation 5.8 also holds for Case 2. □

Finally, we calculate the finish time of $\tau_{j,k}$.

Theorem 5.4.3. *The finish time for a task instance $\tau_{j,k}$ is given by the equation:*

$$finish_{j,k} = start_{j,k} + C_{j,k} + I_{j,k} \tag{5.11}$$

$I_{j,k}$ is given by:

$$I_{j,k} = \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x} \tag{5.12}$$

where $\Psi(j,k,l,x)$ is given by the equation:

$$\Psi(j,k,l,x) = \begin{cases} 1, & if\ start_{j,k} < start_{l,x} < finish_{j,k} \\ 0, & otherwise \end{cases} \tag{5.13}$$

Proof. The proof is similar to the one of theorem 5.4.1. □

Recalculation of the start times and finish times aims to investigate the impact of one preemption elimination on the rest of the schedule, i.e., whether any new preemptions have been introduced or any additional "old" preemptions have been removed. Additionally, a schedulability test is performed in order to ensure the task completions before their deadlines.

$$\forall i \in [1,n], j \in [1, \frac{LCM}{T_i}], finish_{i,j} \leq (j-1) \times T_i + D_i$$

In this paper we use 4 different heuristics, i.e., HPF, LPF, LOPF, FOPF, to recursively eliminate the preemptions in a given set of tasks, schedulable by preemptive FPS, until all preemptions are eliminated or no feasible solutions can be found for the remaining ones.

5.5 Example

We illustrate the proposed preemption reduction method with a simple example. We assume a set of tasks as described in the Table 5.1 scheduled according to the rate monotonic scheduling policy, using a default frequency of 40 MHz provided by the hardware. The time used in the example is expressed in milliseconds (ms). In this example, our method identifies 7 initial preemptions that may occur at run time (Figure 5.3) when the tasks execute for their worst case execution times.

For explaining how a single preemption is detected, eliminated, and its effects over the rest of the schedule, we describe the removal of the preemption of C_1 by A_4. The preemption is detected as it satisfies the following condition:

$$\{P_A > P_C\} \wedge \{rel_{A,4} > rel_{C,1}\} \wedge \{start_{A,4} > start_{C,1}\}$$

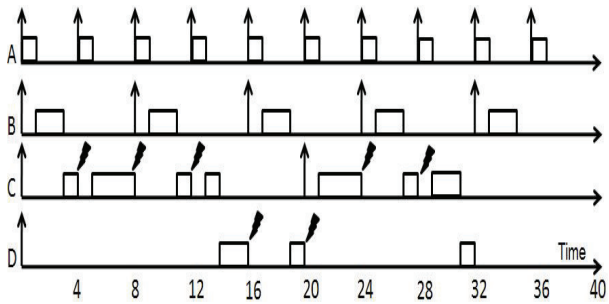


Figure 5.3: Original RM schedule

Task	Time period	Execution Time	Priority
A	4	1	1(highest)
B	8	2	2
C	20	6	3
D	40	4	4

Table 5.1: Example: task set

$$\wedge\{finish_{C,1} > rel_{A,4}\}$$

To eliminate the preemption, C_1 needs to complete before the release of A_4.

$$finish_{C,1}^{new} \leq rel_{A,4} = 12$$

Consequently, the new execution time for C_1 is calculated using the equation 5.5:

$$C_{C,1} = 12 - 3 - (1 + 1 + 2) = 5.$$

At this point, we need to check the possibility of eliminating this preemption by ensuring that the corresponding frequency is within the permissible range. For the analysis we take a variable frequency processor having different operating modes as described in Table 5.2 [17].

We find the frequency at which the task instance C_1 must execute to eliminate it being preempted by A_4 using equation 6.2.

$$F_2 = \frac{6}{5} \times 40 = 48$$

Mode	0	1	2	3	4	5
Frequency(MHz)	0	5	30	40	50	80
Power Consumption(mW)	0	20	50	50	200	500

Table 5.2: Example: CPU operating modes

This is approximated to 50 MHz which is the next highest frequency supported, which can guarantee this execution time. C_1 will execute for 4.8 ms when it is run at 50 MHz.

Eliminating this particular preemption will affect the lower priority task's start times and finish times. Hence, we re-calculate the start times and finish times of all the lower priority task instances based on the newly calculated execution and finish time of C_1, according to the equation 5.11:

$$finish_{C,1} = 3 + 4.8 + (1 + 1 + 2) = 11.8$$

We calculate the start time of D_1 using equation 5.8:

$$\begin{aligned} f_{hp}(D,1) &= rel_{D,1} = 0, initially \\ f_{hp}(D,1) &= max(finish_{A,\lceil \frac{0+1}{4} \rceil}, finish_{B,\lceil \frac{0+1}{8} \rceil}, \\ &\quad finish_{C,\lceil \frac{0+1}{20} \rceil}) \\ &= max(finish_{A,1}, finish_{B,1}, finish_{C,1}) \\ &= max(1, 3, 11.8) = 11.8 \end{aligned}$$

Since $0 \neq 11.8$, we recursively calculate the new value for the start time for D_1 until we reach a fixed point. Here the start time of D_1 is 11.8. The newly computed value of the finish time of D_1 is:

$$finish_{D,1} = 11.8 + 4 + (1 + 1 + 2) = 19.8$$

It is now possible to do a schedulability analysis on the finish times of the task instances or find the total number of preemptions and take a decision on whether or not to eliminate this preemption. After the removal of the preemption of C_1, the total number of preemptions in the schedule is reduced to 6 (figure 5.4). We use this process to eliminate preemptions according to the last occurring preemption first strategy (LOPF) as described in section 5.4, until no

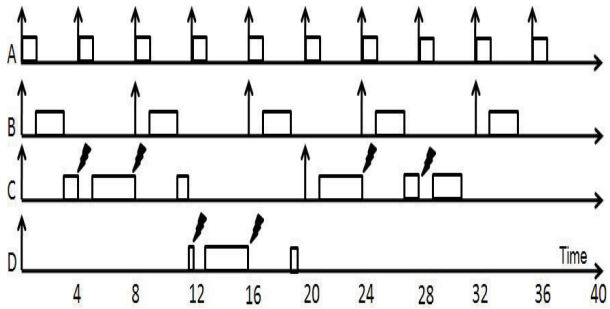


Figure 5.4: RM schedule after eliminating one preemption

more preemptions can be eliminated. Figure 5.5 shows the resulting schedule eliminating preemptions in the reversed order of their occurrences in the schedule. The number of initial preemptions is reduced from 7 to 2, with a cost of 2.7 times increase in energy, according to equation 6.1. Finally, the frequencies

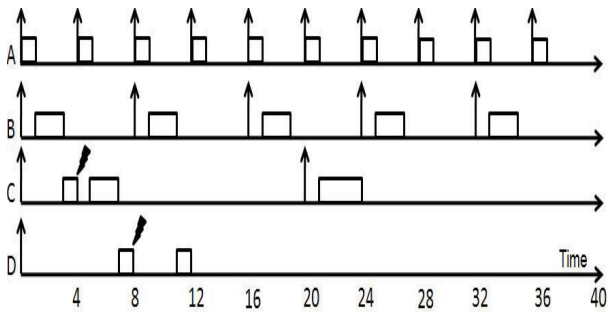


Figure 5.5: RM schedule after reducing preemptions using LOPF

for all the task instances are computed and illustrated in table 5.3.

5.6 Performance Evaluation

We performed a number of experiments to evaluate the efficiency of our proposed method. We used synthetic tasks with randomly generated attributes,

Instance	1	2	3	4	5	6	7	8	9	10
τ_A	40	40	40	40	40	40	40	40	40	40
τ_B	40	40	40	40	40	-	-	-	-	-
τ_C	80	80	-	-	-	-	-	-	-	-
τ_D	80	-	-	-	-	-	-	-	-	-

Table 5.3: Derived frequencies for each task instances

schedulable by FPS. We studied the effect of the removal of preemptions in different orders. We generated task sets with utilizations ranging from 0.6 to 1.0 using the UUniFast [23] algorithm that were used to compare the efficiency of the different approaches. The tasks priorities were assigned according to the RM policy. Each set consisted of 5 to 15 tasks respectively, with time periods ranging from 5 to 1500. For the purpose of obtaining integer values of execution times, we assumed that the calculated CPU frequency is supported by the processor. However we assumed that the tasks cannot be scaled to a value less than 60% of their actual execution times i.e., any value above 60% of the original execution time was deemed acceptable, and those below unacceptable.

5.6.1 Experiment 1

In this scenario, we experimented the preemption elimination *based on the task priority order*. We performed rwo different runs for each taskset. In the first run we eliminated the preemptions starting with the one incurred by the first instance of the highest priority task to the last instance of the lowest priority task i.e., highest priority first (HPF), in the order $\{\tau_{1,1}, \tau_{1,2}, ...\tau_{1,\frac{LCM}{T_1}}\}$, $\{\tau_{2,1}, \tau_{2,2}, ...\tau_{2,\frac{LCM}{T_2}}\}$, $....\{\tau_{n,1}, \tau_{n,2}, ...\tau_{n,\frac{LCM}{T_n}}\}$. In the second run, we eliminated the preemptions starting from the first instance of the lowest priority task to the last instance of the highest priority task i.e, lowest priority first (LPF), in the order $\{\tau_{n,1}, \tau_{n,2}, ..\tau_{n,\frac{LCM}{T_n}}\}$, $..\{\tau_{2,1}, \tau_{2,2}, ..\tau_{2,\frac{LCM}{T_2}}\}$, $\{\tau_{1,1}, \tau_{1,2}, ..\tau_{1,\frac{LCM}{T_1}}\}$.

5.6.2 Experiment 2

In this experiment we eliminated the preemptions in the *order of their occurrence in the schedule*. We conducted two runs, where in the first run we removed preemptions from the first occurring preemption to the last (FOPF) and in the second run from the last occurring preemption to the first (LOPF). Our

simulations results for the four heuristics used in the 2 experiments are illustrated in Figure 5.6.

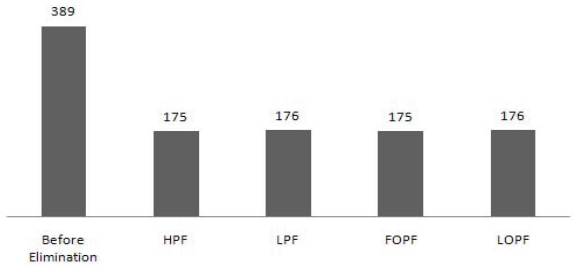


Figure 5.6: Average number of *initial preemptions* after preemption elimination

We also observed that a significant number of potential preemptions are also eliminated automatically in the process of removing initial preemptions (Figure 5.7). In this case, LPF and LOPF performed slightly better with respect to reducing the number of potential preemptions.

Figure 5.7: Average number of *potential preemptions* after preemption elimination

In some cases, some preemptions of medium priority tasks are automatically eliminated which will not result in a reduction of execution times of those medium priority tasks. However while eliminating preemptions in the LOPF,

the preemptions which are removed automatically in the other three cases are detected and eliminated first. This result in a reduction in execution times of these medium priority task instances. As a result of this the preemptions (both initial and potential) of the lower priority tasks are reduced since they complete earlier due to this reduction in execution times of medium priority tasks.

LOPF fares slightly better than LPF in our simulations. This is because in LPF, the preemption that is removed first need not be the last preemption in the timeline. Removal of such preemptions might result in automatic removal of preemptions occurring later in time without reducing execution times of task instances. This can result in low priority tasks completing later than as observed in LOPF.

5.6.3 Energy Consumption

The elimination of a preemption caused a 4.2 times increase in energy consumption when using LOPF. We found that for tasksets with high utilizations, the increase in energy was more prominent. Naturally, tasksets with a large number of tasks also showed a high increase in energy as this can be attributed to the high number of preemptions in these task sets. However, our proposed approach provides for trade-off between the number of preemptions and the energy consumption, as the user can selectively choose which preemptions are desirable to eliminate.

5.7 Discussion

So far, in our methodology we have not addressed two issues:

1. Speeding up tasks in the busy period before the start of the preempted task to eliminate the preemption.
2. Explicit removal of potential preemptions (although, as shown in the experiments, many of them are eliminated automatically when eliminating initial preemptions).

Consider a preemption where $\tau_{i,l}$ preempts $\tau_{j,k}$. It can be either a potential preemption or an initial preemption. In order to address both the cases described above, we must find the set Υ where,

$$\Upsilon = \tau_{j,k} \cup \{\sum_{p=1}^{n} \sum_{q=1}^{LCM/T_p} busy\_period(i,l,p,q) \times \tau_{p,q}\}$$

where, $busy\_period(i, m, p, q)$ returns 1 if $\tau_{p,q}$ is in the busy period just before $start_{i,l}$. Now we need to find $C_{a,b}$ for each $\tau_{a,b}$ in Υ such that $\forall \tau_{a,b} \in \Upsilon$:

$$\begin{aligned} finish_{a,b} &< (b-1) \times T_a + D_a, \; and \\ finish_{a,b} &< start_{i,m} \end{aligned}$$

Speeding up task executions in the busy period raises two issues:

1. One issue is finding the best execution times for each $\tau_{a,b}$ such that all of them finish before $start_{i,m}$ while meeting their individual deadlines. This is an optimization problem and has to be performed for each preemption elimination, as speeding up tasks may not be the best option to remove a preemption. It may also be that slowing down tasks in the busy period such that the preempted task starts after the preempting task can be a valid alternative. We plan to address this question in the future work by incorporating energy reduction and minimization of preemptions into the goal function of an optimization problem.

2. Eliminating potential preemptions by scaling up individual task execution times can result in a drastic increase in energy consumption. Hence, an attractive solution may be to remove potential preemptions at run time by slowing down tasks to ensure that the potential preemptions are not converted to actual preemptions. This approach again has the additional advantage of compensating the increased energy consumption due to the removal of preemptions by speeding up tasks.

5.8 Conclusions and Future Work

In this paper, we have proposed a methodology to reduce the number of preemptions in real-time scheduling by using CPU frequency scaling. We have provided an instantiation to FPS by analyzing a schedulable task set and calculating individual frequencies at which task instances need to execute such that the preemptions are eliminated, by taking into account the effect of preemption elimination on the rest of the schedule. The proposed approach does not imply modifications to the task attributes or the underlaying scheduler.

As the main element of cost introduced by our method is the energy consumption, the proposed framework provides for tradeoff between the number of preemptions and the cost by keeping track of the increase of energy required for each preemption elimination. Though runtime variations in the execution

time of task instances can introduce (or remove) additional preemptions, the offline method can be complemented with online approaches by enabling the use of efficient algorithms to remove/minimize preemptions.

Future work will focus on optimizing the approach such that the number of preemptions is minimized while minimizing the energy consumption, as well as online extensions to cope with execution variations between best and worst case, including cache related preemption delay cost. At the same time, the method will be extended to the sporadic task model.

5.9 Acknowledgment

The authors wishes to thank the anonymous reviewers for their useful comments on the paper. This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research center PROGRESS and the Erasmus Mundus External Co-operation Window programme EURECA.

Bibliography

[1] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1993.

[2] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1995.

[3] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 1998.

[4] Schneider J. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. *The 21st Real-Time Systems Symposium*, 2000.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *The Journal of ACM*, 1973.

[6] Harini Ramaprasad and Frank Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *The IEEE Real-Time Embedded Technology and Applications Symposium*, 2006.

[7] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *The Proceedings of the IEEE*, 1994.

[8] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. In *Real-Time Systems Journal*, January 2005.

[9] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *The 16th Euromicro Conference on Real-time Systems*, 2004.

[10] Gang Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[11] Yun Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Sixth International Conference on Real-Time Computing Systems and Applications, 1999. RTCSA '99.*, 1999.

[12] Manas Saksena and Yun Wang. Scalable multi-tasking using preemption thresholds. In *In Digest of Short Papers For Work In Progress Session, The 6th IEEE Real-Time Technology and Application Symposium*, 2000.

[13] Liu Yang, Man Lin, and Laurence T. Yang. Integrating preemption threshold to fixed priority dvs scheduling algorithms. In *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, 2009.

[14] Woonseok Kim, Jihong Kim, and Sang Lyul Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, New York, NY, USA, 2004. ACM.

[15] Giorgio Buttazzo. Research trends in real-time computing for embedded systems. *SIGBED Rev.*, pages 1–10, 2006.

[16] Hakan Aydin, Rami Melhem, Daniel Moss, and Pedro Meja-Alvarez. Power-aware scheduling for periodic real-time tasks. *The IEEE Transactions on Computers*, 2004.

[17] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe Lipari. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Transaction on Embedded Computer Systems*, 2009.

[18] Mauro Marinoni and Giorgio C. Buttazzo. Elastic dvs management in processors with discrete voltage/frequency modes. *IEEE Transactions on Industrial Informatics*, 2007.

[19] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *The 18th ACM symposium on Operating systems principles*, 2001.

[20] Saehwa Kim, Seongsoo Hong, and Tae-Hyung Kim. Integrating real-time synchronization schemes into preemption threshold scheduling. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[21] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 73–83, 2001.

[22] Ravindra Jejurikar and Rajesh K. Gupta. Integrating processor slowdown and preemption threshold scheduling for energy efficiency in real time embedded systems. In *The IEEE Real-Time Computing Systems and Applications*, 2004.

[23] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 2005.

Chapter 6

Paper B: Probabilistic Preemption Control using Frequency Scaling for Sporadic Real-time Tasks

Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat
In proceedings of the 7th International Symposium on Industrial Embedded Systems, IEEE, Karlsruhe, Germany, June, 2012

Abstract

Preemption related costs are major sources of unpredictability in the task execution times in a real-time system. We examine the possibility of using CPU frequency scaling to control the preemption behavior of real-time sporadic tasks scheduled using a preemptive Fixed Priority Scheduling (FPS) policy. Our combined offline-online method provides probabilistic preemption control guarantees by making use of the release time probabilities of the sporadic tasks. The offline phase derives the probability related deviation from the minimum inter-arrival time of tasks. The online algorithm uses this information to calculate appropriate CPU frequencies that guarantees non-preemptive task executions while preserving the overall system schedulability. The online algorithm has a linear complexity and does not lead to significant implementation overheads. Our evaluations demonstrate the effectiveness of the method as well as the possibility of energy-preemption trade offs. Even though we have considered FPS, our method can easily be extended to dynamic priority scheduling schemes.

6.1 Introduction

Predictable execution of real-time tasks is one of the major requirements to guarantee the temporal properties of safety- and mission critical real-time systems. These systems typically employ the preemptive fixed priority scheduling (FPS) policy, that, since the pioneering work of Liu and Leyland [1], has been widely used in industrial real-time applications mainly due to its simple runtime scheduling mechanisms and low overhead, as well as its ability to handle even task sets with incomplete attribute specifications. However, preemptions are one of the major causes for the unpredictability in, e.g., the execution times of real-time tasks in such systems, thus potentially jeopardizing the system schedulability. Preemptions incur additional costs e.g., cache related preemption delays and context switch overheads, which negatively impact the temporal behavior of the system. These costs are difficult to bound given that they vary with the point of preemption and even possibly with the state of the system at the time of preemption [2]. The preemption related costs are difficult to be accounted in the schedulability analysis, which is typically done offline. On the other hand, too pessimistic assumptions regarding the preemption related costs may lead to inefficient utilization of the resources.

Preemption reduction, besides reducing preemption related costs, can also be beneficial in systems with low power consumption requirements. Preemptions can increase the accesses to off-chip memory, thereby increasing the power consumption in the system. This is because, some of the cache lines are evicted during a preemption and when the preempted task resumes its execution, the evicted cache lines have to be restored, increasing the off-chip access. It has been shown that [3], an off-chip memory access is typically more expensive than an on-chip cache access in terms of energy consumption. Preemption reduction, hence, reduce these additional energy requirements that occurs due to an increased cache pollution. Preemptions can also potentially accelerate the wear and tear of the hardware that the real-time system is controlling. A non-preemptive FPS, on the other hand, may be an attractive alternative due to its lower runtime overhead. A major drawback in using non-preemptive scheduling, however, is that, due to the blocking of the higher priority tasks by the lower priority tasks, a portion of the processor time is typically *wasted*. This loss of utilization [4] cannot be bounded and hence non-preemptive scheduling can prove to be infeasible even for arbitrarily low utilizations. This loss of utilization makes non-preemptive scheduling unfavorable for most practical applications.

