FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Real-time Limited Preemptive Scheduling

José Manuel Silva dos Santos Marinho

Doutoramento em Engenharia Electrotécnica e de Computadores

Orientador: Stefan Markus Ernst Petters (Dr.)

2015

Real-time Limited Preemptive Scheduling

José Manuel Silva dos Santos Marinho

Doutoramento em Engenharia Electrotécnica e de Computadores

2015

This thesis was partially suported by FCT (Portuguese Foundation for Science and Technology) under PhD grant SFRH/BD/81085/2011

POPH
QUALIFICAR É CRESCER.

QREN
QUADRO
DE REFERÊNCIA
ESTRATÉGICO
NACIONAL
PORTUGAL 2007.2013

GOVERNO DE
PORTUGAL

UNIÃO EUROPEIA
Fundo Social Europeu

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

O desenho de sistemas reativos de alto desempenho depende do controlo preciso de variáveis impactantes no comportamento do sistema. As entidades responsáveis por este controlo tomam, comummente, corpo num dispositivo computacional digital. As propriedades dinâmicas dos fenómenos sob controlo, em conjunto com os requisitos de comportamento pretendido pelo projetista, impõem restrições ao máximo espaçamento temporal entre a observação de um evento e a correspondente atuação.

A área de sistemas de tempo-real providencia uma base teórica sobre a qual propriedades temporais do sistema são estudadas, possibilitando a derivação de garantias mínimas de desempenho.

As análises providenciadas são por definição seguras, i.e. , qualquer garantia dada sobre a obtenção das metas temporais será verificada, sob qualquer pretexto concebível, durante a utilização do sistema. Uma análise segura tem geralmente associada a si um nível de pessimismo. Pessimismo, em termos gerais, é uma medida da dificuldade de caracterização do pior caso possível verificável durante a utilização do sistema, por mais improvável que seja a ocorrência desse pior caso. É somente possível reduzir o pessimismo das análises através do uso de estratégias de escalonamento que se caracterizem por uma maior e melhor especificação das circunstâncias observadas durante a utilização do sistema. Estas políticas de escalonamento deverão obter um desempenho melhor ou semelhante àquelas em que existe menor informação sobre os possíveis cenários.

As características temporais do sistema são largamente afetadas pelo fenómeno da preempção. Uma preempção define-se pela interrupção de uma dada tarefa, por outra de maior prioridade, durante um intervalo de tempo. A tarefa que interrompe acede a recursos partilhados alterando o seu estado. Quando a tarefa inicial tornar a executar, irá sofrer uma interferência temporal adicional devido à alteração de estado dos recursos partilhados. A natureza deste tipo de interferência é bastante complexa. Inerentemente, a caracterização do pior cenário possível esta dotada de um pessimismo acentuado. Nesta tese o comportamento das preempções e as suas consequências, no pior caso, são estudadas num enquadramento de prioridades limitadas, o que permite a redução considerável do pessimismo das análises temporais em comparação com os métodos existentes na literatura.

Os algoritmos de escalonamento providenciados oferecem melhores garantias de desempenho na fase de conceção, bem como melhor comportamento em média durante a execução do sistema.

Abstract

Physical phenomenons, regardless of its degree of human intervention, from the more pristine to the more vain facets of some human made appliance, require controlling strategies to achieve an intended set of properties or some performance level. The controllers' implementation may range from the simpler mechanical or analog circuitry but more commonly are embodied as a recurrent set of commands on some computational boolean logic device. The dynamic properties of such physical phenomenons, paired with the designer intended behaviour, impose restrictions on the maximum delay between an event and the desired actuation. Real-time systems strive to provide a framework where the considerations about the overall system's temporal feasibility are drawn. In a nutshell, real-time systems enable guarantees on the temporal properties of the computational apparatus which are used in such control loops. Any analysis used in the hard real-time framework should be proven safe. This generally means that the outcome of any analysis states whether all the temporal properties can be guaranteed or that some cannot be trusted upon. The quantity of pessimism involved in the analysis – which leads to an abundance of false negatives or to an over-provisioning of resources – should be reduced as much as possible. Analysis' pessimism can be thought of, in broad terms, as an artefact of the lack of information necessary to accurately characterize the worst-case temporal behaviour of each application. The pessimism can only be mitigated by employing mechanisms where more abundant information about the worst-case run-time behaviour is available. These mechanism should nevertheless have a better or comparable performance to the ones with reduced certainties.