On the other hand, the need for energy efficient systems necessitates the use

of adequate energy management techniques. One of the methods adopted to reduce the energy consumption is to utilize the possibility of Dynamic Voltage and Frequency Scaling (DVS) to reduce the CPU frequency and voltage whenever possible, without jeopardizing the temporal guarantees of the real-time system. Reducing the CPU frequency reduces the performance and increases the execution times of the real-time tasks in the system, increasing the processor utilization. The increase in CPU utilization, in its turn, increases the number of preemptions in the system [5]. The ability to scale up/down the CPU frequency to change task execution times provides the designer the possibility of using energy manipulation to influence the execution behaviour of real-time tasks, in order to achieve specified requirements. Traditionally, DVS has been used in real-time systems mainly to conserve energy, while meeting the temporal requirements [6][7]. On the other hand, increasing the CPU frequency leads to shorter task execution times, and, implicitly, less preempting opportunities. We use the possibility of using frequency scaling to influence the tasks' execution times in order to control the number of preemptions.

In real-time systems, the physical events occurring in the system are mapped to a set of real-time tasks. The events are sampled at a minimum or an exact frequency that is sufficient to meet the physical requirements of the system. A sporadic task model is adopted to represent events where no two events can occur more frequently than a certain known frequency. In such systems, the task arrival rates are assumed at a maximum frequency to analyze its worst case behavior in a predictable manner. However, in many cases the probabilities of the event occurrences can be found, thus enabling the use of a probabilistic analysis. Probabilistic approaches reduce pessimism by considering the probabilistic distribution of the task attributes such as minimum inter-arrival times or WCETs [8].

In this paper, we present a method to control the preempting behavior in sporadic task systems with probabilistic release times, using CPU frequency scaling. The task release time probabilities are considered to find deviations in the inter-arrival times. This deviation is derived from the task release time probabilities such that the probability of a task release is greater than a known threshold e.g., the time instant where the probability of preemption is the highest. This information is used by an online algorithm to control the preemptions online. Considering probabilities while performing preemption control using CPU frequency scaling provides the designer the possibility of trade-off between preemption costs and energy-consumption. Simulation results clearly indicate that considering task release probabilities can provide energy preemption trade-offs. Our algorithm has a linear complexity and does not add any

significant runtime overhead.

The paper is organized as follows. In section 6.2, we discuss the related work. Section 6.3 details the system model and the various notations used throughout this paper. In Section 6.4, we present our methodology followed by an example in Section 6.5. We conclude in Section 6.7.

6.2 Related Work

Preemptions are widely known to increase costs in the system and the need for reducing the preemptions in a real-time system is widely recognized in the literature [9][10][11][2]. The main preemption related costs are composed of the direct costs to perform the context-switches [11], the costs to manipulate the task queues [10][11], as well as the generally unpredictable cost of cache-related preemption delays [2]. It has been observed by Bui et. al in [9] that cache related preemption delays can increase the task execution times by 33% as the overhead due to cache related preemption delays can be as high as $655\mu S$ for a single preemption.

Since the work of Liu and Layland [1], preemptive FPS has been widely studied and several extensions were proposed for different task models. Preemptive FPS has also found widespread acceptance in the industry and is used in a large number of applications, mainly due to its flexibility and simple runtime overhead. However in reality, due to the overheads involved during a preemption, the use of preemptive FPS might not be ideal [10][12][2]. Buttazzo [5], for example, showed that the rate monotonic algorithm (RM), a widely used preemptive FPS technique, introduces a higher number of preemptions than earliest deadline first algorithm (EDF). He also observed that the number of preemption constantly increases with the task utilization.

Due to the widespread recognition for the need for preemption reduction, several methods have been proposed to reduce the number of preemptions in real-time scheduling. Preemption Threshold Scheduling (PTS) for FPS, first introduced in the ThreadX operating system [13], later formalized by Wang and Saksena [14], improves schedulability and reduces the number of preemptions and the number of threads in the system. The main disadvantage of this method is the need for a dual priority system which may not be directly suitable for, e.g., legacy systems, where scheduler modifications may not be possible. Baruah [4] studied the feasibility of limited preemption techniques and calculated the length of the longest possible non-preemptive execution of a task in a sporadic task system. Yao et. al. [15], evaluated and compared the vari-

ous limited preemption methods using experiments. Earlier they had extended [4] to FPS, finding an upper bound on the length of the largest possible non-preemptive execution of tasks under FPS [16]. Bertogna et. al. [17], presented a method to place preemption points within task code assuming a fixed preemption overhead.

DVS techniques, traditionally, were used for reducing energy consumption by slowing down tasks' executions [6][7]. It was observed [6][7] that the energy consumption increases linearly with frequency and quadratically with the applied voltage. One of the disadvantages of using DVS is the increase in the number of preemptions due to an increase in task execution times. In [18], the authors observed a less than $140\mu S$ of time for a frequency switch. Another work by Lu et. al. [19] reported $2\mu S$ on an Intel StrongARM processor for the same. This cost is not significant compared to the overhead incurred due to preemptions, making CPU frequency scaling a promising approach towards controlling preemption behavior in real-time systems. Also, since this is a technology dependent cost, with the advances in technology, this overhead is expected to come down.

Dobrin and Fohler [20], proposed a method to minimize the number of preemptions by re-assigning task attributes, such as priorities, periods and offsets, without affecting the schedulability of the taskset. Later in [21], we proposed an offline method to control the preemption behavior of periodic real-time tasks scheduled by FPS using CPU frequency scaling, by finding job level frequencies that guarantee preemption control. Later in [22] and [23], we extended this to the sporadic task model. However, the algorithm presented was a minimal algorithm and it did not make use of the task release probabilities, while determining the earliest point of preemption. In [22], for instance, only the preempting job was speed-ed up to avoid the preemption before the preemption point determined by the minimum inter-arrival time of the higher priority tasks. In [23], this was extended to speed-up the busy period before the preemption to gain more slack and thus require a lower speed-up. In this paper, we extend our previous works [22] and [23] to propose a method to control the preemption behavior of a sporadic task system scheduled by FPS by making use of task release probabilities to obtain better energy savings. We use the probabilities of task releases in order to derive probabilistic guarantees on the preemption behavior of the schedule while saving on the energy consumption. The use of probabilities to determine the earliest probable preemption point is shown to provide energy preemption trade offs.

6.3 System Model

In this section, we describe the system model and the notations used in this paper.

6.3.1 Processor Model

We assume a processor model that supports a set of discrete operating modes denoted by $M = \{m_1, m_2, m_3, ...m_p\}$, where each m_q is characterized by $m_q = (F_q, W_q)$. Each F_q denotes the processor frequency associated with mode m_q, and W_q is the set $W_q = \{w_q^1, w_q^2, .., w_q^r\}$, that represents the power consumption per clock cycle by the r resources used by the tasks in mode m_q. We assume that a known upper-bounded frequency-switch overhead exists. F_{max} and F_{min} respectively represents the maximum and minimum frequency supported by the processor. The task set is initially assumed to be executing at a default processor frequency $X \geq F_{sched}$, where F_{sched} denotes the minimum frequency that guarantees the system schedulability.

6.3.2 Task model

We consider a set of sporadic tasks [24] [25] denoted by $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$, where each τ_i has a minimum inter-arrival time T_i, a worst case execution requirement C_i, a unique priority P_i and a deadline D_i, relative to its release. Moreover, we assume that C_i, which is given by the largest number of clock cycles required for the execution of task τ_i, is independent of the clock frequency and is a constant [26]. Additionally, let $hp(i)$ represent the set of tasks with higher priorities than τ_i i.e., $P_j > P_i$, $\forall j \in hp(i)$ and LCM represents the Least Common Multiple of the minimum inter-arrival times of all the tasks in the taskset. We define the outstanding computations at a time instant *t* as the set of remaining clock cycles required to complete the execution of all tasks in the ready queue. The ready queue is denoted as $readyQ$. We define $C_j^r(t)$ as the time required to complete the outstanding computations of τ_j at any time t, at the default frequency, X. Moreover, we define $rel_j(t)$ as the earliest release time of the next job of τ_j at the time instant t and is easily obtained by adding T_i to the release time of its latest job that has been released before time *t*.

In addition to the above task parameters, we associate a probability mass function $f_i(t), t \in \mathbb{Z}$ with every task τ_i. $f_i(t)$ gives the probability that a job of the task τ_i is actually released at time t, relative to the its earliest release time.

Thus, the probability that a job of τ_i is released at a time $rel_j(t) + t_1$ is given by $f_i(t_1)$[8].

6.3.3 Energy Model

We represent the total energy consumption required by all task executions until time t by:

$$E_t = \sum_{i=1}^{n} \sum_{l=1}^{k} e_{i,l} \tag{6.1}$$

where k is given by the smallest integer satisfying:

$$(k+1)T_i + \sum_{d=1}^{k} \phi_{i,d} \geq t$$

and e by:

$$e_{i,l} = \sum_{b=1}^{C_i} \left\{ \sum_{a=1}^{r} w_q^a \right\}$$

In the above equation, m_q is the execution mode of the processor during the b^{th} clock cycle of $\tau_{i,l}$ and $\phi_{i,d}$ is the offset in the release of the job $\tau_{i,d}$ since the end of the inter-arrival time of $\tau_{i,d-1}$. Hence, $e_{i,l}$ is the sum of the actual power consumption of all the clock cycles, which gives the total energy used for the execution of the job $\tau_{i,l}$.

6.3.4 Execution Time Model

We assume a linear relationship between frequency and execution time of a job i.e., the execution time of a job $\tau_{i,j}$, denoted by $C_{i,j}$, is inversely proportional to the processor frequency. Note that C_i represents the *execution requirement* of τ_i and $C_{i,j}$ represents the *execution time* of its job $\tau_{i,j}$ at the default processor frequency, X. Consequently, the frequency required for scaling $C_{i,j}$ to $C'_{i,j}$ is given by the equation [21]:

$$X' = \frac{C_{i,j}}{C'_{i,j}} \times X \tag{6.2}$$

The above equation gives the maximum frequency that guarantees a required worst case execution time for a particular job. Also note that, if $C_{i,j}$ represents the initial execution time of $\tau_{i,j}$, $\frac{C_{i,j}}{C'_{i,j}}$ also denotes the speed at which the processor must execute to complete $\tau_{i,j}$ in $C'_{i,j}$ time units.

6.4 Methodology

Our solution consists of a joined offline - online approach: 1) an offline phase which calculates the deviation between the most probable inter-arrival time and the minimum inter-arrival time and 2) an online algorithm that determines the earliest possible preemption point for each task, using the values derived in the offline analysis, and calculates the optimum CPU frequencies that guarantees the preemption avoidance.

In the offline phase, the task release probabilities are considered to calculate the deviation between the most probable inter-arrival time and the minimum inter-arrival time such that the probability that a higher priority task is released does not exceed a certain pre-determined threshold. Later, in the online phase, while calculating a particular earliest preemption point we obtain a probabilistic preemption guarantee, which is less pessimistic than the probability of preemption occurrence based on the minimum inter-arrival time of the preempting task. By less pessimistic we mean that the preemption point determined by our algorithm will be farther in time than in the case when the actual inter-arrival times of the preempting tasks are considered. This is beneficial from an energy usage point of view, since the speed up required to avoid the preemption is less than the case when the actual minimum inter-arrival times are considered. In the following, we discuss the details of our method.

6.4.1 Offline Phase

Every sporadic task τ_i has a minimum inter-arrival time T_i and a task release probability $f_i(t)$ from the point of its earliest release time. For instance, if a job of τ_i was released at time t_k, the earliest release time of the next job is $t_k + T_i$ and $f_i(t)$ gives the probability of its release at time $t_k + T_i + t$, $t \in \mathbb{Z}$. Consequently, the next job of τ_i will be released at at a time

$$t_k + T_i + t_{k+1} \; s.t. \; f_i(t_{k+1}) = l_i$$

where l_i represents the threshold probability for release of a job of τ_i.

Thus the deviation between the most probable inter-arrival time and the minimum inter-arrival time can be calculated using the task release probabilities. Let R_i be the deviation from the minimum inter-arrival time to the most probable inter-arrival time of the task τ_i i.e., $f_i(R_i) = l_i$. At any time instant t, the next job of a task τ_i will be released at

$$rel_i(t) + R_i$$

The most probable time instant for a task release at a time t is:

$$rel_i(t) + R_i \; s.t. \; f_i(R_i) = max(f_i(t)) \;\; \forall \;\; t \in \mathbb{Z}$$

6.4.2 Online Preemption Control Algorithm

In a sporadic task system, since the inter-arrival times of the tasks are bounded by a lower bound, it is impossible to know the time at which a job of the task will be *actually* released. Consequently, preemptions on lower priority tasks cannot be predicted because of the indeterminism in the higher priority task releases after their minimum inter-arrival times. However, the minimum time interval during which the next job of a particular task will *not* be released can be determined during runtime. Hence, for a lower priority task, it is possible to find the maximum time for which a higher priority task will not be released i.e., it gives the maximum time for which the task can be guaranteed a non-preemption. This gives the maximum time within which the set of outstanding computations must be executed, in order to *guarantee* its non-preemptiveness considering the minimum time interval during which a higher priority job will not be released.

In our task model, every job of τ_i is released with a probability $f_i(t)$, t time units after its earliest release time. It is quite evident that higher the value of $f_i(t)$, higher the probability that the task is released. We use these task release probabilities to provide a probabilistic guarantee for removing the preemption by executing the outstanding computations before the probable point in time at which higher priority task is released.

In our previous work [23], we used the earliest release time of a job as the earliest possible preemption point. However, when considering the task release probabilities, we can relax the above assumption by exploiting the probabilities of the higher priority task releases in future, thereby deriving probabilistic preemption points. We can make use of these probabilistic preemption points to derive processor speeds such that we are able to provide probabilistic guarantees on the preemption behavior of the schedule. Algorithm 1, finds the lowest priority task (τ_u) in the ready queue whenever a job of a task τ_i starts its execution. It finds the earliest time in the future at which a job having a priority higher than τ_u can be possibly released with a known threshold probability. This gives the earliest time at which at least the lowest priority job from among the jobs in the ready queue can be preempted by the higher priority task with a probability equal to the threshold. It then computes the minimum frequency at which the processor must execute the outstanding computations to avoid a

preemption at this point.

Whenever a job starts its execution, if τ_u is the lowest priority task in the ready queue, the earliest release time of a job with a higher priority than τ_u is given by:

$$t_{hp\_rel} = \min_{\forall \tau_i \in hp(u)} (rel_i(t) + R_i)$$

The maximum time for which the outstanding computations can execute non-preemptively relative to a time instant t, is given by:

$$t_{available} = t_{hp\_rel} - t$$

Hence, in order to guarantee the non-preemptive execution of the outstanding computations at any time t, its execution time should be no greater than $t_{available}$.

Algorithm 1 The algorithm is executed at the start time of a job in order to control the number of frequency switches required and to also leverage on the potential gains due to tasks executing for less than their WCET. For instance, we consider a job of τ_i starting its execution at a time instant t. Let the lowest priority task in the ready queue at time t be τ_u. The outstanding computations must execute no greater than $t_{available}$ units of time in order to finish execution before the preemption, where,

$$t_{available} = t_{hp\_rel} - t$$

The outstanding computations require t_{out} time units to execute at the default frequency X, where:

$$t_{out} = \sum_{\forall \tau_a \in readyQ} C_a^r(t)$$

If $t_{available} \leq 0$, it means that the time instant when the release probability of a higher priority task is equal to its threshold probability has elapsed, and its job was not released. Here we could use the same reasoning by using a secondary threshold. However, to preserve the simplicity of the method, we execute the processor at the maximum speed so that the low priority computations complete as early as possible.

If $t_{available} > 0$, we have three cases,

1. $t_{available} < t_{out}$
2. $t_{available} = t_{out}$

3. $t_{available} > t_{out}$

Consider case 1, $t_{available} < t_{out}$, i.e., the time required to execute the outstanding computations at time t is greater than the minimum time to the next preemption. In this case, if the outstanding computations finish their executions in $t_{available}$ time units, their non-preemptive execution can be guaranteed. Hence, the new frequency X', that guarantees their non-preemptive execution is given by:

$$X' = \frac{t_{out}}{t_{available}} \times X$$

If the calculated frequency is higher than F_{max}, i.e., the preemption avoidance

Algorithm 1: Find the minimum processor frequency at time t for the non-preemptive execution of the outstanding computations.

τ_u : *the lowest priority task active in readyQ*
$t_{hp\_rel} \leftarrow 9999999 (a\ large\ value)$
$i \leftarrow 1$
while $P_i > P_u$ **do**
  if $t_{hp\_rel} > rel_i(t) + R_i$ **then**
    $t_{hp\_rel} \leftarrow rel_i(t) + R_i$
  end if
  $i \leftarrow i + 1$
end while
$t_{available} \leftarrow t_{hp\_rel} - t$
if $t_{available} > 0$ **then**
  $t_{out} = \sum_{\forall \tau_a \in readyQ} C_a^r(t)$
  $X' \leftarrow \frac{t_{out}}{t_{available}} \times X$
  if $X' > F_{max}$ **then**
    $X' \leftarrow F_{max}$
  end if
  if $X' < F_{sched}$ **then**
    $X' \leftarrow F_{sched}$
  end if
else
  $X' \leftarrow F_{max}$
end if

cannot be guaranteed due to hardware limitations, the processor frequency is

set to F_{max}. If in such a scenario, more complex algorithms are used online to calculate the probabilistic preemption points on the outstanding computations, the associated overheads increase. For example, we could use a secondary threshold probability to determine the next possible preemption point. However, to keep the method simple, we set the processor frequency to F_{max}.

In case 2, the above equation becomes $X' = X$, i.e., the processor executes at the default frequency. In case 3, i.e., $t_{available} > t_{out}$, there is a possibility to slow down the processor to conserve energy. The equation to find the new frequency is valid for this case as well. It will find a lower frequency (thus a lower voltage) that guarantees a non-preemptive execution of the outstanding computations. If the calculated frequency is lower than F_{sched}, the processor executes at F_{sched}, thus preserving the overall schedulability of the task set. Even though we have presented our method in the context of FPS, our methodology can be easily extended to dynamic priority scheduling e.g., the EDF scheduling. This can be achieved by considering the task instance priorities rather than the task priorities in the algorithm.

Computational Complexity : The algorithm 1 has a linear complexity assuming that the existence of the jobs in the ready queue is kept track of by, e.g., a simple associative array. The number of jobs in the ready queue at any time t cannot exceed the total number of tasks n. The lowest priority job is the last task in the ready queue, and finding it does not add any significant complexity to the approach. The earliest possible preemption point for the outstanding computations can be found by a simple search in $rel_i(t), \forall i \in hp(u)$. This can also be done in a time linear in the number of tasks as $rel_i(t)$ contains a maximum of n release times. Also, finding the outstanding computations can also be done in linear time because the number of jobs in the ready queue cannot exceed n.

Implementation Considerations : The online preemption control algorithm can be easily implemented using techniques similar to the DVS algorithms. The implementation typically should occur at the operating system level, where the scheduler is modified to calculate frequencies that can enable preemption control. Whenever a new task arrives, the total outstanding computations can be updated without significant overhead, by just adding the computation requirement of the new task to the current total outstanding computations. The lowest priority task in the ready queue is the last task in the queue, thus a search through the queue can be avoided. In order to find the earliest possible preemption point on the outstanding computations, the OS has to maintain a data structure which stores the next earliest release times of each sporadic task. This is calculated by adding the inter-arrival time of the task to

Task	C_i	T_i
X	1	5
Y	3	10
Z	3	20

Table 6.1: Example taskset

its latest release time. The earliest possible preemption point can be found by a simple search through this data structure.

6.5 Example

We illustrate our proposed method with an example. Consider a set of sporadic tasks with execution requirements C_i and minimum inter-arrival times T_i, as given in table 6.1. Let the time required to execute C_i computations of each task be C_i time units at speed 1. Let the probability of task releases from their respective earliest release times be given by the probability mass function in Figure 6.1. Note that the probabilities can be different for different tasks. In this example, for the purpose of simplicity we assume the same probability mass function for all the tasks. Due to the sporadic nature of task releases, one possible runtime scenario for the task executions is shown in Figure 6.2 where there are 2 preemptions when the tasks execute for their WCET's, are scheduled using FPS and the priorities are assigned according to the rate monotonic priority ordering. Let the i^{th} job of task X be represented by X_i, that of Y be represented by Y_i and of Z by Z_i.

In the offline analysis, we assume that the threshold probability is 0.20. The corresponding deviation from the minimum inter-arrival time that gives a release probability equal to the specified threshold probability is 1 time unit. Thus, the most probable time instant at which a job can be released is 1 time unit after the its earliest release time.

When X_1 starts its execution at time $t_1 = 0$, the lowest priority job in the ready queue is Z_1. The next possible release time of a job having a higher priority than Z_1 is by X_2 at time $t_2 = 5$. We add the deviation to the inter-arrival times based on the probability mass function in figure 6.1. Thus the next probable higher priority task release will happen at time $t_2' = 5 + 1 = 6$. Thus the processor has $t_2' - t_1 = 6$ time units available to execute the outstanding computations non-preemptively. The outstanding computations take $1+3+3 =$

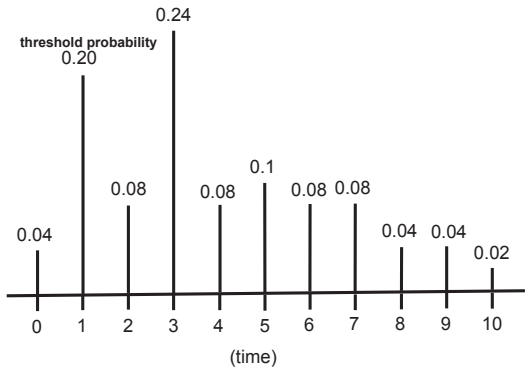


Figure 6.1: Example probability mass function

7 time units to execute. The processor has to be speed-ed up by a factor of $\frac{7}{6}$ such that the outstanding computations execute non-preemptively. A part of the sporadic task schedule implementing algorithm 1 is shown in Figure 6.3. When Y_2 starts its execution at time 12, the earliest preemption point has already elapsed (at time instant 11). After adding the deviation of 1 time unit, we can see that the probable release time of X_3 has already elapsed. Here we could use a secondary threshold value to determine the earliest preemption point on Y_2 by a job of X. However for the purpose of simplicity, we speed up the processor such that Y_2 completes its execution as early as possible.

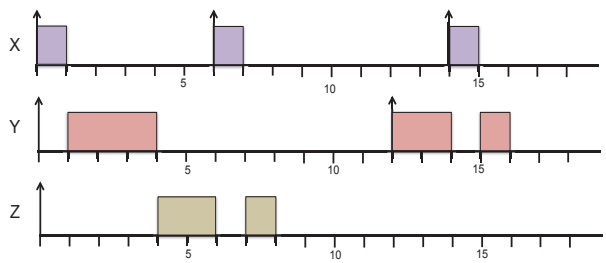


Figure 6.2: A part of the original FPS schedule of the sporadic task set

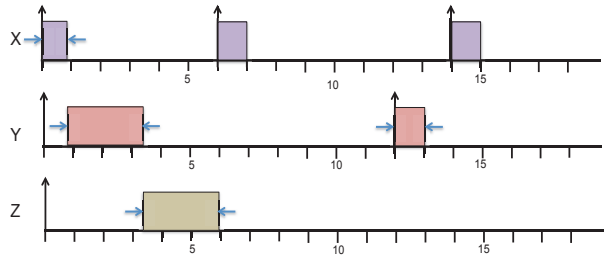


Figure 6.3: The sporadic task schedule after preemption control

Processor speed	0	1	2	3	4	5
Power consumption per clock cycle (mW)	0	20	50	50	200	500

Table 6.2: Processor Model

6.6 Evaluation

We evaluated our method on synthetic tasks by generating 1400 task sets, having 3 - 15 tasks per task set with LCM ≤ 2000, using the UUniFast [27] algorithm. The processor model that we used in our evaluations, which we adapted from [28], is given in table 6.2. The task sets were generated such that they are schedulable at speed=1. In our experiments, for each task τ_i in Γ, we generated $\frac{LCM}{T_i}$ number of instances where every task instance was released after a time t, with a probability as given in Figure 6.1. We assumed threshold probabilities of 0.20 and 0.24, to bound the number of preemptions. Consequently, the deviations determined in the offline phase are 1 and 3 time units. Each simulation was run until all the $\frac{LCM}{T_i}$ jobs of every τ_i were executed. We calculated the average number of preemptions that occurred for the following cases i) normal FPS ii) algorithm 1 with $R_i = 0$ iii) algorithm 1 with $R_i = 1$ and iv) algorithm 1 with $R_i = 3$. We present our results in Figures 6.4 and 6.5. Figure 6.4 shows the average number of preemptions for various task utilizations under the different cases described previously and figure 6.5 shows the average power consumption under the different cases.

Our method showed significant reduction in the number of preemptions as seen from the evaluation results presented in Figure 6.4. It is seen that con-

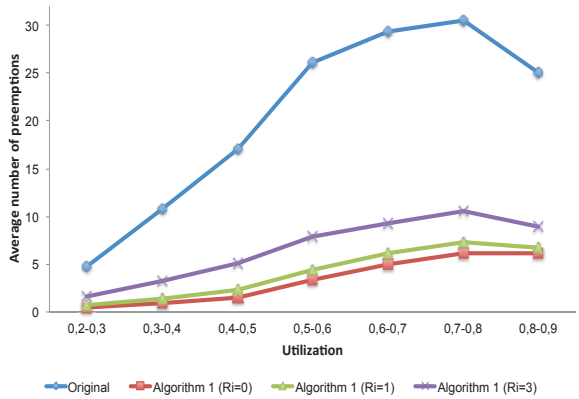


Figure 6.4: Average number of preemptions for various threshold probabilities

sidering probabilities while calculating the CPU frequency achieves almost an equal reduction in the number of preemptions as for the case where probabilities are not considered (i.e., $R_i = 0$) while saving energy in the system. The number of preemptions for the highest utilization range (0.8-0.9) shows a decrease for $R_i = 0$ because these task sets where found to have less number of tasks causing less preemptions. The reduction in the energy consumption achieved is particularly significant for tasks with higher utilizations where the number of preemptions is typically higher.

The task release probabilities can be used to achieve preemption-energy trade-offs in the system as can be seen from the two figures. By varying the relaxations permitted to the probable earliest release times, we observe that the preemption reduction achieved varied and so did the energy consumption, demonstrating the possibility of energy preemption trade-off.

6.7 Conclusions

In this paper we presented a combined offline-online approach to control the preemptive behaviour of sporadic task systems with probabilistic inter-arrival times by using CPU frequency scaling. While an offline analysis derives the probability related deviation from the minimum inter-arrival time, an online algorithm uses this information to provide appropriate CPU frequencies that

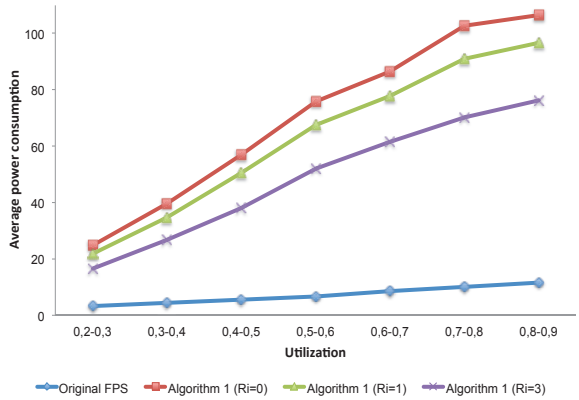


Figure 6.5: Average power consumption for various threshold probabilities

guarantees the non-preemptive task executions while preserving the overall system schedulability. We do so by finding the earliest time instant at which at least one of the jobs in the busy period can be preempted with a probability above a certain known threshold, whenever a task starts its execution. We then calculate the processor frequency such that the jobs in the busy period finishes execution before this point so that a preemption is avoided. The online algorithm has a linear complexity and does not lead to significant implementation overheads. Evaluation results show the effectiveness of our method in reducing the number of preemptions in the schedule, as well as it also demonstrates the methods' ability to provide for trade-offs between the number of preemptions and overall energy consumption.

Ongoing efforts focus on deriving upper bounds on the speed-up required for guaranteeing a specified preemption behaviour and extensions to the multiprocessor platform.

Bibliography

[1] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *The Journal of ACM*, 1973.

[2] H Ramaprasad and F Mueller. Tightening the bounds on feasible preemptions. In *The ACM Transactions on Embedded Computing Systems*, 2008.

[3] Michael Zhang and Krste Asanovic. Highly-associative caches for low-power processors. In *In Kool Chips Workshop, Micro-33*, 2000.

[4] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *The 17th Euromicro Conference on Real-Time Systems*, 2005.

[5] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. In *Real-Time Systems Journal*, January 2005.

[6] Hakan Aydin, Rami Melhem, Daniel Moss, and Pedro Meja-Alvarez. Power-aware scheduling for periodic real-time tasks. *The IEEE Transactions on Computers*, 2004.

[7] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *The 18th ACM symposium on Operating systems principles*, 2001.

[8] Liliana Cucu and Eduardo Tovar. A framework for the response time analysis of fixed-priority tasks with stochastic inter-arrival times. *SIGBED Rev.*, 2006.

[9] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In

The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008.

[10] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1995.

[11] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1993.

[12] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *The Proceedings of the IEEE*, 1994.

[13] William Lamie. Preemption threshold. *Whitepaper*, 1997.

[14] Yun Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Sixth International Conference on Real-Time Computing Systems and Applications, 1999. RTCSA '99.*, 1999.

[15] Gang Yao, G. Buttazzo, and M. Bertogna. Comparitive evaluation of limited preemptive methods. In *The 15th International Conference on Emerging Technologies and Factory Automation*, 2010.

[16] Gang Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[17] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *The 22nd Euromicro Conference on Real-Time Systems*, 2010.

[18] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *The 7th annual international conference on Mobile computing and networking*, 2001.

[19] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Dynamic frequency scaling with buffer insertion for mixed workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2002.

[20] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *The 16th Euromicro Conference on Real-time Systems*, 2004.

[21] Abhilash Thekkilakattil, Anju S Pillai, Radu Dobrin, and Sasikumar Punnekkat. Reducing the number of preemptions in real-time systems scheduling by CPU frequency scaling. In *The 18th International Conference on Real-Time and Network Systems*, 2010.

[22] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Preemption control using CPU frequency scaling in real-time systems. In *The 18th International Conference on Control Systsems and Computer Science*, 2011.

[23] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Towards preemption control using CPU frequency scaling in sporadic task systems. In *Proceedings of the WiP of The 6th International Symposium on Industrial Embedded Systems*, 2011.

[24] Aloysius Ka Lau Mok. Fundamental design problems of distributed systems for the hard-real-time environment. *Massachusetts Institute of Technology, PhD thesis*, 1983.

[25] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *The 11th Real-Time Systems Symposium*, 1990.

[26] Rami Melhem, Daniel Mosse, and Elmootazbellah (Mootaz) Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Transactions on Computers*, 2004.

[27] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 2005.

[28] Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing cpu energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computer Systems*, 2009.

Chapter 7

Paper C: Quantifying the Sub-Optimality of Non-Preemptive Real-time Scheduling

Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat
Technical Report, Mälardalen Real Time Research Centre, Mälardalen University, Västerås, Sweden, November, 2012

Abstract

Many preemptive real-time scheduling algorithms, such as the Earliest Deadline First (EDF), are known to be optimal on a uni-processor. However, no such algorithms exist under the non-idling non-preemptive scheduling paradigm. Hence preemptive schemes strictly dominate non-preemptive schemes with respect to feasibility. However, the 'goodness' of non-preemptive schemes in successfully scheduling feasible task sets when compared to uni-processor optimal preemptive scheduling schemes such as the EDF, which can also be referred to as its sub-optimality, is unknown. In this paper, we apply resource augmentation, specifically processor speed-up, to quantify the sub-optimality of non-preemptive scheduling with respect to an optimal uni-processor scheduling scheme such as the EDF. We also present a method to guarantee user specified upper-bounds on the preemption related costs in the schedule.

We prove that the speed-up required to guarantee the feasibility of a non-preemptive execution of any task τ_i, for a duration of L_i, is upper-bounded by $\frac{4L_i}{D_{min}}$, where D_{min} is the smallest relative deadline in the task set. Consequently, we show that the upper-bound on the processor speed that guarantees the feasibility of a non-preemptive schedule for the task set is $\frac{4C_{max}}{D_{min}}$, where C_{max} is the largest execution time in the task set. The derived upper-bound is used in a sensitivity analysis based method to calculate the optimal processor speed that guarantees a specified upper-bound on the preemption related costs in the schedule. For this, we first present a method to translate the system-level requirements of meeting specified upper bounds on the preemption related costs to task level non-preemption requirements. We then use sensitivity analysis technique to calculate the optimal processor speed that guarantees the feasibility of the derived task level non-preemption requirements, which in its turn guarantees the desired bounds on the preemption related overheads.

Our contribution quantifies the sub-optimality of non-preemptive scheduling in terms of the processor speed-up required to successfully schedule all the uni-processor feasible task sets. It also enables a system designer to use a faster processor to guarantee specified upper-bounds on the preemption related overheads.

7.1 Introduction

Real-time scheduling theory has matured to the point where most of the fundamental questions regarding preemptive uni-processor scheduling have been fairly answered and many of the results have been passed on to the industry. A major part of this research deals with schedulability analysis [1] [2] [3] and feasibility analysis [4] [5] [6] [7] of hard real-time periodic and sporadic tasks under a preemptive scheduling scheme. However, the feasibility of non-preemptive scheduling has received relatively much less attention [8] and non-preemptive scheduling can be considered as less understood when compared to preemptive scheduling [9]. While there exists utilization based tests for schedulability and feasibility of preemptive real-time tasks under various assumptions [1], no such tests exists for non-preemptive scheduling even under restrictive assumptions. The Earliest Deadline First (EDF) is known to be optimal [4], while Fixed Priority Scheduling (FPS) scheme is not optimal with respect to uni-processor scheduling. Consequently, many works focused on finding the sub-optimality of FPS with respect to an optimal scheduling scheme such as the EDF under preemptive and non-preemptive scheduling [10] using resource augmentation. However, no such attempts have been made to quantify the sub-optimality of non-preemptive scheduling with respect to preemptive scheduling even though it is known that preemptive real-time scheduling strictly dominates non-preemptive scheduling [7]. Investigating the sub-optimality of non-preemptive scheduling with respect to a preemptive scheduling may also provide significant insights into the development of a utilization based test for non-preemptive scheduling.

A major factor which influences task set feasibility in modern real-time systems, where hardware features such as caches influences the task executions, is the number of preemptions and the points at which these preemptions occur. A detailed discussion on the preemption related costs is given in the related works section. The feasibility analysis of real-time tasks [6] [5] assumes zero preemption related overheads. Whenever these preemption related overheads are not negligible, they are assumed to be accounted for in the worst case execution time (WCET) of the tasks, leading to significant pessimism in the resulting analysis. Hence, the task sets which were originally feasible, without considering the preemption related overheads, may not be feasible when the worst case preemption related overheads are added to the WCET. As a consequence of this assumption, the set of feasible task sets identified by the feasibility analysis, may not include those task sets which might be feasible by restricting preemptions. In order to account for such task sets, Baruah and Bertogna in-

troduced the limited preemption technique called the floating non-preemptive region (f-NPR) [7] [11] scheduling, which disables preemptions per task for a bounded time period, without compromising task feasibility. However, the limited preemption scheduling scheme may still identify some task sets as infeasible e.g., due to increased preemption costs resulting from the number of preemptions being greater than a known threshold. This is because the largest non-preemptive regions are not 'large' enough to provide any specified non-preemption guarantees e.g., in this case guarantee that the number of preemptions are less than the known threshold. It is known that the task WCET scales with processor frequency. Consequently, there exists a possibility to speed-up the processor to control the length of the largest non-preemptive region, and guarantee its feasibility by ensuring that the largest non-preemptive regions are 'large' enough to achieve the specified non-preemption behavior. Augmenting the scheduler with a faster processor, can guarantee the feasibility of a specified non-preemption behavior that minimizes the preemption related overheads, and hence broaden the set of feasible task sets.

Resource augmentation was first introduced by Kalyanasundaram et. al. [12], showing that faster processors can achieve the same effect as clairvoyance. We [13] derived the upper-bound on the minimum processor speed-up required, that can guarantee the fault tolerance feasibility of a set of real-time tasks under an error burst of known length. They showed that if the error burst length is no larger than half the shortest deadline, the processor speed-up required that guarantees the fault tolerance feasibility is upper-bounded by 6. Davis et. al. [10] derived the upper and lower bounds on the processor speed-up required for a fixed priority scheduler to schedule all the task sets scheduled by an optimal scheduling algorithm, leveraging on the optimality of the EDF. In this work, the 'goodness' or sub-optimality of FPS with respect to an optimal scheduling algorithm such as the EDF, is quantified by the processor speed-up required to guarantee the FPS schedulability of the set of all feasible tasks that are schedulable by the optimal algorithm.

In this paper, our aim is to quantify the sub-optimality of non-preemptive scheduling with respect to uni-processor optimal preemptive schemes, such as EDF, using resource augmentation. We derive the resource augmentation bound, specifically the upper-bound on the processor speed-up, that guarantees the feasibility of a *user specified non-preemption behavior* of the real-time tasks. Using this, we derive the upper-bound on the processor speed-up that guarantees the feasibility of a *fully non-preemptive schedule*, which allows us to quantify the sub-optimality of non-preemptive scheduling. Processor speed-up can be used to achieve effective preemption related cost control by guaran-

teeing a non-preemption behavior that bounds the preemption related costs by a user specified value. To achieve this, we first present a method to translate the preemption related cost control requirements to a set of non-preemption requirements on the tasks e.g., the length of the non-preemptive region that guarantees no more than a user specified number of preemptions per task. We then use sensitivity analysis using the derived upper-bound to calculate the optimal processor speed-up factor that guarantees the feasibility of the derived non-preemption requirements, consequently guaranteeing the specified bounds on the preemption related costs. Our proposed approach provides a real-time system designer with the possibility of controlling the length of the non-preemptive regions in order to achieve finer control on the preemption related costs, besides quantifying the sub-optimality of non-preemptive real-time scheduling.

The paper is organized as follows: section 7.2 details the system model and the various notations used throughout this paper. We recall and reinterpret some key results about feasibility of preemptive and non-preemptive real-time scheduling in section 7.3. We present our main contribution, the resource augmentation bound for non-preemptive real-time scheduling, in section 7.4. We then build on the theory presented in section 7.4 to derive a methodology for minimizing preemption related costs by using processor speed-up to guarantee a specified preemption behavior in section 7.5. The two steps of the method are presented in sections 7.5.1 and 7.5.2, followed by an example in section 7.5.3. In section 7.6, we present a discussion on relaxing the assumption made previously. Finally, we present our conclusions in section 7.8 after giving an overview of some related works in section 7.7.

7.2 System Model

In this section, we introduce the notations used in this paper whilst describing the task model, scheduling model and the execution time model.

7.2.1 Task model

We consider a set of sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$, where each τ_i has a minimum inter-arrival time, T_i, a worst case execution time, C_i^S at processor speed S, and a relative deadline, D_i. We assume that the tasks are ordered according to the increasing order of their deadlines, which means that $D_{min} = D_1$. Let the length of the longest critical section in τ_i be denoted

by CS_i^S at speed S. We assume that every task τ_i has m_i optimal preemption points within its execution, where the m_i^{th} point denotes the end of the task execution. Let $q_{i,j}^S$, $j = 1...m_i$ denote the length of the task execution of τ_i up to its j^{th} optimal preemption point on a processor at speed S, from the start of the task execution. We assume negligible preemption related overheads at these optimal preemption points for the sake of clarity of presentation, the relaxation of which is discussed in section 7.6.

Let β_i^S denote the blocking tolerance of τ_i, which is the largest time for which τ_i can be blocked without causing a deadline miss. Also, let $B_i^S \leq \beta_i^S$ denote the largest time for which τ_i is actually blocked. LCM denotes the Least Common Multiple of the time periods of all the tasks in the set. The utilization U_i of a task τ_i executing on a processor at speed S is defined as $U_i^S = \frac{C_i^S}{T_i}$ and the utilization of the entire task set is given by $U^S = \sum_{i=1}^{n} U_i^S$. The demand bound function of a task τ_i, on a processor of speed S, during a time interval $[0, t]$ is given by [5],

$$DBF_i^S(t) = max\left(0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i^S$$

For example, $DBF_i^1(t)$ denote the cumulative processor time requested by τ_i during a time interval $[0, t]$ on a processor of speed $S = 1$.