In this thesis both scheduling algorithms and accompanying analysis tools are provided which, by enhancing the available information about what might happen at run-time, allow for a reduction on the level of pessimism associated with the analysis outcomes and bring a better performance in the average case situation. An interesting aspect pertaining to real-time systems is the nature and implications associated with pre-emption. A pre-emption occurs when an application is swapped for another in the execution platform to which it eventually returns. Besides from the time the pre-empting application prevents the pre-empted one from executing, some shared resources are accessed by the former which will potentially interfere with the remaining execution of the latter. The nature of the interference occurring at such resources as the caches or dynamic branch predictors just to name a few is highly complex to analyse and generally a single and oftenly quite isolated worst-case quantity is assumed in the state-of-the-art real-time analysis.

The quantification of the worst-case penalty associated to pre-emptions and the bounding their frequency of occurrence constitutes the bulk of this thesis' contribution. Both scheduling algorithms as well as analysis are provided that both decrease the worst-case number of pre-emptions and also diminish the penalty considered per instance of this event.

Acknowledgements

I feel fortunate for having had the opportunity to conduct research in such a supportive Laboratory where much of the strain of being a PhD student is waned. I am grateful to Eduardo Tovar for having built such a sustainable research environment where any reasonable material request is hardly ever denied. CISTER's team – non-static in time – was always extremely capable. I wish to acknowledge all of the people whose time frame in this team overlaps with mine (even if partially) and to thank all of whom positively contributed to my experience here, whether with fruitful discussions or by easing much of the every-day bureaucracy as Sandra Almeida, Inês Almeida and lately Cristiana have so efficiently done.

A note of appreciation is due to Dr. Vincent Nélis for his contribution to this work and for his habit of finding holes in other people's work, which I find most amusing and ever since tried to mimic.

I am invaluably grateful to Dr. Stefan Petters for enduring me throughout these years. In many accounts he has been a thoughtful and kind supervisor. I wish to thank him especially for his patience with my deambulations, my random neglect for bibtex entries and for always making the effort to motivate me. His expertise on the pre-emption delay, WCET computation, systems architecture (processor and O.S.) and on general real-time scheduling subjects were fundamental for the development of many aspects of my work. More recently, his commitment to reviewing this document was truly remarkable. Research duties aside, I will always remember the time when we (along with Filipe Pacheco and Björn Andersson) did a 2 day road-trip across Europe (from the approximate N.E. extreme to the S.W. corner – all thanks to Eyjafjallajökull, RTAS 2010 and CISTER money), this one is truly memorable for me, despite it being strenuous at the time.

Several contributions in this thesis resulted from collaborations with researchers external to CISTER, namely Dr. Isabelle Puaut, Dr. Robert I. Davis and Dr. Marko Bertogna. I am deeply thankful to all the co-authors for the time they put on the discussions we had and solutions collaboratively derived.

I wish to thank Dr. Michael Paulitsch for having hosted me at Airbus Group Innovations and for giving me a task to perform which was as much to my liking as it was challenging.

Several people exerted a positive influence on me through the years, out of that set I find it only fair to mention Professor Gabriel Falcão in this context. As my MSc. thesis co-advisor he was largely responsible for getting me interested by academic research.

Finally, I would like to thank my Mother for nurturing my curiosity from early age. I am mostly grateful to my family for their encouragement and for accepting the choice of delaying my entry into *adulthood*, and to Sara for having shared all this time with me, keeping me focused whenever I went astray, for being my partner in the full sense of the word.

Contributions

Over the course of this Ph.D. the following works have been accepted for publication in peer reviewed venues:

ETFA 2014 - José Marinho, Vincent Nélis, Stefan M. Petters, "Temporal Isolation with Preemption Delay Accounting" [MNP14]

RTSS 2013 - José Marinho, Vincent Nélis, Stefan M. Petters, Marko Bertogna, Rob Davis, "Limited Pre-emptive Global Fixed Task Priority" [MNP+13]

RTCSA 2013 - Rob Davis, Alan Burns, José Marinho, Vincent Nélis, Stefan M. Petters, Marko Bertogna, "Global Fixed Priority Scheduling with Deferred Pre-emption" [DBM+13], **[Best Paper Award]**