7.2.2 Scheduling Model

In our scheduling model, we assume that, whenever a higher priority task is released during the execution of a lower priority task τ_i, instead of immediately preempting τ_i, the scheduler blocks the higher priority task for Q_i^S time units on a processor of speed S. This Q_i^S is the largest length of the non-preemptive region of τ_i derived from the task attributes [7] [11]. This type of scheduler is referred to as an f-NPR scheduler [14] (f-NPR for Floating Non-Preemptive Regions). Consequently, the maximum number of times the task τ_i can be preempted, when a processor of speed S is used, is given by, $\left\lceil \frac{C_i^S}{Q_i^S} \right\rceil - 1$.

We assume a work conserving scheduler, i.e., the scheduler does not idle the processor when there are active tasks awaiting the processor. It is known that the Earliest Deadline First scheduling (EDF) is optimal under a work conserving uniprocessor scheduling scheme under both preemptive and non-preemptive paradigms [4]. We leverage on the optimality of EDF to study the processor speed-up required to guarantee the feasibility of a required non-preemptive behavior for real-time tasks.

7.2.3 Execution Time Model

We assume a linear relationship between execution time and processor speed as assumed by [12] and [10]. This assumption can be easily relaxed by dividing the execution requirement into processor speed dependent and processor speed independent parts [15] which we discuss in section 7.6. In this paper we are focusing mainly on the theoretical consequences of resource augmentation on the preemption behavior, and hence we have assumed a linear speed-up of task execution times.

To ease the readability, and without loss of generality, we assume that the task set is initially executing on a processor of speed $S = 1$ and that the execution time of τ_i at that speed $S = 1$ is equal to the number of clock ticks required to execute it.

We assume that, if C_i^1 is the execution time at speed $S = 1$, the task execution time of τ_i scales linearly as follows:

$$C_i^S = \frac{C_i^1}{S}$$

Conversely, the speed S required to obtain an execution time of C_i^S is given by:

$$S = \frac{C_i^1}{C_i^S}$$

This model also allows us to use processor speed-up factors and processor speeds interchangeably. Changing the processor speed from $S = 1$ to $S = a$, is equivalent to speeding up the processor by a factor of 'a'.

7.3 Feasibility Analysis of Real-time Systems

Limited preemption models are considered to be generalizations of non-preemptive and preemptive scheduling models. A limited preemption scheduler can be used to simulate a non-preemptive or a preemptive schedule by setting the non-preemption parameters appropriately. The floating Non-Preemptive Model (f-NPR model) [7] [11] can be seen as a general scheduling model, i.e., if Q_i^S is set equal to 0, for all τ_i, the system simulates a fully preemptive model, while if Q_i^S is set equal to C_i^S, the system simulates a fully non-preemptive model [14]. In our approach we build on the f-NPR scheduling paradigm to study the feasibility of preemptive, non-preemptive and limited preemptive scheduling of real-time tasks, with or without synchronization schemes, by varying the

length of the non-preemptive regions of the tasks by changing the processor speed.

Let us now recall with modifications, some previously published theoretical results on the feasibility of real-time tasks. Due to the sustainability of the EDF scheduling algorithm [16], we can generalize the following theorems to a processor of speed S ($S \geq 1$). Baruah et. al. [5] derived the demand bound function, which, for any time interval, calculates the total processor time requested by the jobs of a task completely scheduled in the interval. A real-time task set is feasible if the cumulative processor time requested by the set of tasks during any time interval does not exceed the size of that time interval.

The following theorem determines the feasibility of uni-processor scheduling based on the blocking tolerance (β_i^S) of the tasks.

Theorem 7.3.1. *[7] [11] A task set is feasible on a speed S processor, if and only if,* $\forall i \in [1, n]$,

$$\beta_i^S \geq 0$$

where, β_i^S *is given by,*

$$\beta_i^S = \min_{D_i \leq t < D_{i+1}} \left(t - \sum_{j=1}^{n} DBF_j^S(t) \right)$$

$$t = kT_j + D_j, \forall k \in N, j \in [1, n]$$

In the above theorem, D_{n+1} is set as,

$$D_{n+1} = min\,(LCM, P)$$

Where,

$$P = max \left\{ D_1, D_2, ..., D_n, \frac{\sum_{i=1}^{n} (T_i - D_i) U_i^S}{1 - U^S} \right\}$$

When the $\beta_i^S = 0, \forall i \in [i, n]$, the task set is schedulable only under a fully preemptive scheduling scheme.

The above theorem can be used to determine the feasibility of limited preemption scheduling on a processor at speed S and is stated by the following theorem.

Theorem 7.3.2. *[7] [11] [17] A task set is feasible under limited preemptive scheduling on a speed S processor, if* $\forall i \in [1, n]$,

$$B_i^S \leq \beta_i^S$$

where, β_i^S is given by,

$$\beta_i^S = \min_{D_i \leq t < D_{i+1}} \left(t - \sum_{i=1}^{n} DBF_i^S(t) \right)$$

$$t = kT_j + D_j, \forall k \in N, j \in [1, n]$$

and B_i^S is the largest blocking actually experienced by τ_i due to the limited preemptions on a processor of speed S.

The bound Q_k^S on the length of the non-preemptive region of a task τ_k, on a processor of speed S, is given by the following theorem.

Theorem 7.3.3. *[17] A task set is feasible under limited preemptive scheduling on a speed S processor, if $\forall k \in [1, n]$,*

$$Q_k^S = \min_{1 \leq i < k} \beta_i^S$$

The task can execute entirely non-preemptively if the calculated value of Q_k^S is greater than its execution time C_k^S. We can use the above theorem to state the non-preemptive feasibility of the task set i.e., whether it is possible to find a non-preemptive schedule.

Theorem 7.3.4. *[8] [7] [17] A task set is feasible under non-preemptive scheduling on a speed S processor, if $\forall k \in [1, n]$,*

$$C_k^S \leq Q_k^S$$

7.4 Quantifying the Sub-Optimality of Non-Preemptive Scheduling

In this section, we derive the resource augmentation bound that guarantees the feasibility of a specified non-preemption behavior of a given task set. An illustration of the main contribution of this paper is presented in figure 7.1. We refer to the figure as the *feasibility bucket*. The depth of the bucket provides the processor speed-up required to guarantee a specified preemption behavior. Each cross-section of the bucket indicates the set of all task sets feasible under a limited preemption behavior, guaranteed at the corresponding processor speed. The radius is largest at the mouth of the bucket which denotes the set of all uniprocessor feasible task sets on a processor of speed $S = 1$. The radius, which

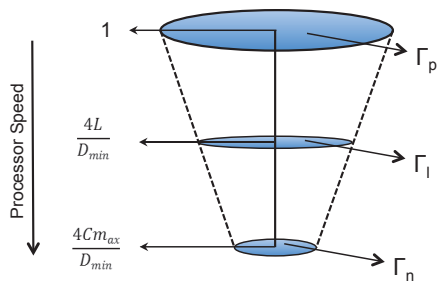
indicates the size of the set, is the smallest at its base, for the set of all task sets that is feasible under a fully non-preemptive scheduling scheme. On increasing the processor speed, the set of all uni-processor feasible task sets becomes feasible under a corresponding limited preemption scheduling scheme, finally becoming feasible under a fully non-preemptive scheme at speed $S = \frac{4C_{max}}{D_{min}}$.

Definition 1. *A non-preemption requirement on a task τ_i is defined as the lower bound on the lengths of the non-preemptive regions of τ_i, that, for example, guarantees a user defined upper bound on the preemption related cost on τ_i.*

We denote the non-preemption requirement on a task τ_i at speed S by L_i^S. A non-preemption requirement can also be denoted by L_i in case it does not change with the processor speed. We formally define the feasibility of a *specified non-preemption behavior* as follows.

Definition 2. *The feasibility of a specified non-preemption behavior of a task set is defined as the existence of a real-time schedule that guarantees the non-preemptive execution of every task for a user specified time duration, given by the non-preemption requirement.*

The feasibility of the *specified non-preemption behavior* of the task set can be guaranteed by guaranteeing the feasibility of the *specified non-preemption requirement* for every task τ_i in the task set.



Γp: Set of all uni-processor feasible task sets

Γn: Set of all task sets that are non-preemptive feasible on a uniprocessor

Γl: Set of all task sets where every task is guaranteed a non-preemption requirement of L

Figure 7.1: The feasibility bucket

We assume that during any arbitrary time interval, the processor is busy executing only the higher priority jobs and derive the maximum speed-up factor that guarantees the feasibility of a *specified non-preemption requirement* during that time interval. We then derive the upper-bound on the required processor speed-up factor that guarantees the feasibility of a non-preemptive schedule, thereby allowing us to quantify the sub-optimality of non-preemptive scheduling with respect to an optimal scheduling scheme. Later, we use this bound, in section 7.5, to derive the optimal processor speed-up factor that guarantees a specified upper-bound on the cumulative preemption related costs.

Let us now derive the processor speed that guarantees the feasibility of a non-preemption requirement L_i for a task τ_i.

Theorem 7.4.1. *The processor speed S_i that guarantees the feasibility of a non-preemption requirement L_i for a task τ_i is given by,*

$$S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\sum_{j=1}^{n} DBF_j^1(t)}{t - L_i} \right\}$$

Proof. The length of the non-preemptive region for τ_i at speed 1 is given by [7],

$$Q_i^1 = \min_{D_1 \leq t < D_i} \left\{ t - \sum_{j=1}^{n} DBF_j^1(t) \right\}$$

$$\Rightarrow Q_i^1 \leq t - \sum_{j=1}^{n} DBF_j^1(t), \forall t, D_1 \leq t < D_i$$

Our aim is to find the processor speed S_i that guarantees the feasibility of a non-preemption requirement L_i. Thus,

$$L_i \leq t - \frac{\sum_{j=1}^{n} DBF_j^1(t)}{S_i}, \forall t, D_1 \leq t < D_i$$

Solving for S_i, we get,

$$S_i \geq \left\{ \frac{\sum_{j=1}^{n} DBF_j^1(t)}{t - L_i} \right\}, \forall t, D_1 \leq t < D_i$$

i.e.,

$$S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\sum_{j=1}^{n} DBF_j^1(t)}{t - L_i} \right\}$$

□

Thus the processor speed S_{opt}, that guarantees the feasibility of a non-preemption requirement L_i for a task τ_i is the maximum of the processor speeds that guarantees the non-preemption requirement for each task individually.

Corollary 7.4.1. *The optimal processor speed S_{opt} that guarantees the feasibility of a non-preemption requirement L_i for a task τ_i is given by,*

$$S_{opt} = max(S_i)$$

We now find the upper-bound on the required processor speed that can guarantee a non-preemption requirement L_i for any task τ_i.

Lemma 7.4.1. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i, during a time interval t is upper-bounded by,*

$$S_i \leq \frac{y}{y-1}$$

where, $y = \frac{t}{L_i}$ $\forall t \in [D_1, D_i)$.

Proof. In order to ensure a specified non-preemption requirement L_i, the total processor demand during any time interval must be decreased such that a slack with a length of at most L_i is generated:

$$\sum_{j=1}^{n} DBF_j^1(t) - \frac{\sum_{j=1}^{n} DBF_j^1(t)}{S_i} \leq L_i$$

Solving for S_i gives,

$$S_i \leq \frac{\sum_{j=1}^{n} DBF_j^1(t)}{\sum_{j=1}^{n} DBF_j^1(t) - L_i}$$

The maximum value of $\sum_{j=1}^{n} DBF_j^1(t)$ on a processor at speed 1 is t, since we assume that the task set is feasible under a preemptive scheme. Substituting for $\sum_{j=1}^{n} DBF_j^1(t) = t$, we get:

$$S_i \leq \frac{t}{t - L_i}$$

And, finally, substituting $y = \frac{t}{L_i}$,

$$S_i \leq \frac{y}{y-1}$$

□

We now find the upper-bound on the required processor speed that guarantees the feasibility of a non-preemption requirement L_i, for any τ_i, during any time interval t such that L_i is no greater $\frac{t}{2}$.

Lemma 7.4.2. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i is upper-bounded by 2, if $\frac{t}{L_i} \geq 2$, $\forall t \in [D_1, D_i)$.*

Proof. Evaluating the limit of the equation in lemma 7.4.1 at $y = 2$, we get,

$$S_i = 2$$

Evaluating the limit using l'Hopital's rule as y tends to infinity (∞), we get,

$$S_i = 1$$

For any value of $y \in [2, \infty]$,

$$S_i \leq 2$$

□

We now find the upper-bound on the required processor speed that guarantees the feasibility of a non-preemption requirement L_i for any τ_i when $1 \leq \frac{t}{L_i} < 2$ in any time interval t.

Lemma 7.4.3. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i is upper-bounded by 4, if $1 \leq \frac{t}{L_i} < 2$ $\forall t \in [D_1, D_i)$.*

Proof. Since $1 \leq \frac{t}{L_i} < 2$, we have $t \geq L_i$ and $t < 2L_i$ Let us now increase the processor speed by a factor of 2. We effectively have $t' = 2t$ clock ticks in the time interval t on a processor of speed 2. Thus, $\frac{t'}{L_i} \geq 2$ since $1 \leq \frac{t}{L_i} < 2$. By using lemma 7.4.2, the speed-up S_i' required to guarantee the non-preemption requirement, for the increased processor speed, is upper-bounded by 2. As that we have already increased the processor speed by a factor of 2, the upper-bound on the processor speed that guarantees the non-preemption requirement for the case $1 \leq \frac{t}{L_i} < 2$ is $S_i \leq 4$. □

Observation 7.4.1. *The speed S_i that guarantees the feasibility of a specified non-preemption requirement L_i for any task τ_i is exactly upper-bounded by 2 if $1 \leq \frac{t}{L_i} < 2$ $\forall t \in [D_1, D_i)$.*

We know that, $1 \leq \frac{t}{L_i} < 2$ and thus, we have $t \geq L_i$ and $t < 2L_i$. At speed $S = 1$, there are t clock ticks available in any time interval t. Thus it is evident that in a worst case, when the processor is fully occupied during the interval t, L_i computations of the non-preemption requirement can not be feasibly executed within the interval t. Let us assume an increase in the processor speed by a factor of 2. This implies that within an interval of time t, there are in effect $t' = 2t$ clock ticks. In this case, it is clear that $2t \geq L_i + t$ since $t \geq L_i$. Thus, a non-preemption requirement L_i can be successfully scheduled within t without causing any deadline miss. Hence, the speed-up required for this case is exactly upper-bounded by 2.

Lemma 7.4.4. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i is upper-bounded by $\frac{4L_i}{t}$, if $0 < \frac{t}{L_i} < 1$, $\forall t \in [D_1, D_i)$.*

Proof. In this case we know that $t < L_i$. Let us now assume an increase in the processor speed to $S = \frac{L_i}{t}$. The number of available clock ticks in the time interval t increases from t to $t' = t \times \frac{L_i}{t} = L_i$. We thus obtain, $\frac{t'}{L_i} = 1$. This is a special case of lemma 7.4.3, and hence the speed-up S'_i required to guarantee the non-preemption requirement of τ_i is,

$$S'_i \leq 4$$

Since we had already increased the processor speed by $\frac{L_i}{t}$, the upper-bound on the actual speed S_i is:

$$S_i \leq \frac{4L_i}{t}$$

□

Observation 7.4.2. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i is exactly upper-bounded by $\frac{2L_i}{t}$ if $0 < \frac{t}{L_i} < 1$ $\forall t \in [D_1, D_i)$.*

On increasing the processor speed to $S = \frac{L_i}{t}$, the number of clock ticks in the time interval t increases from t to $t' = t \times \frac{L_i}{t} = L_i$. We can now execute the original t computations, and the $L_i - t$ computations of the non-preemption requirement L_i, using the L_i clock ticks in the time interval t, at speed $S = \frac{L_i}{t}$. Let the remaining non-preemption requirement that cannot be executed without a deadline miss in the interval t, be denoted by $L'_i = t$. We know that $t < L_i$, thus, in effect we get $\frac{t'}{L'_i} = \frac{L_i}{t} > 1$. Using lemma 7.4.2 and

the exact upper-bound for the case $1 \leq \frac{t}{L_i} < 2 \, \forall t \in [D_1, D_i)$, the exact upper-bound on the speed, denoted by S'_i, that ensure the specified non-preemption requirement L'_i is 2 i.e.,

$$S'_i \leq 2$$

Since we had already increased the processor speed by $\frac{L_i}{t}$, the exact upper-bound on the actual speed S_i is:

$$S_i \leq \frac{2L_i}{t}$$

We now find the upper-bound on the processor speed that guarantees a non-preemption requirement L_i for any task τ_i in the general case.

Theorem 7.4.2. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i is upper-bounded by $\frac{4L_i}{t} \; \forall t \in [D_1, D_i)$.*

Proof. In the general case, L_i is bounded by the maximum of the execution times of the tasks at speed $S = 1$ (i.e., for its non-preemptive execution) in the task set, and t by the shortest deadline and hence $\frac{t}{L_i} > 0$.

It follows from lemmas 7.4.2 7.4.3 and 8.5.4 that the speed-up required in the general case is $\frac{4L_i}{t}$. When $\frac{t}{L_i} \geq 2$, we obtain $S_i \leq \frac{4}{\frac{t}{L_i}} = \frac{4}{2} = 2$ and when $1 \leq \frac{t}{L_i} < 2$, we obtain $S_i \leq \frac{4}{\frac{t}{L_i}} = \frac{4}{1} = 4$ and when $0 < \frac{t}{L_i} < 1$ the speed-up required is $S_i \leq \frac{4L_i}{t}$.

Thus, for any $\frac{t}{L_i} > 0$, the speed-up required is $S_i \leq \frac{4L_i}{t}$. □

Hence, we have derived the upper-bound on the processor speed that guarantees the feasibility of a specified non-preemption behavior for any task set Γ. We use this upper-bound later in section 7.5.2 to find the exact processor speed-up that can guarantee a specified preemption behavior to control the preemption related costs.

We now examine the feasibility of a non-preemptive schedule for any task set Γ.

Corollary 7.4.2. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i is upper-bounded by $\frac{4L_i}{D_{min}}$.*

This is straightforward as the value of t that maximizes $\frac{4L_i}{t}$, $t \in [D_1, D_i)$, is the smallest value of t given by the shortest relative deadline $t = D_{min}$ (remember that $D_{min} = D_1$).

Corollary 7.4.3. *The speed S_i that guarantees the feasibility of the non-preemptive execution of τ_i is upper-bounded by $\frac{4C_i^1}{D_{min}}$.*

This is easily seen from theorem 7.4.2 by substituting $L_i = C_i^1$ and using the smallest value of t, which is the shortest deadline D_{min}. Moreover, Baruah [7] derived the feasibility of limited preemption scheduling by building on the optimality of the Earliest Deadline First (EDF) scheduling. We have derived the resource augmentation bounds based on the feasibility of limited preemptive scheduling proposed by Baruah [7]. Hence our resource augmentation bounds guarantees the feasibility of the specified non-preemption behavior.

Corollary 7.4.4. *The speed S that guarantees the feasibility of a non-preemptive execution of all the tasks in the task set is upper-bounded by $\frac{4C_{max}}{D_{min}}$, where $C_{max} = max(C_i^1) \forall \tau_i \in \Gamma$.*

The limited-preemptive execution of the tasks are independent of each other [7]. Hence, the processor speed-up that guarantees the non-preemptive execution of the task with the largest execution time, will also guarantee the non-preemptive execution of other tasks.

Hence, the sub-optimality of non-preemptive scheduling with respect to an optimal uni-processor preemptive scheduling scheme for any task set Γ is given by $\frac{4C_{max}}{D_{min}}$ where $C_{max} = max(C_i^1) \forall \tau_i \in \Gamma$.

7.5 Guaranteeing a Specified Preemption Behavior using Processor Speed-up

In the previous section, we derived the upper-bound on the required processor speed-up that guarantees the feasibility of a user specified non-preemption behavior. In this section, we apply this bound to guarantee a user specified upper-bound on the preemption related costs in the schedule. This is achieved by first deriving the non-preemption requirement that guarantees the upper-bounds on the preemption related costs and then finding the processor speed-up that guarantees the feasibility of the derived non-preemption requirement. We propose a sensitivity analysis method that uses the bound derived in section 7.4, to calculate the optimal processor speed that guarantees the derived non-preemption requirements which in turn guarantees specified upper-bound on the preemption related costs.

Definition 3. *The optimal processor speed S_{opt} that guarantees the feasibility of a specified non-preemption behavior is defined as $S_{opt} = min(S)$, where*

$S \in$ the set of available processor speeds such that, $\forall\ \tau_i$ in Γ,

$$Q_i^{S_{opt}} \geq L_i^{S_{opt}}$$

Here, $L_i^{S_{opt}}$, is the specified length of the non-preemptive region for τ_i (its non-preemption requirement), that guarantees the feasibility of a specified non-preemption behavior per τ_i.

According to our scheduling model, the length Q_i^S determines the upper-bound on the preemption related costs in the schedule e.g., upper-bound on the number of preemptions or the possibility of preemptions only at optimal preemption points. However, if, for any task τ_i, Q_i^S does not guarantee the specified upper-bound on the preemption related costs, we can derive the non-preemption requirement L_i^S that guarantees the specified bound. We can then calculate the speed-up required to guarantee the feasibility of the non-preemption requirement L_i^S, which will in turn guarantee the specified upper-bounds on the preemption related costs.

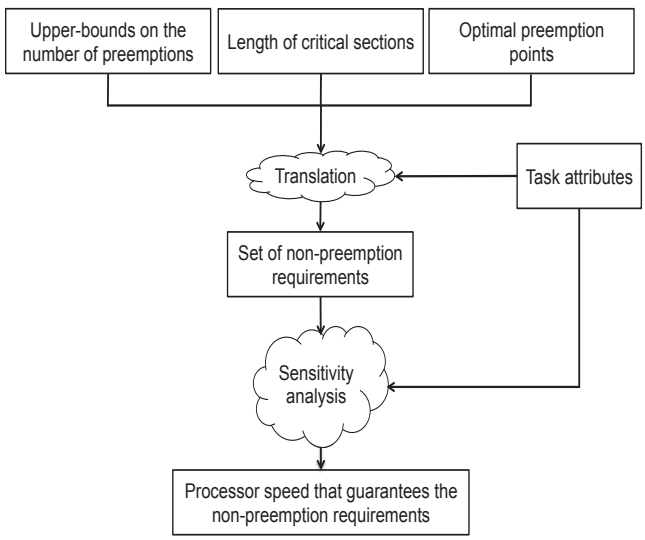


Figure 7.2: Methodology overview

Methodology Overview : Our approach is performed in two steps. In the first step, we translate the requirements of meeting the specified upper bounds on the preemption related costs into a set of non-preemption requirements. We then perform a sensitivity analysis using the task parameters and the non-preemption requirements to derive the optimal processor speed that guarantees the desired non-preemption behavior. An overview of our methodology is given in figure 7.2. In the following two sections, we describe each of the steps in detail.

7.5.1 Translating Preemption Cost Control Requirements to Non-Preemption Requirements

The key idea behind the approach is that the system level requirements of meeting specified bounds for various preemption related costs can be translated into a set of task level non-preemption requirements, i.e., a lower bound on the length of the non-preemptive regions for each task.

In the following, we outline the first step in our approach i.e., the translation of preemption cost control requirements to specified non-preemption requirements for each task that will, for example, lower the overall preemption related overheads. Our aim is to find the optimal processor speed that will guarantee that the length of the non-preemptive region per task at that speed is greater than or equal to the non-preemption requirement.

Controlling the Number of Preemptions

The cumulative preemption related costs for a task set depends both on the number of preemptions as well as the points at which the preemptions occur. The maximum number of times a task τ_i, characterized by a non-preemptive region Q_i^S, can be preempted while executing on a speed S processor is given by [7] [14]:

$$\left\lceil \frac{C_i^S}{Q_i^S} \right\rceil - 1$$

If in order to guarantee no more than p_i preemptions per τ_i, its non-preemption requirement L_i^S on a speed S processor, should be:

$$L_i^S \geq \frac{C_i^S}{p_i + 1} \Rightarrow L_i^S = \left\lceil \frac{C_i^S}{p_i + 1} \right\rceil \tag{7.1}$$

Note that Q_i^S is the actual length of the non-preemptive region of τ_i at speed S, i.e., given by the original task attributes, and L_i^S is the lower bound on the

lengths of the non-preemptive region of τ_i that will guarantee no more than p_i preemptions per τ_i. When $p_i = 0$, the task τ_i executes non-preemptively.

It is evident that, on a speed 1 processor, if $Q_i^1 < L_i^1$, where L_i^1 is calculated according to equation 7.1, τ_i cannot be guaranteed to incur no more than p_i preemptions. Hence, we have to find a processor speed, S, which ensures that:

$$Q_i^S \geq L_i^S = \left\lceil \frac{C_i^S}{p_i + 1} \right\rceil \tag{7.2}$$

Thus if the processor runs at speed S, τ_i can be guaranteed to be preempted at

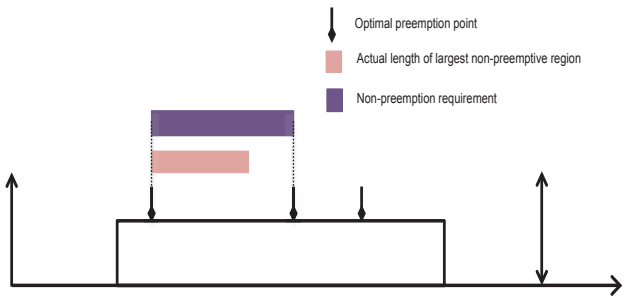


Figure 7.3: Non-preemption requirement to enable preemptions only at optimal preemption points

most p_i times.

Enabling Preemptions at Optimal Preemption Points

As mentioned earlier the preemption related costs also depends on the points at which the preemptions occur. If a preemption cannot be avoided, it is preferable to have it at points where the cost of a preemption is the least, i.e., optimal preemption points. The possibility of enforcing preemptions only at these optimal preemption points depends on the length of the non-preemptive region on a processor of a given speed S. Remember that $q_{i,j}^S$, $j = 1...m_i$ denote the length of the task execution of τ_i up to its j^{th} optimal preemption point on a processor at speed S. Hence, the non-preemption requirement for a task τ_i is given by the largest interval between any two consecutive optimal preemption points of τ_i when it executes on a processor at a speed S:

$$L_i^S = \max_{1 \leq j < m} (q_{i,j+1}^S - q_{i,j}^S, q_{i,1}^S)$$

Consequently, our goal is to find the processor speed, S, that satisfies:

$$Q_i^S \geq L_i^S = \max_{1 \leq j < m} (q_{i,j+1}^S - q_{i,j}^S, q_{i,1}^S) \tag{7.3}$$

Thus if the processor executes at speed S, any premption on τ_i can be deferred to the closest optimal preemption point. An illustrative example is given by figure 7.3 where the non-preemption requirement of a task that guarantees preemptions only at optimal preemption points, is greater than its non-preemptive execution.

Executing Critical Sections within Non-Preemptive Regions

An attractive feature of the limited preemption scheduling paradigm is that it has the potential to enable access to shared resources without the need for synchronization mechanisms, by executing the critical sections within non-preemptive regions. However, this is not always possible, e.g., in case the length of the non-preemptive region Q_i^S, for a task τ_i is shorter than its largest critical section CS_i^S on a processor of speed S. This issue, on the other hand, can be solved by finding an adequate processor speed S that enables $Q_i^S \geq L_i^S$. The processor speed that guarantees this is given by the speed S that will satisfy the relation:

$$Q_i^S \geq L_i^S = CS_i^S$$

An illustrative example is given by figure 7.4 where the non-preemption re-

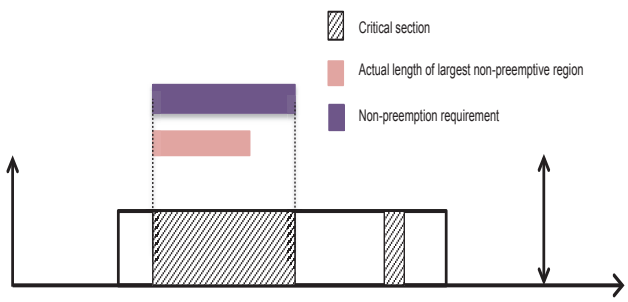


Figure 7.4: Non-preemption requirement to always enable critical section execution inside non-preemptive regions

quirement of a task that guarantees the non-preemptive execution of its critical sections is greater its actual length of the non-preemptive region.

7.5.2 Sensitivity Analysis for Preemption Control

In most situations, changing the processor speed may also change the required length of the non-preemptive regions to satisfy the desired preemption related cost control requirements. It also changes the possible lengths of the non-preemptive regions of some of the tasks in the system. Thus, we need to perform a sensitivity analysis on the task set to derive the required processor speed-up that guarantees a given non-preemption requirement. The length of the non-preemptive region of τ_i is known to be Q_i^1 at speed 1 (calculated using the results by [7] [11] [17]. If $Q_i^1 < L_i^1$, we need to use a faster processor to ensure that for each τ_i, its possible non-preemptive region satisfies the required non-preemption requirement. Remember that L_i^1 is the non-preemption requirement that guarantees a certain non-preemption behavior for τ_i on a speed 1 processor. Since we use a binary search to find the optimal speed S_{opt}, the challenge is to find an upper-bound on the processor speed below which S_{opt} lie. We can use the upper-bounds derived in the earlier section. The upper-bound on the processor speed-up that guarantees a non-preemption requirement of L_i^1 for any τ_i, on a processor of speed 1, is given by:

$$S_i = \frac{4L_i^1}{D_{min}}$$

Thus, the processor speed which guarantees that every task can execute non-preemptively is given by,

$$S_{high} = \max_{1 \leq i \leq n} (S_i)$$

However, when we use a faster processor, the execution times ($C_i^S \; \forall \; \tau_i$) of all the tasks decrease. This, in its turn, changes the lengths of the longest non-preemptive regions of the tasks. The lowest processor speed that guarantees the non-preemption requirements lies in the interval $[S_{low} = 1, S_{high}]$. We can now perform a sensitivity analysis on the speeds between 1 and S_{high} in order to come up with the lowest processor speed S_{opt} which guarantees that every task τ_i can exhibit the required non-preemption behavior. The correctness and optimality of our method is given by the correctness of the binary search.

In the algorithm, at each iteration of the binary search, $L_i^{S_{mid}}$ is re-calculated for the speed S_{mid} as follows:

$$L_i^{S_{mid}} = max(a, b, c)$$

Algorithm 2: Algorithm to find the processor speed S_{opt} that guarantees $Q_i^{S_{opt}} \geq L_i^{S_{opt}}$ for every τ_i.

```
1  S_low=1
2  S_high = 0
3  foreach Task i do
4      Calculate Q_i^{S_low}
5      Calculate L_i^{S_low}
6      S_i = 4L_i^{S_low} / D_min
7      if S_high < S_i then
8          S_high = S_i
9  while TRUE do
10     S_mid = (S_high + S_low) / 2
11     S_temp = 0
12     foreach Task i do
13         Calculate Q_i^{S_mid}
14         Calculate L_i^{S_mid}
15         S_i = 4L_i^{S_low} / D_min
16         if S_temp < S_i then
17             S_temp = S_i
18     if S_low ≠ S_high then
19         flag = 0
20         foreach Task i do
21             if L_i^{S_mid} > Q_i^{S_mid} then
22                 flag = 1
23         if flag = 1 then
24             S_low = S_mid
25             if S_temp < S_high then
26                 S_high = S_temp
27         else
28             S_high = S_mid
29     else
30         S_opt = S_mid
31         return S_opt
```

where a is given by,

$$a = \left\lceil \frac{C_i^{S_{mid}}}{p_i + 1} \right\rceil$$

b is given by,

$$b = CS_i^{S_{mid}}$$

and c is given by,

$$c = \max_{1 \leq j < m} (q_{i,j+1}^{S_{mid}} - q_{i,j}^{S_{mid}}, q_{i,1}^{S_{mid}})$$

The binary search converges when a suitable value of S_{mid} is obtained such that for every value of $S < S_{mid}$, $Q_i^S < L_i^S$ and for all values $S \geq S_{mid}$, $Q_i^S \geq L_i^S$. Thus, S_{mid} is the optimal speed that guarantees the non-preemption requirement for all the tasks $\tau_i \in \Gamma$.

7.5.3 Example

We illustrate our method using a simple example. Consider the task set given in table 7.1 executing on a processor of speed 1. We assume an execution time model, where the task executions scale linearly with the processor speed. In other words, if the processor speed is increased to 2 from a default speed of 1, the tasks execute twice as fast. The longest possible non-preemptive regions per task for each speed is given in table 7.2. Note that, at speed 1, there are 16 preemptions on task τ_2, 23 preemptions on τ_3, 19 preemptions on τ_4 and 26 preemptions on τ_5. Assume that τ_4 is a task controlling a physical system and more than 3 preemption on τ_2 will degrade the physical system. Thus, we want to ensure that task τ_4 in the task set is preempted no more than 3 times, which is, however, not possible on a speed 1 processor. We perform a sensitivity analysis, as described in the previous section to find the lowest processor speed that guarantees that task τ_4 is preempted no more than thrice. In this case, our algorithm gave an output of $S_{opt} = 3.4$. In order to show that our derived speed is the lowest one which can guarantee the desired non-preemption behavior, we also calculated the number of preemptions that the task set would incur at a speed less than S_{opt}. We observed that, for speeds, arbitrarily lower than S_{opt}, the desired non-preemption property cannot be satisfied, as seen from the table 7.3.

The execution times, length of the longest non-preemptive regions and the longest number of preemptions possible at this speed are calculated and enumerated in table 7.3. It can be easily seen that using a processor of speed

Task	C_i^1	D_i	T_i
τ_1	2	5	50
τ_2	50	230	230
τ_3	70	360	370
τ_4	60	900	900
τ_5	80	990	1000

Table 7.1: Example task set (speed=1)

Task	**Speed (S)=1**			**Speed (S)=3.4** (S_{opt})		
	C_i^S	Q_i^S	No. of preemptions	C_i^S	Q_i^S	No. of preemptions
τ_1	2	2	0	0.588235	0.588235	0
τ_2	50	3	16	14.705882	4.411765	3
τ_3	70	3	23	20.588234	4.411765	4
τ_4	60	3	19	17.647058	4.411765	3
τ_5	80	3	26	23.529411	4.411765	5

Table 7.2: The task execution times, length of the longest non-preemptive regions and the number of preemptions at different processor speeds

$S = 3.39999$ increases the number of preemptions on τ_4 to 4, while it was 3 at the optimal speed $S_{opt} = 3.4$.

7.6 Discussions

Let us now discuss how we can relax some of the assumptions made in this paper. Specifically, we consider relaxing the assumptions of linear speed-up and negligible preemption related overheads at optimal preemption points, which we address partially.

7.6.1 Relaxing the Assumption of Linear Speed-up

We have considered the possibility of a linear speed-up in this paper. While such an assumption allows us to derive interesting theoretical results on the resource augmentation bounds for preemption control, it might not be a reasonable assumption from a practical point of view because of the effects of memory wall [18]. In this sub-section, we consider relaxing the assumption of a linear speed-up by considering the execution time model proposed by [15].

Task	Speed (S)=3.39999 ($< S_{opt}$)		
	C_i^S	Q_i^S	No. of preemptions
τ_1	0.588237	0.58824	0
τ_2	14.705925	4.411763	3
τ_3	20.588295	4.411763	4
τ_4	17.647110	4.411763	**4**
τ_5	23.529480	4.411763	5

Table 7.3: The number of preemptions at a speed arbitrarily less than S_{opt}

Marinoni et. al. [15] assumed that the execution time of a task consists of two parts- one that scales linearly with the processor frequency and one that does not scale with the processor frequency. Following their notation, let us define ϕ as the percentage of the execution time for every τ_i, that scales with the processor frequency.

Let us see how the theorem 7.4.1 changes when we consider our new assumption.

Theorem 7.6.1. *The processor speed S_i that guarantees the feasibility of a non-preemption requirement L_i for a task τ_i is given by,*

$$S_i = \max_{D_1 \le t < D_i} \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1-\phi)\sum_{j=1}^{n} DBF_j^1(t)} \right\}$$

Proof. The length of the non-preemptive region for τ_i at speed 1 is given by [7] [11],

$$Q_i^1 = \min_{D_1 \le t < D_i} \left\{ t - \sum_{j=1}^{n} DBF_j^1(t) \right\}$$

$$\Rightarrow Q_i^1 \le t - \sum_{j=1}^{n} DBF_j^1(t), \forall t, D_1 \le t < D_i$$

Our aim is to find the processor speed S_i that guarantees the feasibility of a non-preemption requirement L_i. We know that, of the total demand bound in any interval t, only ϕ percentage scales with the processor frequency. Thus,

$$L_i \le t - \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{S_i} + (1-\phi) \sum_{j=1}^{n} DBF_j^1(t) \right\}, \forall t, D_1 \le t < D_i$$

Hence,

$$S_i L_i \leq S_i t - \left\{ \phi \sum_{j=1}^{n} DBF_j^1(t) + S_i(1-\phi) \sum_{j=1}^{n} DBF_j^1(t) \right\}, \forall t, D_1 \leq t < D_i$$

Solving for S_i, we get,

$$S_i \geq \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1-\phi) \sum_{j=1}^{n} DBF_j^1(t)} \right\}, \forall t, D_1 \leq t < D_i$$

i.e.,

$$S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1-\phi) \sum_{j=1}^{n} DBF_j^1(t)} \right\}$$

□

Lemma 7.6.1. *The speed S_i that guarantees the feasibility of a non-preemption requirement L_i for any task τ_i, during a time interval t is upper-bounded by,*

$$S_i \leq \frac{\phi y}{\phi y - 1}$$

where, $y = \frac{t}{L_i}$, $\forall t \in [D_1, D_i)$.

Proof. We know from theorem 7.6.1 that,

$$S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1-\phi) \sum_{j=1}^{n} DBF_j^1(t)} \right\}$$

Since we have assumed that the task set is feasible, the upper-bound on the value of $\sum_{j=1}^{n} DBF_j^1(t)$ is t. Hence,

$$S_i \leq \left\{ \frac{\phi t}{t - L_i - (1-\phi)t} \right\}$$

which gives,

$$S_i \leq \frac{t}{\phi t - L_i}$$

Finally, substituting $y = \frac{t}{L_i}$,

$$S_i \leq \frac{\phi y}{\phi y - 1}$$

□

We can now use similar reasoning as in lemma 7.4.2, 7.4.3 and 8.5.4 to derive upper-bounds on the processor speed-up required that provides the required non-preemption guarantees. We however leave more details to a future work. In future, we plan to consider more realistic execution time models to find upper-bounds on the processor speed-up to provide the required guarantees.

7.6.2 Relaxing the Assumption of Negligible Preemption Related Overheads at Optimal Preemption Points

The assumption of negligible preemption related overheads at optimal preemption points can be very conservative for actual systems. This however can be easily accounted for in the worst case execution times of the tasks without being pessimistic. To account for the preemption related costs in our proposed method, the optimal preemption points must be identified first and the preemption related costs at these points should be calculated. These preemption related costs must then be considered while calculating the non-preemption requirement. We can then use an optimal preemption point placement strategy e.g., proposed by Bertogna et. al. [17]. This will enable the application of our method to real world applications. We however leave more details to a future work.

7.7 Related Work

Preemptive real-time schedulers are associated with preemption related overheads, and their effects are challenging to analyze because they typically vary with the point of preemption e.g., cache related preemption delays, and even with the state of the physical process that the real-time system is controlling. Moreover, preemptive scheduling typically requires the use of resource access protocols [19] to enable mutual exclusion, in cases where tasks communicate through shared resources. These resource access protocols, though predictable, introduce schedulability overheads in the system, as well as may lead to pessimistic assumptions in the schedulability analysis. Even though preemptive scheduling schemes are used in a large number of applications, mostly due to its ability to achieve high processor utilization, the detrimental impact of preemptions is widely recognized in the community [20] [21] [22]. The preemption related costs includes the context-switch overhead [23] and to manipulate the task queues [21], as well as the indirect cost of cache-related preemption

delays [24]. Bui et. al [20] observed a worst case increment in task execution time of upto 33% on a PowerPC MPC7410 with a 2 MB two way associative L2 cache, due to the cache related preemption delays. The worst case temporal overhead due to cache related preemption delays were found to be as high as $655\mu S$ for a single preemption. The rate monotonic algorithm (RM) was shown to introduce a higher number of preemptions than earliest deadline first algorithm (EDF) by Buttazzo [25], increasing the overheads in the system and hence reducing the benefits of its simple runtime implementation.