RTNS 2012 - José Marinho, Stefan M. Petters, Marko Bertogna, "Extending Fixed Task-Priority Schedulability by Interference Limitation" [MPB12]

SIES 2012 - José Marinho, Vincent Nélis, Stefan M. Petters, Isabelle Puaut, "An Improved Preemption Delay Upper Bound for Floating Non-preemptive Region" [MNPP12b]

DATE 2012 - José Marinho, Vincent Nélis, Stefan M. Petters, Isabelle Puaut, "Preemption Delay Analysis for Floating Non-Preemptive Region Scheduling" [MNPP12a]

EUC 2011 - José Marinho, Stefan M. Petters, "Job Phasing Aware Preemption Deferral" [MP11]

RTSOPS 2011 - José Marinho, Gurulingesh Raravi, Vincent Nélis, Stefan M. Petters, "Partitioned Scheduling of Multimode Systems on Multiprocessor Platforms: when to do the Mode Transition?" [MRNP11]

ECRTS 2011 - Vincent Nélis , Björn Andersson, José Marinho, Stefan M. Petters, "Global-EDF Scheduling of Multimode Real-Time Systems Considering Mode Independent Tasks" [NAMP11]

WARM/CPSWEEK 2010 - José Marinho, Stefan M. Petters, "Runtime CRPD Management for Rate-Based Scheduling" [MP10]

“Wo gehobelt wird, fallen Späne”

– German Proverb

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

ACS	Abstract Cache States
BB	Basic Block
BRT	Block Reload Time
CFG	Control Flow Graph
CPMD	Cache Related Preemption and Migration Delay
CRPD	Cache Related Preemption Delay
CUV	Cache Utility Vector
DJP	Dynamic Job Priority
ECB	Evicting Cache Blocks
ECS	Evicting Cache Set
EDF	Earliest Deadline First
FJP	Fixed Job Priority
FTP	Fixed Task Priority
GFTP	Global Fixed Task Priority
LCS	Live Cache State
LMB	Leaving Memory Blocks
RCS	Reaching Cache State
RMB	Reaching Memory Blocks
RQL	Ready Queue Locking
RTA	Response Time Analysis
SIL	Safety Integrity Level
UCB	Usefull Cache Blocks
WCET	Worst-case Execution Time
WCRT	Worst-case Response Time

Chapter 1

Introduction

The vast majority of the digital computing devices are embedded into larger systems, where the functionality is greater than the one provided by the computing apparatus. Some of these embedded devices, due to the constraints of the systems in which they are embedded, need to meet temporal requirements.

The term real-time is sometimes miss-understood. Real-time systems as a subject does not necessarily deal with high performance devices where the focus is on the presentation of the results to the user such that the illusion of latency absence would be produced. Straightforwardly, real-time system design focuses on obtaining upper-bounds on the completion time of a given workload on a given hardware platform.

In order to derive such upper-bounds the specifics of the workload and of the execution platform need to be studied. Generally a model of the workload is created and the assessment of the schedulability is done considering its characteristics. Any considered system is assumed to be deterministic. This means that if the initial state is fully characterized and the model is sufficiently detailed then a perfect description of future states is obtained. Both the attainment of the full initial state description and the model might be prohibitively expensive in time and complexity terms. Hence generally over-approximations of both are considered in the analysis. The level of detail contained in the workload and the hardware platform models influences the pessimism involved in the analysis and the time consumed to perform it. It is common that increasing the detail level of the model leads to lengthy and more complex analysis processes.

Rather than dealing with average-case situations, the analysis in real-time systems is concerned with the worst-case scenario. For this scenario, which has to be specified on the framework provided by the concrete model adopted, each task in the system is certified to finish the entire work issued by it before a pre-specified time instant. This time instant is commonly referred to as the deadline. If the worst-case workload completion time is smaller than the workload deadline than the workload is termed schedulable on that platform.

1.1 Real-world Timing Requirements

Any outcome emerging from the processing carried out from a digital computer loses interest as time passes. The sole purpose of computers is to process and transmit signals to a given entity which at times takes decisions and acts based on the provided output. An entity may entrust a computer with the task of answering a given question, say, the "Ultimate Question of Life, the Universe, and Everything" [1]. If the answer is presented 7.5 million years later, so long that the entity forgot what the question was, little usage exists for the glorious outcome of 42. Whether 42 is a correct outcome or not is not of importance in real-time systems research but rather the focus of the areas of dependability and reliability. Real-time systems research focuses only on the timing aspects of the solution generation.

In the most evident scenarios a timing drift of the output generation instant from the actual useful output generation time interval brings no consequence other than lack of system responsiveness and frustration for the eventual operator.

There are cases though, where the failure to generate the computation output on time has substantially more severe consequences. Let us observe the requirements of a given system responsible for adjusting and maintaining a plane cruising altitude. This system, given an altitude set-point, will be responsible for actuating on the airplane control surfaces and engines such that the set-point is reached while meeting certain performance requirements.

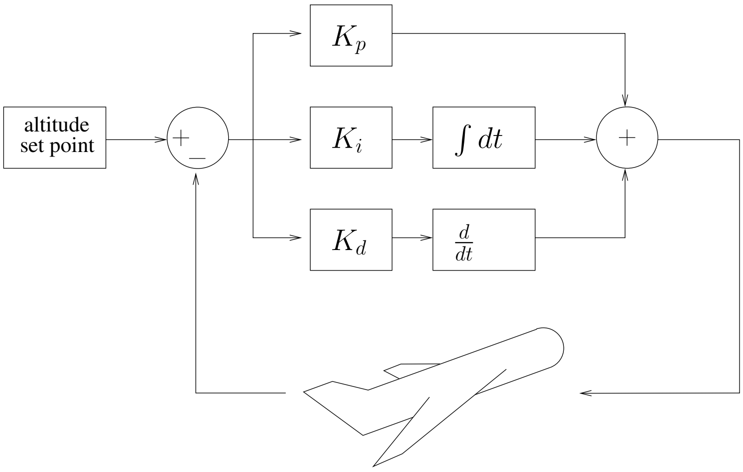


Figure 1.1: Flight Level Closed Loop Controller Diagram

In this example the altitude controller is implemented with a classic Proportional Integral and Derivative (PID) controller (Figure 1.1). After the controller parameters are tuned in continuous time so as to achieve the intended closed loop system behaviour, the controller

[1] illustrative example extracted from the book "The hitchhiker's guide to the galaxy" , ISBN:0517542099 9780517542095

is mapped into an equivalent discrete time controller. The discrete time version of the controller is implemented in software and run in the execution platform. The specificities of the controller parameter tuning and mapping process from continuous time to discrete are not fundamental for the understanding of the issues up for discussion. The single fact that the reader must acknowledge is that during the mapping procedure one sampling period is chosen (T_s). This sampling period is at the heart of the controller description and behaviour. In order for the closed loop transfer function to be in accordance with the one derived, this sampling rate must be stringently met. This means that at every T_s time units the set-point has to be evaluated as well as the current airplane altitude. Having the two input variables the control signal is computed. This computation has to be completed before the next control period. The consequences of not meeting the strict timing requirements of this controller may be mild, in the sense that the system may take longer than intended to match the actual altitude to the set-point. In a worst case the controller may become unstable, leading to violent altitude oscillations from the plane as depicted in Figure 1.2. If the system does not behave according to the specification, given some circumstances, it may lead to life threatening situations or significant financial losses. It is fundamental to ensure that such situation can never happen irrespective of the circumstances [2].

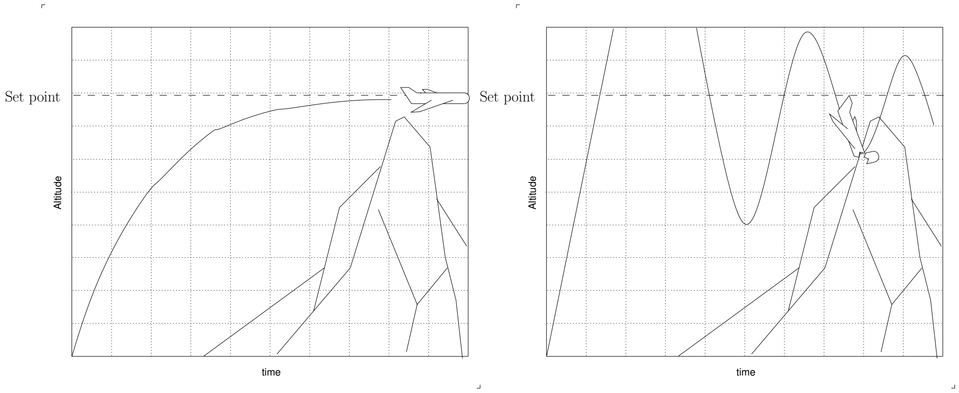


Figure 1.2: Correct controller behaviour versus incorrect behaviour

In order to prove the correct temporal behaviour of the system a framework where such proofs can be derived is required. In this framework the interaction between different softwares and the executing platform is modelled. Let us describe a possible system abstraction where the correct temporal behaviour of the system can be proven.