The applicability of a non-preemptive scheduling scheme, on the other hand, is limited to only a small fraction of the feasible task sets [5] due to its inability to fully utilize the computational resources in most of the cases [7]. Task sets scheduled by non-preemptive schemes can be deemed as unschedulable even at arbitrarily low utilizations [14]. On the other hand, the major benefits of using non-preemptive scheduling is the absence of preemption related costs and that it does not require any resource access protocols, as mutual exclusion is guaranteed by the scheduler. However, the problem of finding a non-preemptive schedule for a given task set is either NP-hard or infeasible for most task sets [8].

In order to take advantage of the benefits of both preemptive and non-preemptive scheduling paradigms, various limited preemption scheduling models were proposed, a detailed survey of which can be found in [26]. A benefit of using these approaches is that non-preemptive regions can be enforced within the task executions. The use of a limited preemption technique reduces the preemption related costs, i.e., it basically ensures that a preemption occurs only when absolutely necessary and/or at an optimal point with the least cost. It also ensures that, whenever possible, critical sections can be executed entirely non-preemptively. However, the limited preemption approach does not provide for a fine grained ability to control the preemption behavior of real-time tasks e.g., to guarantee a user specified bound on the number of preemptions per task. To guarantee mutual exclusion during critical section execution, it is essential that the non-preemptive region of each task is larger than its largest critical section. Similarly, to minimize preemption related costs, the length of the non-preemptive region of any task must be no less than the length of the task execution between any two consecutive optimal preemption points, where the cost of a preemption is the least. If the length of the non-preemptive region does not guarantee a specified preemption behavior, a preemptive or non-preemptive schedule may not be feasible. For example, if the preemptions are not possible at optimal preemption points, it may increase the task execution time by 33% [20], potentially causing deadline misses. Augmenting the limited preemption

scheduling schemes with the flexibility for specifying a certain non-preemption behavior can further enhance its applicability in modern real-time systems to control the preemption related costs.

Of the several methods that have been proposed to reduce the number of preemptions in real-time scheduling, Preemption Threshold Scheduling (PTS) for FPS was first introduced in the ThreadX operating system by Lamie [27]. This scheme was later formalized by Wang and Saksena [28], by providing an accurate analysis and also proposing an optimal algorithm to calculate the preemption thresholds. Jeffay et. al. [8] derived the sufficient and necessary conditions for non-preemptive feasibility of periodic and sporadic tasks. They also showed that EDF is an optimal algorithm for scheduling tasks non-preemptively under a work conserving paradigm. [7] proposed the limited preemption model, which was subsequently named as Floating Non-Preemptive Region model (f-NPR model), in which they proposed an algorithm to calculate the length of the longest possible non-preemptive execution of a task in a sporadic task system. Later, Bertogna and Baruah[11] evaluated the approach for randomly generated task sets, showing its effectiveness. In an earlier work, they had extended the f-NPR scheduling scheme to FPS [29], where they found an upper bound on the length of the largest possible non-preemptive execution of a task under FPS and presented extensive simulation results. Later, Bertogna et. al. [17] presented a method to optimally place preemption points within the task code, assuming a fixed preemption overhead, and presented extensive simulation results for both EDF and FPS. The evaluation results showed that the limited preemption models are more effective than non-preemptive and fully preemptive scheduling schemes with preemption costs, in successfully scheduling task sets. Bertogna et. al. [30] also proposed a method to improve the schedulabilility of FPS by executing the last portion of the tasks in a non-preemptive fashion, as long as possible. Dobrin and Fohler [31] proposed a method to identify preemption offline and to control the number of preemptions by changing the task parameters, such as the priority, to avoid these conditions required for a preemption. An extensive survey of the various limited preemption methods can be found in [26] and a detailed comparison can be found in [14].

The energy consumption in a processor can be modeled by the relation $P = CV^2F$, where P is the power consumed by the processor, V is the applied voltage, C is the effective capacitance and F is the operating frequency [32]. This relation indicates that using a slower processor would decrease the energy consumption. However, when using Dynamic Voltage Scaling (DVS) [32] for energy efficiency, due to an increase in task execution times, the number of pre-

emptions increases significantly. Pouwelse et. al. [33] found that a frequency switch can be done in less than $140\mu S$, which amounts to just one fifth of the cost of a single preemption making CPU frequency scaling a promising approach towards controlling preemption behavior in real-time systems. In [34] [35] [36], we proposed methods to control the preemption behavior of sporadic and periodic tasks scheduled by FPS using CPU frequency scaling. In [37], an approach to combine PTS with DVS to enable energy efficient scheduling was presented. However, it does not provide for controlling the preemption behavior of the schedule.

All the above limited preemption approaches still require the support of a preemptive scheduler on top of which additional support mechanisms are required for their implementation e.g., dual priority scheduler for PTS [28] and timers for limited preemption models [7] [11]. None of the above methods are able to completely eliminate preemptions, even though they are able to greatly reduce the preemption overheads. These methods also do not provide for controlling the preemption related costs in the schedule e.g., ensuring no more than a user desired number of preemptions per task or guaranteeing that a preemption is always possible at an optimal preemption point in the task. In this context, we examine and prove the possibility of using a faster processor to achieve effective preemption related cost control in the schedule. Controlling the preemption related costs provides a system designer with the ability to perform trade-offs between, e.g., energy- preemption overhead trade-offs.

7.8 Conclusions

In this paper, we have derived the upper-bound on the required processor speed-up that guarantees the feasibility of a non-preemptive schedule for any task set that is feasible on a uni-processor. We have proved that the speed-up required to guarantee the non-preemptive execution of any task τ_i, for a duration L_i, is no greater than $\frac{4L_i}{D_{min}}$ where D_{min} is the smallest relative deadline in the task set. Consequently, the upper-bound on the processor speed that guarantees a *fully non-preemptive schedule* is given by $\frac{4C_{max}}{D_{min}}$, where C_{max} is the largest execution time in the task set. Our sensitivity analysis based method derives the optimal processor speed that guarantees specified upper-bounds on the preemption related costs in the schedule. In this method, we first translate the system level requirements of meeting specified upper bounds on the preemption related costs to a set of non-preemption requirements on the task set. We then use sensitivity analysis to calculate the optimal processor speed that guarantees

the derived non-preemption requirements. This in turn guarantees the specified bounds on the preemption related costs in the system. The method empowers a real-time system designer to guarantee user specified bounds on the preemption related costs, using a faster processor, while maintaining schedulability.

Ongoing efforts include extensions to multi-processor scheduling to guarantee a specified preemption behavior in multi-processor systems.

Bibliography

[1] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *The Journal of ACM*, 1973.

[2] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 1986.

[3] N.C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[4] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, 1974.

[5] Sanjoy K. Baruah, Louis E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 1990.

[6] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *The 11th Real-Time Systems Symposium*, 1990.

[7] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *The 17th Euromicro Conference on Real-Time Systems*, 2005.

[8] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *The 12th IEEE International Real-time Systems Symposium*, 1991.

[9] M. Marouf and Y. Sorel. Scheduling non-preemptive hard real-time tasks with strict periods. In *The 16th Conference on Emerging Technologies Factory Automation*, September 2011.

[10] Robert Davis, Thomas Rothvo, Sanjoy Baruah, and Alan Burns. Exact quantification of the sub-optimality of uniprocessor fixed priority preemptive scheduling. *Real-Time Systems*, 2009.

[11] M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *IEEE Transactions on Industrial Informatics*, November 2010.

[12] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 2000.

[13] Abhilash Thekkilakattil, Radu Dobrin, Sasikumar Punnekkat, and Hüseyin Aysan. Resource augmentation for fault-tolerance feasibility of real-time tasks under error bursts. In *The 20th International Conference on Real-Time and Network Systems*. ACM, November 2012.

[14] Gang Yao, G. Buttazzo, and M. Bertogna. Comparitive evaluation of limited preemptive methods. In *The 15th International Conference on Emerging Technologies and Factory Automation*, 2010.

[15] Mauro Marinoni and Giorgio Buttazzo. Elastic dvs management in processors with discrete voltage/frequency modes. *IEEE Transactions on Industrial Informatics*, February 2007.

[16] Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *The 27th IEEE International Real-Time Systems Symposium*, 2006.

[17] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *The 22nd Euromicro Conference on Real-Time Systems*, 2010.

[18] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, 2004.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *The IEEE Transactions on Computers*, 1990.

[20] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[21] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1995.

[22] H Ramaprasad and F Mueller. Tightening the bounds on feasible preemptions. In *The ACM Transactions on Embedded Computing Systems*, 2008.

[23] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *The IEEE Transactions on Software Engineering*, 1993.

[24] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 1998.

[25] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. In *Real-Time Systems Journal*, January 2005.

[26] G.C. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems: A survey. *The IEEE Transactions on Industrial Informatics*, 2012.

[27] William Lamie. Preemption threshold. *Whitepaper*, 1997.

[28] Yun Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Sixth International Conference on Real-Time Computing Systems and Applications, 1999. RTCSA '99.*, 1999.

[29] Gang Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[30] M. Bertogna, G. Buttazzo, and Gang Yao. Improving feasibility of fixed priority tasks using non-preemptive regions. In *The IEEE Real-Time Systems Symposium*, 2011.

[31] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *The 16th Euromicro Conference on Real-time Systems*, 2004.

[32] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *The 18th ACM symposium on Operating systems principles*, 2001.

[33] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *The 7th annual international conference on Mobile computing and networking*, 2001.

[34] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Towards preemption control using CPU frequency scaling in sporadic task systems. In *Proceedings of the WiP of The 6th International Symposium on Industrial Embedded Systems*, 2011.

[35] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Preemption control using CPU frequency scaling in real-time systems. In *The 18th International Conference on Control Systsems and Computer Science*, 2011.

[36] Abhilash Thekkilakattil, Anju S Pillai, Radu Dobrin, and Sasikumar Punnekkat. Reducing the number of preemptions in real-time systems scheduling by CPU frequency scaling. In *The 18th International Conference on Real-Time and Network Systems*, 2010.

[37] Ravindra Jejurikar and Rajesh K. Gupta. Integrating processor slowdown and preemption threshold scheduling for energy efficiency in real time embedded systems. In *The IEEE Real-Time Computing Systems and Applications*, 2004.

Chapter 8

Paper D: Resource Augmentation for Fault-Tolerance Feasibility of Real-time Tasks under Error Bursts

Abhilash Thekkilakattil, Radu Dobrin, Sasikumar Punnekkat and Huseyin Aysan
In proceedings of the 20th International Conference on Real-Time and Network Systems, ACM, Pont á Mousson, France, November, 2012 (*Shortlisted for Best Student Paper Award*)

Abstract

Dependability is a vital system requirement, particularly in safety critical and mission critical real-time systems, due to the potentially catastrophic consequences of failures. In most critical applications different fault tolerance mechanisms using redundancy are employed to prevent possible failures. In the case of real-time systems the system designer must ensure that the task set is feasible even under faults, which we refer to as 'fault tolerance feasibility'. Due to cost considerations, often temporal redundancy has been prevalently used to meet this objective.

In this paper we focus on guaranteeing fault-tolerance feasibility under error bursts on uni-processor systems by the usage of resource augmentation, specifically through processor speed-up. Firstly, we derive a processor demand bound based sufficient condition for a set of real-time tasks to be fault tolerance feasible under an assumption that no more than one error burst occurs during the hyper-period of the task set. Subsequently, we derive the necessary resource augmentation bounds (i.e., the processor speed-up), that guarantees the fault tolerance feasibility, if the sufficient test fails. Finally, we prove that, if the error burst length is no more than half the shortest relative deadline of the task set, the processor speed-up required to guarantee fault tolerance feasibility is upper-bounded by 6.

8.1 Introduction

Mission and safety critical real-time systems typically have to perform a number of functionalities of mixed criticality levels, ranging from ultra-critical to non-critical. In addition to the temporal correctness, these systems need to provide for a high degree of reliability, due to the catastrophic consequences a failure may lead to. The reliability of the system is typically achieved by the use of fault tolerance mechanisms that aim to prevent potential system failures while guaranteeing the real-time constraints. Consequently, any reasoning about the correctness of the system needs to take into account an appropriate fault model, as well as the overheads associated with the employed fault-tolerance mechanisms.

In a real-time system, the events occurring in the system are typically mapped to a set of real-time tasks, with the requirement that the task executions must complete by their respective deadlines. Additionally, reliability constraints require the use of an appropriate fault tolerance strategy, most commonly in the form of temporal redundancy, which involves the re-execution of the original task or the execution of an alternate task, before its predefined deadline [1][2][3][4]. In this context, the original task execution is typically referred to as the primary and the re-executions upon faults are referred to as alternates. If the fault occurrence persists during the execution of the recovery attempts, alternates are executed until one successful execution is achieved. In order to reason about the temporal correctness of the system, safe upper-bounds on the task execution times, i.e., the Worst Case Execution Times (WCET), derived using suitable techniques [5], are required. Additionally, the use of temporal redundancy requires that the schedulability analysis techniques consider the transient overloads generated by the execution of alternates in order to guarantee the overall system schedulability under fault occurrences.

Many safety critical and mission critical real-time systems employ a preemptive Fixed Priority Scheduler (FPS) due to its simple scheduling mechanism that enables an easy implementation, even on operating systems that do not provide explicit support for timing constraints [6]. However, in the general case, preemptive FPS may not be able to guarantee schedulability of the task sets if the total task utilization is greater than 69% [7]. Dynamic priority schemes, on the other hand, e.g., Earliest Deadline First scheduling [8] [7], have the ability to utilize the processor more effectively, and are being promoted in the academia [6], as well as in many commercial operating systems [9]. Hence a real-time systems designer can choose from a wide variety of scheduling schemes while designing the system. During the design stage of the

system, the choice of a scheduler is influenced by the task attributes. In most cases, the task attributes are derived from the physical characteristics of the environment that the real-time system is controlling and are thus unchangeable. Hence an important question is which scheduling scheme will yield a feasible schedule that can tolerate faults under a specified fault hypothesis. However, even before answering this question it is more appropriate to ask whether it is possible to determine *if there exists* a scheduling scheme that can tolerate faults under the specified fault hypothesis for the given task set.

A real-time fault tolerant scheduler that employs the temporal redundancy approach needs to ensure the execution of either a primary or an alternate, of all critical tasks, before their respective deadlines under the specified fault hypothesis. The existence of a real-time scheduling algorithm that can tolerate faults can be demonstrated by showing that, in any time interval, the total processor demand requested by the task primaries and the alternates that results in a worst case scenario is no greater than the size of the interval [10] [2]. Hence, for a real-time scheduling scheme to be fault tolerant, there must exist sufficient slack in the schedule for the execution of the task primaries and the alternates. EDF is known to be an optimal uniprocessor scheduling algorithm, i.e., if it is possible to schedule the original task executions together with the required alternates without causing a deadline miss, then EDF will also schedule them.

A real-time task set is said to be Fault Tolerance feasible (FT-feasible) if there exists a schedule that is capable of tolerating worst case fault occurrences under a specified fault hypothesis [2]. If the task set is not FT-feasible then there exists no sufficient slack in the schedule which can be utilized by the fault tolerance scheduling algorithm in order to recover from faults. In this case, the use of a faster processor can compensate for the slack deficit, thus enabling feasible recovery from faults. Thus the system designer can select a faster processor that guarantees the fault tolerance feasibility, but at the same time may be interested in choosing the one with the lowest speed among those eligible due to cost factors. However, the system designer has to first know if FT-feasibility can be achieved by speeding up the processor by a practicable and a reasonably low factor. Consequently it demands the knowledge of an upper-bound on the minimum processor speed-up required that can guarantee FT-feasibility. This information is interesting because 1) it provides the system designer with a quick test to check whether a processor of appropriate speed is available in his inventory and 2) it can also provide significant insights into developing a simple utilization based test for FT-feasibility.

In this paper, we examine the FT-feasibility of real-time tasks under at most a single error burst of known length occurring during the hyper-period of the

task set-which is the least common multiple of the task periods. We first derive a sufficient condition for the fault tolerance feasibility, leveraging on the optimality of EDF under uni-processor scheduling. We then derive the resource augmentation bounds, specifically the processor speed-up, required to make a real-time task set which is not fault tolerant feasible to be feasible under the error burst. We also show that, if the error burst length is no longer than half the shortest deadline of the task set, the upper-bound on the minimum processor speed-up that guarantees FT-feasibility is 6.

The rest of the section is organized as follows: section 8.2 discusses the related works and section 8.3 details the system model. In section 8.4 we formally define the problem, followed by the fault tolerance feasibility analysis in section 8.5. We present an example in section 8.7 followed by our conclusions in 8.8.

8.2 Related Work

Avizienis et. al. [1] defines dependability as the ability of a system to deliver a justifiably trusted service. They proposed the use of fault tolerance mechanisms as one of the means to achieve dependability to tackle the threat of faults, that compromise the dependability of the system. The fault tolerance strategy typically involves two stages: error detection and recovery. The recovery process can be classified as error handling and fault handling depending on the process involved in the recovery. The commonly used error handling schemes are rollback, roll forward and compensation using redundancy. The most commonly adopted redundancy technique is the temporal redundancy which involves either the re-execution of the failed software component or the execution of an alternate. In [11], the authors proposed a fault tolerant multi-processor scheduling algorithm for aperiodic tasks. A global optimization method called simulated Annealing [12] was derived from the slow cooling of molten metal to form regular crystalline structure. Attiya and Hamam [13] used Simulated Annealing to allocate tasks in a heterogenous real-time system, maximizing the reliability of the system. Bannister and Trivedi [14] proposed a simple heuristic algorithm that evenly distributes the computational load of the tasks over the nodes. More recently, Islam et.al.[15] proposed a heuristic approach to perform allocation by considering dependability and real-time constraints as well as communication efficiency.

Baruah et. al. [10] derived a sufficient and necessary condition for a set of real-time tasks to be feasible on a uni-processor. They used the optimality of

EDF to derive these conditions i.e., if the task set if EDF schedulable, then it is feasible. Here the feasibility refers to the existence of a real-time scheduling algorithm that can schedule the task set without any deadline misses. In [16], the authors presented an exact schedulability test for fault tolerant real-time task sets for the Fixed Priority Scheduling (FPS) scheme. They considered time redundancy as the fault tolerance strategy while deriving these tests. Aydin [2] considered the uni-processor fault tolerance feasibility of a real-time task set under a k-fault scenario. They presented an exact feasibility analysis for the real-time task set to be fault tolerant, leveraging on the optimality of EDF. The paper also proposed a dynamic programming technique to calculate the worst case recovery overhead for task sets scheduled using EDF. The k-fault scenario may not be a realistic model; a more realistic model might be to consider fault/error bursts e.g., single event upsets caused due to radiation when an automobile passes through the vicinity of a radiation source, rather than considering a maximum of k faults [17]. Pathan et. al. [18] extended this [2] analysis to FPS and derived a necessary and sufficient condition for the fault tolerance feasibility of real-time tasks scheduled using FPS. They assumed no more than k faults every largest relative deadline in the task set. However, as mentioned earlier, the k fault model may not be realistic as the faults normally may occur for a duration. Zhu et. al. [19] studied the effects of power management on the reliability of the system and showed that energy management techniques detrimentally affect the reliability of the system. Later, they [20] proposed reliability aware energy management techniques. The technique involves, scheduling a recovery at the maximum processor frequency before executing any task at a lower frequency. Many et. al. [4] considered the FPS schedulability of a set of real-time tasks under a fault burst and derived an equation to find the response times of tasks scheduled under the burst. Additionally they also presented a fault resilience evaluation method. Aysan et.al. [3] derived a sufficient condition to guarantee the schedulability of a task set using FPS under an error burst. They presented a probabilistic burst error model and derived probabilistic schedulability guarantees for the task set. Earlier, they [21] presented a method to maximize the schedulability of mixed criticality real-time tasks using FPS. This was achieved by exploiting the ability of EDF to achieve 100% utilization to embed primary and alternates in the schedule. They achieved this by deriving feasibility windows and then deriving fixed priorities for the tasks and their alternates [22] which was later extended to schedule mixed criticality messages on the Controller Area Network (CAN) [23], as well as to schedule tasks on a distributed real-time system under safety constraints [24].