[2] The presented PID example constitutes an oversimplification of a flight controller implemented in a digital apparatus, its intention is solely to illustrate the origin of the temporal constraints in hard real-time systems

1.2 System Model

The control system is generally devised assuming that all the procedures involved take an infinitesimally small time interval. This is of course not true in a real system. All the procedures when implemented in software will take a non-negligible time interval to complete and in order for the correct system execution to be guaranteed, the intended timing properties of such setup must be guaranteed as well.

The given controller has to execute a set of software routines every T_s time units. These routines are constituted of for instance: sampling the relevant state variables (accelerometers, pressure sensor, etc.), computing the open loop quantities (proportional, integral and derivative part); lastly the control command is issued to the actuators. These computational steps are modelled in our system by a workload quantity.

Definition 1 [Workload]: *metric of computation requirement which executes upon a given platform. One unit of workload takes one units of time to execute in a given platform.*

The workload requirement of the filter may eventually be variable. As variable as it may be, the software running implementing the controller will always be composed of a finite set of instructions (assuming that for every loop in the program the maximum number of iterations is known and is bounded). If each instruction takes a finite amount of time to complete then the software will execute until completion in a finite amount of time as well. An upper-bound on this time quantity may be derived, and is commonly termed Worst-case Execution Time (Chapter 2).

The entity to which the execution requirement is associated to is termed job.

Definition 2 [Job]: *a given quantity of workload released in the system at a specific time instant with absolute deadline and a maximum workload requirement.*

The jobs are nothing more than an abstract representation of software executing upon a given platform. The amount of software instructions is well bounded. This bound on the maximum number of software lines executing within each job is modelled as a workload quantity. In concrete terms, a job models an execution requirement that will take no more than a prespecified number of time units of access to the platform to execute until completion.

With a job we can model the execution of the controller software implementation during *one* sampling period. However, the controller will require its software implementation to execute once every sampling period. In order to model this recurrent execution a concept termed task is employed.

Definition 3 [Task]: *entity responsible for releasing jobs with a minimum separation between each individual release.*

A task is an overall abstract representation of the execution behaviour of a given piece of software on a given platform as well as of its temporal requirements.

To summarize we state that a task is then characterized by the three tuple $\langle C_i, D_i, T_i \rangle$. This document focuses purely on tasks where $D_i \leqslant T_i$ which is commonly referred to as constrained deadline model. The parameter C_i represents the worst-case execution time of each job from a task τ_i, D_i is the deadline relative to the job release and T_i the (minimum) distance between consecutive job releases. Each task τ_i may release a potentially infinite sequence of jobs with releases separated by at least T_i time units. Every job k from task τ_i has an absolute deadline defined at time $d_{i,k} = r_{i,k} + D_i$, where $r_{i,k}$ is the absolute release time instant of job k from task τ_i.

Returning to the controller example context: in order to ensure that all the workload of the control loop is completed before the next period, one can arbitrate the relative deadline of the control task to be equal to T_s, since only after T_s time units will the controller have to provide the actuator commands. If only the mentioned controller was executing on the platform it would suffice to check whether $C_i \leqslant D_i$. If that condition is met the controller is guaranteed to execute all software instructions before the deadline, hence the timing properties would be met, leading to a controller which exhibits the behaviour intended at the design stage.

As we have seen that a real-time system is composed both of software and corresponding execution platform. The software part of the real-time system constitutes a real-time application (Figure 1.3). Several distinct applications might eventually coexist in a single platform.

A real-time application may in fact incorporate more than a single software component.

Definition 4 [Real-time application]: *Collection of software constructs which by executing upon the hardware platform provide the functionality intended by the system designer respecting the pre-specified timing constraints.*

A real-time application may be seen as a collection of, for instance, several digital controllers and filters, some interface applications and avionics display generation. Where all of these subconstituents share a common execution platform. Each of those software pieces can be modelled as a task. Each of which with their specific timing requirements.

The collection of software constructs is then commonly modelled as a set of real-time tasks termed task-set. The task-sets considered in this document are formally defined as $\tau = \{\tau_1, \ldots, \tau_n\}$, meaning that τ is composed of n tasks elements.

When checking whether the workload of the controller is able to complete before their deadlines (i.e. ensuring the correct system temporal behaviour) one has to take into consideration the remaining constituents of the task-set and the way they interfere with each other when contending for execution on the platform. The entity responsible for mediating the contention for processor usage is denominated by “scheduler”. Considering the task-set

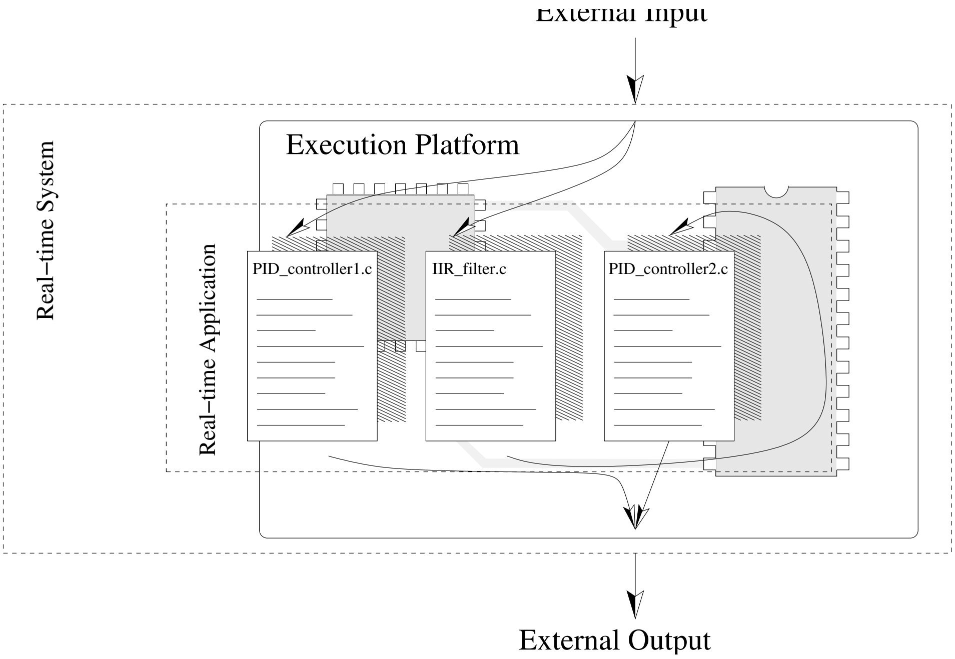


Figure 1.3: Conceptual Real-time System

characteristics as a whole along with the scheduler properties enables the system designer to prove whether all software constructs will manage to execute on the platform while meeting their timing requirements.

1.3 Schedulers

The scheduler takes the decision of which task to execute at any time instant following a specific scheduling policy. A scheduling policy is a well-defined set of rules which dictate which software executes when and on which processor. The scheduling policy used to carry out a given workload will greatly influence the temporal behaviour of the tasks in the system. There are several distinct scheduler classes:

1. Fixed task priority (FTP): each task in the system has a given predefined priority, every job released by that task executes at its priority level;
2. Fixed job priority (FJP): at the time of release of a job from the given task a job priority is decided upon. From the job release until its workload completion the job will hold the same priority.;
3. Dynamic job priority (DJP): The priority of a given job may vary over time.

These three categories tend to be employed when describing on-line schedulers, nevertheless static schedulers such as cyclic executives are also subject to this grouping since the workload is still prioritized according to a set of rules even though this process occurs at design time and not during system execution.

The choice of which scheduling policy to use is an exercise where the up and down sides of the alternatives have to be carefully considered by the system designer. Scheduling disciplines may be weighted according to several metrics. An important one is schedulability (i.e. its ability to schedule task-sets such that all the tasks meet their timing requirements).