Resource augmentation [25], is a technique used to understand how much

extra resources a scheduler requires such that it can provide a specific guarantee with respect to some constraints. Here, the scheduler under study is given extra resources such as more number of processors or faster processors, such that a certain goal is achieved. Kalyanasundaram et. al. [25] first introduced resource augmentation, in which they studied the effectiveness of online scheduling of real-time tasks showing that augmenting the processor with more speed can achieve the same effect as clairvoyance while scheduling tasks online. Davis et. al. [26] used resource augmentation to study the effectiveness of fixed priority schedulers in scheduling all the feasible task sets. They derived resource augmentation bounds on the processor speed-up required for a fixed priority scheduler to schedule all the task sets scheduled by an optimal scheduling algorithm leveraging on the optimality of the Earliest Deadline First (EDF) algorithm.

We leverage on the optimality of EDF to derive a sufficient condition for the FT-feasibility of the real-time tasks under an error burst. Our fault tolerance feasibility analysis is very much similar to [2], with the exception that we consider error bursts affecting the task executions, rather than a bounded number of task execution failures. We then examine the use of processor speed-up to guarantee the FT-feasibility of a task set under the error burst using which we derive resource augmentation bounds for FT-feasibility.

8.3 System Model

In this section, we describe the system model and the notations used in this paper.

8.3.1 Task model

We consider a set of sporadic real-time tasks Γ= $\{\tau_1, \tau_2, ...\tau_n\}$, where each τ_i has a minimum inter-arrival time, T_i, a worst case execution time, C_i^S at speed S, and a relative deadline, D_i. We assume that the tasks are ordered according to their increasing deadlines. Each of these tasks generate a potentially infinite sequence of jobs, where the j^{th} job of the i^{th} task is denoted by $\tau_{i,j}$. A job $\tau_{i,j}$ is released at time $(j-1)T_i$ and has to complete its execution no later than $(j-1)T_i + D_i$ in order to meet its deadline. Additionally, let $\{d_1, d_2, ..., d_m\}$ denote the set of absolute deadlines of the task set in the LCM, ordered in the increasing order i.e,. $\forall \tau_i \in \Gamma$, $d_i < d_{i+1}$, where LCM represents the Least Common Multiple of the time periods of the tasks.

The utilization U_i of a task τ_i executing on a processor at speed S is defined as $U_i^S = \frac{C_i^S}{T_i}$, and the utilization of the entire task set is given by $U^S = \sum_{i=1}^{n} U_i^S$. The demand bound function [10] of a task τ_i, on a processor of speed 1, during an interval t is given by:

$$DBF_i(t) = max\left(0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i^1$$

8.3.2 Scheduling Model and Fault Tolerance Strategy

It is known that EDF is optimal under a work conserving uniprocessor scheduling scheme, i.e., a work conserving EDF can schedule all task sets which are schedulable by any other work conserving scheduler [8]. Thus, if a valid schedule exists for a particular task set, then EDF can feasibly schedule it. We leverage the optimality of EDF to study the fault tolerance feasibility of real-time tasks on a uni-processor.

Most of the previous works treat an error as a singleton event. In this paper, we consider an error burst which is a series of errors occurring within a specific time interval that makes it impossible to perform any meaningful task executions during that interval. We assume a known upper-bound on the length of the error burst during the LCM denoted by T_{length}. We assume that all the task executions during the error burst fails, and the failure detection happens at the end of the task execution, before its completion. The employed fault tolerance strategy is the re-execution of the failed task or the execution of an alternate task before the original deadline. The fault tolerance strategy assumes that the alternates have the same deadline as the original task (the primary) and they are executed along with the rest of the tasks according to EDF. Consequently, the alternates can also be hit by the error burst, and the alternates are scheduled until one successful execution of the task is achieved. The WCET of the alternates is assumed to be no greater than the WCET of the original task.

8.3.3 Execution Time Model

In our approach we assume a linear relationship between execution time and processor speed [26] [25]. To ease the readability, and without loss of generality, we assume that the task set is initially executing on a processor of speed

$S = 1$. Hence, if C_i^1 is the execution time at speed $S = 1$, for any $S > 1$:

$$C_i^S = \frac{C_i^1}{S}$$

Thus the speed required to obtain an execution time of C_i^S is given by:

$$S = \frac{C_i^1}{C_i^S}$$

This model also allows the use of processor speed-up factors and processor speeds interchangeably. Changing the processor speed from $S = 1$ to $S = a$, is equivalent to speeding up the processor by a factor of 'a'. We also assume that the number of clock ticks required to execute a task τ_i is equal to the execution time of τ_i at speed $S = 1$. Hence, $DBF_i(t)$ denotes the number of clock ticks requested in the time interval t on a processor of speed $S = 1$. Consequently, when a processor of speed 'a' is used, the total time requested by the tasks during the time interval t becomes $\frac{DBF_i(t)}{a}$.

8.4 Problem Description

In this paper, we address the following questions:

1 How to determine the FT-feasibility of a given set of temporally redundant real-time tasks under an error burst of known upper-bounded length during LCM?

A followup question is:

2 If the real-time task set is not found to be FT-feasible, what is the lowest processor speed-up that guarantees its FT-feasibility under the error burst?

8.5 Fault Tolerance Feasibility Analysis

In this section, we present the proposed fault tolerance feasibility analysis and derive the processor speed-up required to guarantee the FT-feasibility of a real-time task set, under an error burst.

Due to the error detection mechanism assumed to be performed at the end of the tasks' executions, in the analysis we account for the WCET of the primary and alternate tasks under the error burst. If the error burst starts just

before any job of τ_i finishes its execution, the rest of the execution of τ_i is outside the influence of the error burst. We define the execution of τ_i that occurs outside the error burst as the maximum wasted execution time of τ_i.

Definition 4. *The Maximum Wasted Execution Time (MWET) of a task τ_i hit by an error burst, is defined as the execution time of the primary or an alternate of τ_i which lies outside the error burst, that leads to the largest wastage of the processor utilization.*

An example of the maximum wasted execution time of a task is shown in figure 8.1. In any time interval t, the error burst can hit multiple tasks ($\tau_i's$)

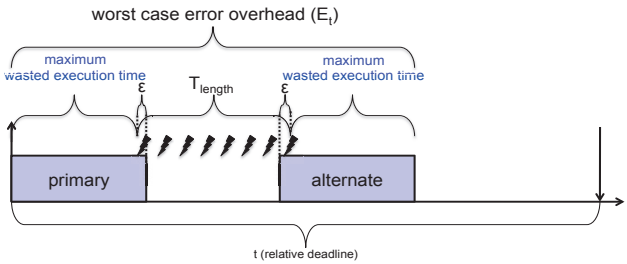


Figure 8.1: The worst case error overhead due to error bursts on a single task

leading to many such maximum wasted execution times (MWET) that wastes the processor time. The worst case sum of all the possible maximum wasted execution times of all the tasks until t gives the worst case temporal wastage (WCTW) in the interval t, that leads to the largest overhead outside the error burst. This is formally defined in the following definition.

Definition 5. *The Worst Case Temporal Wastage (WCTW) during a time interval t, denoted by $W_{err}(t)$, is defined as the largest possible temporal overhead which lies outside the error burst, that occurs due to the execution of the MWETs of all the failed primaries and alternates in the interval t, that have their releases and deadlines within t.*

We now identify a necessary condition for FT-feasibility of the set of real-time tasks.

Lemma 8.5.1. *A necessary condition for the FT-feasibility of a task set Γ is,*

$$T_{length} \leq \min_{\forall \tau_i \in \Gamma} (D_i - 2C_i) + \epsilon$$

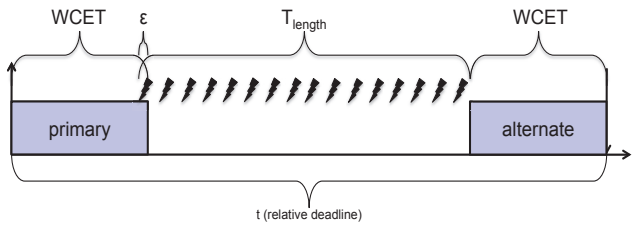


Figure 8.2: The maximum length of the burst error.

Proof. The proof for this lemma can be easily seen from figure 8.2. If the burst length is greater than $\min(D_i - 2C_i + \epsilon)$ where $\forall \tau_i \in \Gamma$, for any task τ_j with a deadline t, it is impossible to guarantee a successful execution τ_j before the deadline t. □

Some assumptions : In the rest of the section, we consider any time instant t' when a job $\tau_{i,j}$ is executing on the processor. Unless stated otherwise, we assume that the job $\tau_{i,j}$ is the first job to be hit by the error burst and the error burst starts at the time instant t'. We also consider a time instant t which is the absolute deadline of $\tau_{i,j}$, $t > t'$. The worst case temporal wastage occur when all the jobs arrive in a strictly periodic manner. Our strategy of finding the FT-feasibility is as follows- we assume that even under the error burst, there are no deadline misses in the schedule, and then we derive the sufficient condition for this to be true.

Lemma 8.5.2. *If no task is released at or after time t', that has an absolute deadline less than or equal to t, the worst case temporal wastage $W_{err}(t)$ at t is given by:*

$$W_{err}(t) = 2(C_i - \epsilon)$$

Proof. According to our assumption, every job of a task released between time t' and t has an absolute deadline greater than the deadline of $\tau_{i,j}$. The job $\tau_{i,j}$ has to finish its execution for any other job to start its execution. Thus, $\tau_{i,j}$ is the only job that is hit by the error burst. This is because, every job present in the ready queue and every job released after time t', has an absolute deadline later than the absolute deadline of $\tau_{i,j}$, and will execute only after $\tau_{i,j}$ completes its execution successfully, since we assume an EDF scheduler. Thus $W_{err}(t)$ is given in the scenario when the error burst starts just before the

primary of $\tau_{i,j}$ completes its execution and just after the last failed alternate of $\tau_{i,j}$ starts its execution (see figure 8.1). Hence, in this case:

$$W_{err}(t) = 2(C_i - \epsilon)$$

The WCTW at time t is thus equal to twice the MWET of τ_i. □

Observation 8.5.1. *Every task that is released at or after time t', having an absolute deadline less than or equal to t, will have a relative deadline less than or equal to D_i.*

This is quite straight forward as $\tau_{i,j}$ has been released at a time instant less than or equal to t'. Thus every task that is released after t' having an absolute deadline less than or equal to t must have a relative deadline less than the relative deadline of τ_i.

All the jobs that are released in the interval $[t', t]$, having a later deadline than t will not be hit by the error burst. This is proved in the following lemma.

Lemma 8.5.3. *No job $\tau_{a,b}$ released in the interval $[t', t]$, having an absolute deadline $bT_a + D_a > t$, can be hit by the error burst.*

Proof. The job $\tau_{a,b}$ will be scheduled only after $\tau_{i,j}$ has completed one successful execution since $\tau_{i,j}$ has the earliest deadline. According to our assumption, the task set is schedulable even under the error burst. Thus, the error burst would have ended before $\tau_{a,b}$ started its execution, since $\tau_{i,j}$ completed one successful execution. □

We now show that the WCTW at the absolute deadlines of jobs released in the interval $[t,' t]$, having a later deadline than t, is equal to the WCTW at time instant t.

Lemma 8.5.4. *The $W_{err}(d_l)$ for any job $\tau_{a,b}$ that is released in the interval $[t', t]$, having an absolute deadline denoted by $d_l = bT_a + D_a > t$, is given by:*

$$W_{err}(d_l) = W_{err}(d_{l-1})$$

Proof. When $\tau_{a,b}$ starts its execution, the value of $W_{err}(d_l)$ is equal to the value of $W_{err}(t)$, since no job with a deadline greater than t is hit by the error burst (consequently no 'new' alternates are executed). Thus, in general we can say that $W_{err}(d_l) = W_{err}(d_{l-1})$ for such a job $\tau_{a,b}$, as the same argument holds for every such job having an earlier absolute deadline than d_l. □

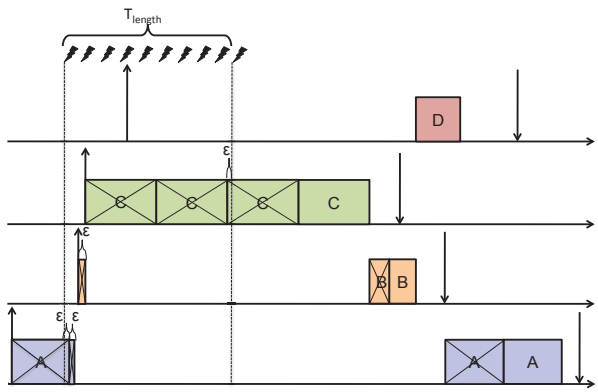


Figure 8.3: Error burst hitting multiple jobs

In the next lemma, we bound the contribution of $\tau_{i,j}$ to the WCTW at t when more than one task is hit by the error burst in the interval $[t', t]$.

Lemma 8.5.5. *If the error burst hits more than one task in the interval* $[t', t]$*, the contribution of* $\tau_{i,j}$ *to* $W_{err}(t)$ *at time* t *is* $2(C_i - \epsilon)$.

Proof. In this case, the primary of $\tau_{i,j}$ is hit ϵ units before it completes its execution, and one of its failed alternates is preempted immediately (ϵ time units) after it starts its execution. Assume that $\tau_{a,b}$ is the task preempting $\tau_{i,j}$, which means that $\tau_{a,b}$ has an earlier absolute deadline than $\tau_{i,j}$. Using the similar argument from lemma 8.5.3, the error burst will end before $\tau_{a,b}$ completes one successful execution. When $\tau_{i,j}$ resumes its execution, its remaining execution, i.e., $C_i - \epsilon$ is wasted as it was hit by the error burst just before it was preempted. The alternate of job $\tau_{i,j}$ then executes successfully as the error burst has already ended. Thus the contribution of $\tau_{i,j}$ to to $W_{err}(t)$ at time t is $2(C_i - \epsilon)$. □

Only either the primary or one of the alternates of the jobs hit by the error burst in the interval $[t', t]$ contributes to the WCTW at time t. While a proof of it under FPS has been presented in [17], in this paper, we extend it to EDF in the following lemma since the assumptions under FPS are no longer valid under EDF.

Lemma 8.5.6. *If the error burst hits more than one task in the interval $[t', t]$, only either the primary, or exactly one alternate of each task, other than τ_i, that is hit by the error burst will contribute to $W_{err}(t)$ at time t.*

Proof. We consider only the jobs that are executing in the interval $[t', t]$. This is because only these jobs have to finish their execution before their corresponding absolute deadlines, so that $\tau_{i,j}$ can finish its execution no later than time instant t. Any job having an absolute deadline greater than t will not affect the execution of $\tau_{i,j}$.

According to our assumption, $\tau_{i,j}$ is the first job to be hit by the error burst. Let the job $\tau_{a,b}$ that has a release time and deadline in the interval $[t', t]$, be the next task to be hit by the error burst. For the task set to be schedulable, $\tau_{a,b}$ needs to recover before its absolute deadline i.e., it must have one successful execution before its absolute deadline. Additionally, there should not be any deadline misses in the rest of the schedule until the LCM. The execution of the job $\tau_{a,b}$ is under the error burst and its contribution to $W_{err}(t)$ is maximum when either:

1. The primary or one of the failed alternates of job $\tau_{a,b}$ is immediately preempted by a higher priority job $\tau_{e,f}$ as soon as it starts execution.

   In this case, the error burst will end before the job $\tau_{e,f}$ completes one successful execution, after which the remaining executions of the failed primary or alternate of $\tau_{a,b}$, which was preempted, execute to completion. Thus the maximum processor time wasted by $\tau_{a,b}$ is $(C_a - \epsilon)$, before it can successfully execute, according to the definition 4.

2. The error burst ends just before the last failed alternate of $\tau_{a,b}$ starts executing.

   This is the case when $\tau_{a,b}$ is the only job other than $\tau_{i,j}$ that is hit by the error burst. In this case, the contribution of $\tau_{a,b}$ to $W_{err}(t)$ is maximum when the error burst ends just after the start of an alternate of $\tau_{a,b}$. Hence, according to definition 4, the maximum processor time wasted is $(C_a - \epsilon)$, before $\tau_{a,b}$ successfully executes.

In both cases, maximum execution of $\tau_{a,b}$ that can lie outside the region of the error burst is $(C_a - \epsilon)$. The above argument can be repeated for all higher priority jobs $\tau_{e,f}$ that are released between the release time of $\tau_{a,b}$ and its absolute deadline.

Thus we can see that, if the error burst hits more than one job, either only the primary or exactly one alternate of each task other than $\tau_{i,j}$, that is hit by the error burst, will contribute to $W_{err}(t)$ at time t. □

We have thus bounded the contributions of the jobs scheduled in the interval $[t', t]$ to the $W_{err}(t)$ at t, when the error burst hits more than one job. An example, when the error burst hits multiple jobs, is given in figure 8.3. We now derive the $W_{err}(t)$ when the error burst hits more than one job in the interval $[t', t]$, in the general case.

Lemma 8.5.7. *If the error burst hits more than one job in the interval $[t', t]$, the worst case temporal wastage $W_{err}(t)$ is given by:*

$$W_{err}(t) = 2(C_i - \epsilon) + \sum_{\forall \tau_k \in \Gamma : D_k \leq D_i} (C_k - \epsilon)$$

Proof. We know from lemma 8.5.6 that only either the primary or one of the alternate of the failed tasks that have release times and deadlines in $[t', t]$ will contribute to the worst case temporal wastage at t. Thus the total contribution to $W_{err}(t)$ is the maximum when every job $\tau_{a,b}$ released in the interval $[t', t]$, that has a deadline no later than t, is hit by an error burst and leaves $C_a - \epsilon$ time units of execution outside the error burst. This scenario occurs when the tasks released in the interval $[t', t]$ preempt each other in a nested manner, with every preemption occurring ϵ units after the start of the execution of the preempted task.

The tasks that may be potentially released in the interval $[t', t]$ are the tasks that have relative deadlines less than or equal to D_i (observation 8.5.1). Thus following the reasoning in lemma 8.5.6, only either the primary or exactly one alternate of each of the failed tasks other than $\tau_{i,j}$ will contribute to the worst case temporal wastage. The worst case contribution of $\tau_{i,j}$ is $2(C_i - \epsilon)$ according to lemma 8.5.5.

Thus, the worst case temporal wastage at time t is equal to,

$$W_{err}(t) = 2(C_i - \epsilon) + \sum_{\forall \tau_k \in \Gamma : D_k \leq D_i} (C_k - \epsilon)$$

We thus obtain the WCTW at t, when the error burst hits multiple jobs. □

Let us now consider the case when the error burst hits only a single job and $\tau_{i,j}$ is not necessarily the job to be hit. This means that any task in the interval $[t', t]$ could be hit by the error burst and the WCTW at t is given by the following lemma.

Lemma 8.5.8. *If the error burst hits only a single job, not necessarily $\tau_{i,j}$, in the time interval $[t', t]$, the worst case temporal wastage at time t is given by:*

$$W_{err}(t) = \max_{\forall \tau_k \in \Gamma : D_k \leq D_i} \{2(C_k - \epsilon), 2(C_i - \epsilon)\}$$

Proof. According to the observation 8.5.1, all the jobs that are completely scheduled in the interval $[t', t]$ are the jobs of the tasks with a relative deadline less than or equal to the relative deadline of τ_i. Thus, if only one job is hit by the error burst in the interval $[t', t]$, the maximum contribution to the worst case temporal wastage at t is twice the maximum of the worst case execution time wastage (w_k) of τ_k, if a job of τ_k is scheduled in the interval $[t', t]$. This is the case when the error burst starts just before the primary of the task hit by the burst finishes its execution and ends just after the last failed alternate has started its execution, as shown in lemma 8.5.2. Here, τ_k can be either τ_i or, according to observation 8.5.1 and using lemma 8.5.3, any τ_k such that $D_k \leq D_i$. Hence,

$$W_{err}(t) = \max_{\forall \tau_k \in \Gamma : D_k \leq D_i} \{2(C_k - \epsilon), 2(C_i - \epsilon)\}$$

We have thus derived the worst case temporal wastage for the final scenario. □

We now propose one of our main theorems which bounds the WCTW at t, which we later use to reason about the FT-feasibility.

Theorem 8.5.1. *The worst case temporal wastage $W_{err}(t)$ at any time instant t, where $t = d_l = jT_i + D_i$ for any job $\tau_{i,j}$, is given by:*

$$W_{err}(t) = \max(x, y, W_{err}(d_{l-1}))$$

Here,

$$x = \max_{\forall \tau_k \in \Gamma : D_k \leq D_i} \{2(C_k - \epsilon)\}$$

$$y = 2(C_i - \epsilon) + \sum_{\forall \tau_k \in \Gamma : D_k \leq D_i} (C_k - \epsilon)$$

Proof. The proof follows from lemma 8.5.4, 8.5.7, 8.5.8. At deadline d_l, according to lemma 8.5.7, if the error burst hits more tasks in addition to $\tau_{i,j}$,

$$W_{err}(t) = 2(C_i - \epsilon) + \sum_{\forall \tau_k \in \Gamma : D_k \leq D_i} (C_k - \epsilon)$$

According to lemma 8.5.8, at deadline d_l, if the error burst hits only one task, the $W_{err}(t)$ is given by the maximum of the MWETs of the tasks scheduled until d_l, thus,

$$W_{err}(t) = \max_{\forall \tau_k \in \Gamma : D_k \leq D_i} \{2(C_k - \epsilon), 2(C_i - \epsilon)\}$$

Finally, according to lemma 8.5.4, if $\tau_{i,j}$ is a job that has an absolute deadline greater than the deadline of the job first hit by the error burst, then,

$$W_{err}(d_l) = W_{err}(d_{l-1})$$

Hence, $W_{err}(t)$, where $t = jT_i + D_i$ for any $\tau_{i,j}$ is given by the maximum of the $W_{err}(t)$ given by lemmas 8.5.4, 8.5.7, 8.5.8. □

We now define the worst case error overhead at any time t which is the worst case overhead involved in tolerating faults.

Definition 6. *The worst case error overhead E_t, in any time interval t, is defined as the sum of the error burst length T_{length} and the worst case temporal overhead in the time interval t.*

$$E_t = T_{length} + W_{err}(t)$$

An example of the worst case error overhead at a time instant t i.e., E_t is shown in figure 8.1. We now build on the demand bound analysis proposed by Baruah et. al [10] and define the sufficient condition for FT-feasibility under an error burst.

Theorem 8.5.2. *A real-time task set Γ is FT-feasible under an error burst of length T_{length} if, $\forall t = kT_j + D_j, \forall \tau_j \in \Gamma$ and $t \leq LCM$,*

$$E_t + \sum_{i=1}^{n} DBF_i(t) \leq t$$

Sketch. When the above condition is satisfied, there is sufficient slack in the schedule, during any time interval t, for the execution of the real-time tasks and the alternates of the failed tasks, outside the region of the error burst.

Suppose that the above condition is not satisfied for some t, i.e.,

$$W_{err}(t) + \sum_{i=1}^{n} DBF_i(t) > t - T_{length}$$

This means that during some time interval, the total execution demanded by the task set exceeds the size of that interval and hence the task set is not feasible. The formal proof is similar to the proof presented in [2]. □

However, depending on the real-time schedule, the actual maximum temporal wastage at t may or may not be equal to the worst case. Hence if the lemma is not satisfied, no guarantees can be given about the FT-feasibility using the above feasibility test. Thus the above theorem is only a sufficient test for FT-feasibility.

8.6 Resource Augmentation for FT-Feasibility

In this section, we examine the resource augmentation bounds that guarantees the FT-feasibility of a set of real-time tasks under a known error burst length. We first, in the following theorem, derive the exact processor speed-up that guarantees the FT-feasibility of the real-time task set.

Theorem 8.6.1. *The minimum processor speed-up required to guarantee the FT-feasibility of a real-time task set* Γ *under a burst error of length* T_{length} *is given by:*

$$S = \max_{\forall t} \left\{ \frac{W_{err}(t) + \sum_{i=1}^{n} DBF_i(t)}{t - T_{length}} \right\}$$

Proof. If any given task set Γ is not FT-feasible on a processor of speed $S = 1$, there exists a time instant t such that,

$$T_{length} + W_{err}(t) + \sum_{i=1}^{n} DBF_i(t) > t$$

Suppose that speeding up the processor by a factor of S will ensure its FT-feasibility. We get,

$$T_{length} + \frac{W_{err}(t)}{S} + \frac{\sum_{i=1}^{n} DBF_i(t)}{S} \leq t$$

Thus, $\forall t$,

$$\frac{W_{err}(t) + \sum_{i=1}^{n} DBF_i(t)}{S} \leq t - T_{length}$$

Solving for S we get $\forall t$,

$$S \geq \frac{W_{err}(t) + \sum_{i=1}^{n} DBF_i(t)}{t - T_{length}}$$

Hence,

$$S = \max_{\forall t \in aT_j + D_j, t \leq LCM} \left\{ \frac{W_{err}(t) + \sum_{i=1}^{n} DBF_i(t)}{t - T_{length}} \right\}$$

We thus obtain the minimum processor speed-up required to guarantee FT-feasibility. □

In order to derive upper-bounds on the processor speed-up that guarantees FT-feasibility, we bound the $W_{err}(t)$ at any time instant t.

Lemma 8.6.1. *The worst case temporal wastage $W_{err}(t)$, $t \in \{d_1, d_2, ..., d_m\}$, is upper-bounded by:*

$$W_{err}(t) \leq 2 \sum_{i=1}^{n} DBF_i(t)$$

Proof. At deadline d_1, which is the shortest relative deadline D_1, the worst case temporal wastage $W_{err}(t)$, according to theorem 8.5.1, is given by:

$$W_{err}(t) = max(x, y)$$

$$x = \max_{\forall \tau_k \in \Gamma : D_k \leq D_1} \{2(C_k - \epsilon)\}$$

$$y = 2(C_1 - \epsilon) + \sum_{\forall \tau_k \in \Gamma : D_k \leq D_1} (C_k - \epsilon) = 2(C_1 - \epsilon)$$

Here, clearly x or y can be upper-bounded by:

$$x \leq 2 \sum_{i=1}^{n} DBF_i(D_1) \; and \; y \leq 2 \sum_{i=1}^{n} DBF_i(D_1)$$

Consider any absolute deadline d_l of any task τ_i, $d_l = jT_i + D_i$. The $W_{err}(t)$ is given by:

$$W_{err}(t) \leq max(x, y, d_{l-1})$$

Here again, we can see that x and y can be bounded by:

$$x \leq 2 \sum_{i=1}^{n} DBF_i(d_l) \; and \; y \leq 2 \sum_{i=1}^{n} DBF_i(d_l)$$

Hence for any t,

$$W_{err}(t) \leq 2 \sum_{i=1}^{n} DBF_i(t)$$

This gives an upper-bound on the worst case temporal wastage in any time interval t. □

Using the above bounds on the $W_{err}(t)$, we derive an upper-bound on the processor speed-up that guarantees FT-feasibility in the following theorem.

Theorem 8.6.2. *The minimum processor speed-up S_b that guarantees the FT-feasibility of a set of real-time tasks Γ under an error burst of length T_{length} is upper-bounded by:*

$$S_b \leq \frac{3y}{y-1}$$

where $y = \frac{t}{T_{length}}$, $t \in \{d_1, d_2, ..., d_m\}$.

Proof. According to lemma 8.6.1, the upper-bound on the $W_{err}(t), t \in \{d_1, d_2, ..., d_m\}$ is given by:

$$W_{err}(t) \leq 2\sum_{i=1}^{n} DBF_i(t)$$

According to theorem 8.6.1,

$$S = \max_{\forall t} \left\{ \frac{W_{err}(t) + \sum_{i=1}^{n} DBF_i(t)}{t - T_{length}} \right\}$$

Substituting the upper-bounds on $W_{err}(t)$, we get, $\forall t \in \{d_1, d_2, ..., d_m\}$,

$$S \leq \frac{3\sum_{i=1}^{n} DBF_i(t)}{t - T_{length}}$$

Since we assume that the original task set is schedulable, $\forall t \in \{d_1, d_2, ..., d_m\}$,

$$\sum_{i=1}^{n} DBF_i(t) \leq t$$

Substituting for $\sum_{i=1}^{n} DBF_i(t)$, $\forall t \in \{d_1, d_2, ..., d_m\}$, we get the upper-bound on the required speed-up denoted by S_b,

$$S_b \leq \frac{3t}{t - T_{length}}$$

Thus,

$$S_b \leq \frac{3y}{y-1}$$

where $y = \frac{t}{T_{length}}$, $t \in \{d_1, d_2, ..., d_m\}$.

The largest value of S_b is obtained at $d_1 = D_1$, the shortest relative deadline of the task set. □

Task	C_i	D_i	T_i
A	1	5	6
B	1	9	9
C	2	18	18

Table 8.1: Example task set

We now derive the resource augmentation bounds for the case when the error burst length is no longer than half the shortest deadline.

Theorem 8.6.3. *The upper-bound on the minimum processor speed-up S_b that guarantees the FT-feasibility of a set of real-time tasks Γ under an error burst of length T_{length} such that for any time interval $t \in \{d_1, d_2, ..., d_m\}$, $y \geq 2$, $y = \frac{t}{T_{length}}$, is given by:*

$$S_b \leq 6$$

Proof. This is straight away obtained from theorem 8.6.2, by evaluating the limits at $y = 2$ and $y = \infty$. □

We have thus presented a sufficient condition for the fault tolerance feasibility of a task set under an error burst, and derived upper-bounds on the processor speed-up required to guarantee the fault tolerance feasibility, if the sufficient condition fails for some task set. We have proved that if the error burst length is no longer than half the shortest deadline of the task set, the resource augmentation bound that guarantees the FT-feasibility is 6.

8.7 Example

We illustrate our feasibility analysis and resource augmentation bounds using a simple example. Consider a real-time task set as shown in table 8.1 with 3 tasks. To illustrate the use of processor speed-up to enable FT-feasibility, let us assume that the error burst length $T_{length} = 4$. The demand bound until the first absolute deadline 5 (demanded by task A) is equal to:

$$\sum_{i=A}^{C} DBF_i(5) = 1$$

Suppose the primary of task A is hit by the error burst, the maximum time is *wasted* when the burst hits the primary just before it finishes its execution. At

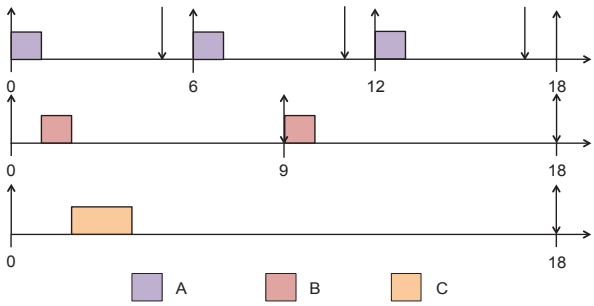


Figure 8.4: EDF schedule

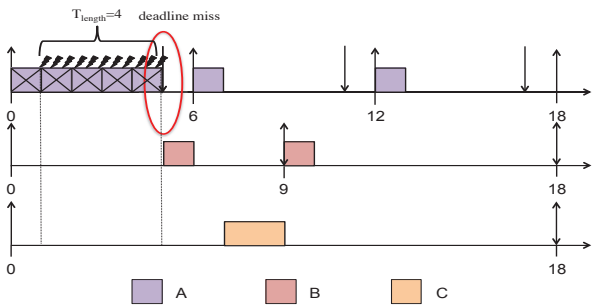


Figure 8.5: EDF schedule under faults with $T_{length} = 4$

time instant $t = 1$, the alternate of task A starts its execution and this is again hit by the burst. At $t = 2$, the alternate is again executed, which is again hit by the error burst. Alternates continue to execute and at time instant $t = 4.9$, during the execution of one of the alternates, the error burst ends. It can be easily seen that task A does not have sufficient slack outside the error burst to complete one successful execution since it has a deadline at $t = 5$. One of the fault scenarios where $T_{length} = 4$ is illustrated in figure 8.5, and there is a deadline miss on task A. Formally,

$$E_5 + \sum_{i=A}^{C} DBF_i(5) = 5.8 + 1 = 6.8 > 5$$

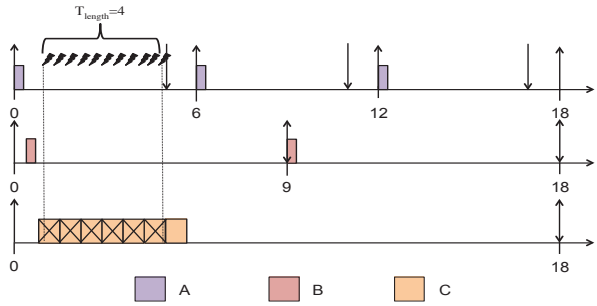


Figure 8.6: EDF schedule under faults after a speed-up of 2.8

Similarly at deadline $t = 9$, the demand bound = 2. Here the worst possible overheads due to the error burst can be bounded by $E_9 = 6.7$. The worst case temporal wastage (WCTW) occurs when the primary of task B is hit leading to a scenario as in the previous deadline. Additionally, we add one failed alternate from the higher priority task to account for the cases where higher priority tasks preempt the primary or one of the alternates of the task B under consideration. Hence,

$$E_9 + \sum_{i=A}^{C} DBF_i(9) = 6.7 + 2 = 8.7 < 9$$

Similarly, we calculate the processor demand bounds at all the absolute deadlines.

$$E_{11} + \sum_{i=A}^{C} DBF_i(11) = 6.7 + 3 = 9.7 < 12$$

$$E_{17} + \sum_{i=A}^{C} DBF_i(17) = 6.7 + 4 = 10.7 < 17$$

$$E_{18} + \sum_{i=A}^{C} DBF_i(18) = 9.6 + 6 = 15.6 < 18$$

Thus, the only possibility of a deadline miss due to the error burst is at time $t = 5$. The speed-up required to guarantee FT-feasibility is,

$$S = max\left(\frac{2.8}{1}, \frac{4.7}{5}, \frac{5.7}{8}, \frac{6.7}{13}, \frac{11.7}{14}\right) = \frac{2.8}{1} = 2.8$$

When we increase the processor speed to 2.8, during the time interval $[0, 5]$, the total value of $W_{err}(t) + \sum_{i=1}^{n} DBF_A^C(t) = \frac{1.8+1}{2.8} = 1$. Hence,

$$E_5 + \sum_{i=1}^{n} DBF_i(5) = 1 + 4 = 5$$

The same scenario in figure 8.5 on a processor that is 2.8 times faster is given in figure 8.6. Observe that there is no deadline miss on task A in the schedule in figure 8.6 under the error burst, after the speed-up. Thus, we can prevent a deadline miss at $t = 5$ by using a processor that is 2.8 times faster.

8.8 Conclusions

In this paper, we have examined the use of resource augmentation to guarantee the fault tolerance feasibility of a set of real-time tasks under an error burst. In this context, we derive a sufficient condition under the assumption of no more than a single error burst occurrence during the hyper-period of the tasks. For the cases where the sufficient condition fails, we also derive the necessary speed-up that guarantees the fault tolerance feasibility under the burst. We show that, if the length of the error burst is no more than half the shortest deadline of the task set, the processor speed-up that guarantees the fault tolerance feasibility is upper-bounded by 6.

The proposed method adds a new capability in the system designer's repertoire for analyzing the fault tolerance feasibility of a given set of real-time tasks under an error burst and derive essential resource augmentation requirements. We intend to extend the proposed approach to more severe error scenarios as well as to distributed systems in future.

Bibliography

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Securr Computing*, January 2004.

[2] H. Aydin. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Transactions on Computers*, October 2007.

[3] Hüseyin Aysan, Radu Dobrin, Sasikumar Punnekkat, and Rolf Johansson. Probabilistic schedulability guarantees for dependable real-time systems under error bursts. In *The 8th IEEE International Conference on Embedded Software and Systems*, November 2011.

[4] Florian Many and David Doose. Scheduling analysis under fault bursts. In *The 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[5] Bjrn Lisper. Trends in timing analysis. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems*. Springer Boston, 2006.

[6] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. In *Real-Time Systems Journal*, January 2005.

[7] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *The Journal of ACM*, 1973.

[8] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, 1974.

[9] Michele Cirinei, Antonio Mancina, Davide Cantini, Paolo Gai, and Luigi Palopoli. An educational open source real-time kernel for small embedded control systems. In *Computer and Information Sciences*. Springer Berlin / Heidelberg, 2004.

[10] Sanjoy K. Baruah, Louis E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 1990.

[11] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Prarallel and Distributed Systems*, March 1997.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, May 1983.

[13] G. Attiya and Y. Hamam. Task allocation for maximizing reliability of distributed systems: A simulated annealing approach. *Journal of Parallel and Distributed Computing*, October 2006.

[14] J. A. Bannister and K. S. Trivedi. Task Allocation in Fault-Tolerant Distributed Systems. *Acta Informatica, Springer-Verlag*, 1983.

[15] S. Islam, R. Lindstrom, and Neeraj Suri. Dependability driven integration of mixed criticality SW components. *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, April 2006.

[16] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *The 8th Euromicro Workshop on Real-Time Systems*, June 1996.

[17] Huseyin Aysan. Fault-tolerance strategies and probabilistic guarantees for real-time systems. In *PhD thesis, Malardalen University*, June 2012.

[18] R.M. Pathan and J. Jonsson. Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks. In *The16th International Conference on Embedded and Real-Time Computing Systems and Applications*, April 2010.

[19] Dakai Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *Proceedings of*

the 2004 IEEE/ACM International conference on Computer-aided design, November 2004.

[20] Dakai Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. In *In Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2006.

[21] Radu Dobrin, Hüseyin Aysan, and Sasikumar Punnekkat. Maximizing the fault tolerance capability of fixed priority schedules. In *The 14th IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications*, August 2008.

[22] Radu Dobrin, Gerhard Fohler, and Peter Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. In *Real-Time Systems Symposium*, December 2001.

[23] Hüseyin Aysan, Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Efficient fault tolerant scheduling on controller area network (CAN). In *15th International Conference on Emerging Technologies and Factory Automation*, September 2010.

[24] Abhilash Thekkilakattil, Hüseyin Aysan, and Sasikumar Punnekkat. Towards a contract-based fault-tolerant scheduling framework for distributed real-time systems. In *The 1st International Workshop on Dependable and Secure Industrial and Embedded Systems*, June 2011.

[25] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 2000.

[26] Robert Davis, Thomas Rothvo, Sanjoy Baruah, and Alan Burns. Exact quantification of the sub-optimality of uniprocessor fixed priority preemptive scheduling. *Real-Time Systems*, 2009.

Populärvetenskaplig svensk sammanfattning

Datorsystem används idag i många enheter som vi använder dagligen, t.ex., låsningsfria bromsar (ABS) i en bil. En stor del av dessa datasystem kallas för realtidssystem, där systemet måste utföra en mängd uppgifter inom en fördefinierad tid. Schemaläggningsalgoritmer används för att bestämma när dessa uppgifter utförs på en viss processor, för att, i slutändan, garantera deras slutförande före en fördefinierad tid, s.k. deadline. I de flesta fall måste uppgifterna schemaläggs omlott, där de avbryter varandra, för att på ett bättre sätt använda systemets resurser. Dessa avbrott, å andra sidan, genererar kostnader som kan, i värsta fall, leda till en situation där en särskilld uppgift inte slutförs före sin deadline, som i sin tur kan leda till att ABS-systemet inte fungerar som det ska. Därför är det avgörande att ha kontroll över dessa omkostnader för att systemet ska fungera korrekt.

Vissa datorstyrda system är säkerhetskritiska, vilket innebär att katastrofala konsekvenser kan inträffa om en enstaka uppgift missar sin deadline. Dessutom används dessa system i en miljö där de utsätts för fel, till exempel bilens elektronik som tidsvis utsätts för elektromagnetisk interferens. Därför måste datorsystemen vara feltoleranta, vilket innebär att de måste slutföra alla uppgifter före deras deadlines eller ett eller flera fel uppstår.

Moderna datorsystem ger designern möjligheten att snabba upp processor för att åstadkomma en snabbare exekvering av vissa uppgifter. Detta medför en kostnad i form av ökad energiförbrukning. I denna avhandling visar vi på möjligheten att använda processorer med variabla hastigheter för att hantera avbrottsrelaterade kostnader i realtidssytem för att optimera systemets prestanda, samt öka systemets feltolerans.