Definition 5 [Schedulable task-set]: *A task-set is said to be schedulable on a given platform by a scheduling algorithm $\mathscr{A}$ if, for all $\tau_i \in \tau$, any job released by τ_i is guaranteed to complete at least C_i units of workload in a time interval from the release of the job ($r_{i,k}$) and the absolute deadline of the job ($d_{i,k} = r_{i,k} + D_i$) in any valid schedule generated by $\mathscr{A}$.*

Increased schedulability guarantees (i.e. the ability to schedule more task-sets) generally come at the cost of increased complexity in the scheduling decisions. This may lead to unnecessary overheads due to scheduler operation, which lead to computational resources being wasted and added complexity in the scheduler implementation.

In single-core there exist FJP (e.g. EDF [LL73]) DJP (e.g. LLF [OY98]) scheduling policies which are optimal when considering that interference between task occurs solely on the processor. The scheduler implementation of these policies will generally take a greater quantity of information as an input for their on-line decision making, thus being prone to also greater run-time overheads. In that respect the fixed task priority scheduling algorithm presents a relatively small complexity.

If a given task-set is schedulable with a lower complexity scheduling mechanism, then there might be little motivation for deciding to use a higher complexity one. In this work the scheduling policy put to use is generally FTP.

The most common scheduling policies considered in the literature are fully pre-emptive. In this type of scheduling mechanism, at any time instant t, the job executing on the processor is the highest priority job in the system. Whenever a task gets assigned to the processor it is either the case that the previously executing task has terminated or that it got pre-empted, which implies that it still had remaining work to complete. When using fully pre-emptive schedulers little information is available on where these pre-emptions may occur, which can be problematic if the pre-emption impact needs to be quantified.

As it turns out, an important aspect pertaining to the operation of the system is the number of pre-emptions the tasks are subject to. The pre-emptive behaviour will give rise to overheads not present when a given workload is integrally executed without interruption of any sort. These overheads are generally taken as null or negligible in scheduling theory, but are in fact substantial.

Non-pre-emptive scheduling policies are safe from these overheads. In this scenario, when a job from a task is dispatched onto the processor it will not suffer any pre-emptions by any workload irrespective of its priority. This generally comes at the cost of decreased ability to schedule workload (i.e. to meet the task-set deadline guarantees). A hybrid method between fully pre-emptive and non-pre-emptive scheduling policies may be considered in order to have better scheduling ability and more information on the pre-emptive behaviour of the system. In these scheduling policies pre-emptions are allowed under certain restrictions.

This might be in the form of setting fixed pre-emption points to enable a tighter bound on the number of pre-emptions. Setting pre-emption points is an effective method to increase the schedulability of fixed task-priority systems [BBM+10]. The mechanism denominated *fixed non-pre-emptive regions*, relies on specific pre-emption points inserted into the task's code. This has to be done at design time, relying on WCET estimation tools that can partition a task into non-pre-emptible sub-jobs [BBM+10]. This is highly restrictive since these points can not be replaced at run-time making changes to the task-set difficult. Some systems require on-line changes whether on the workload itself or just on their rate of execution and temporal deadlines, hence would require the pre-emption points to be replaced. Furthermore, the choice of pre-emption points placement proves to be a non-trivial task for complex control-flow graphs. Which is an additional concern for the application developer and a potential source of errors. Since the incorrect placement of the pre-emption points may lead higher priority workload to violate its temporal constraints. Having a more flexible mechanism helps to reduce development, software maintenance, and update costs. It also facilitates the operation of systems which require run-time workload changes. An example of a considerably more flexible limited pre-emptive mechanism is the *floating non-pre-emptive regions* model. In this model a non-pre-emptive region of execution is started at the time instant t when a job of higher priority than the currently executing job from task τ_i, is released where at time $t-\varepsilon$ the job from task τ_i had the highest priority from all the active jobs at that time instant. This non-pre-emptive region has a limited duration which is a function of the higher priority workload.

1.4 Background on Caches

It is common for current high-performance processors to sport several architectural acceleration units. This is the case of caches due to the big discrepancy between the execution core and memory throughput [EEKS06, RGBW07]. In the embedded world the range of cache levels present in each processor vary from zero up to three. Generally the processors which lack caches present a relatively low clock frequency so the divergence between processor and memory throughput is not relevant. For the high end performance spectrum caches are widely employed and the importance of their analysis grows.

Caches are subsystems which display, at any time-instant, an associated "*state*". These units quasi-continuously face state changes at run-time. In particular, it is the case for task pre-emptions: when a task resumes its execution (after being pre-empted), the cache(s) will display a state which is different from its state at the time the task got interrupted.

Let us briefly describe the workings of caches. A cache is a memory region which takes considerably less time to be accessed in comparison to the main system memory.

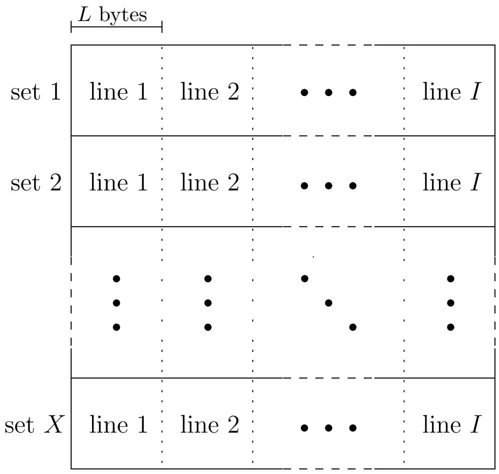


Figure 1.4: Cache Representation

A given cache architecture geometry is completely defined by four parameters (Figure 1.4), namely cache size (cache-size), number of cache sets in the cache (#cache-sets) , number of cache lines per set (#associativity) and line size (line-size). The number of cache lines per set is also termed associativity. In is said that if a cache has #associativity lines per set that it is a #associativity $-$set associative cache. The associativity may vary between two extremes, #associativity $= 1$ which is termed direct mapped and #associativity $= \frac{\text{cache-size}}{\text{line-size}}$ which is termed fully associative. In the direct mapped case each memory block can reside only in a single cache line. In the fully associative scenario each memory block can be loaded into any cache line. Cache lines have a given length (line-size) of several bytes. In regular processor architectures a cache line will hold either line-size $= 64$ or line-size $= 128$ bytes. If a cache is said to have a size of cache-size bytes then it is composed of

$$\text{\#cache-sets} \overset{\text{def}}{=} \frac{\text{cache-size}}{\text{line-size} \times \text{\#associativity}} \tag{1.1}$$

cache sets.

All memory locations in the system map to a specific cache set. The mapping function cache-set-map is defined as follows:

$$\text{cache-set-map}(\text{address}, \text{line-size}, \#\text{cache-sets}) \overset{\text{def}}{=} \left\lfloor \frac{\text{address}}{\text{line-size}} \right\rfloor \mod \#\text{cache-sets}\,. \quad (1.2)$$

This means that whenever a memory location is referenced, when the corresponding memory line is fetched it will be mapped to the $\text{cache-set-map}(\text{address}, \text{line-size}, \#\text{cache-sets})$ cache set and stored in some cache lines belonging to that set.

When a memory *address* is referenced two situations may arise:

- address is in cache in which case a cache hit is said to occur
- address is not in cache at this time a cache miss occurs, and the line where the referenced memory address sits is loaded into the cache.

A memory reference is generally between one to 64 bits in length and the address points to the first byte. When a memory is referenced the entire line where the memory is resides is brought to the cache. This means that when a given address is referenced then the line-size bytes in the set $\{\text{address} \mod \#\text{associativity}, \cdots, (\text{address} \mod \#\text{associativity}) + \text{line-size}\}$ are loaded into the cache.

If the destination set is filled with valid memory lines but does not contain the referenced memory block, one of the lines present in the set has to be evicted in order to make space for the current line reference. There exist several cache replacement policies that mediate the line eviction on the cache sets. The most commonly considered in worst-case execution time assessment are Least Recently Used (LRU), First In First Out (FIFO), Pseudo Least Recently Used (PLRU).

- **LRU:** From all the memory blocks in a given set, the one which was referenced the furthest in the past is the one evicted at the time of a new memory block insertion into the cache set. Since this replacement policy relies on $\lceil \log_2(\#\text{associativity}!) \rceil$ additional bits per cache-set so as to preserve a full track of the entries age it is uncommon for LRU to be employed for $\#\text{associativity} > 4$;
- **FIFO:** if a given memory block is not in the cache set then it is inserted into the cache head and evicts the last element in the queue. A memory reference to a given line in the set does not alter the set ordering. Hence, a cache line can only survive in a set for $I - 1$ new element insertions
- **PLRU:** There exist several possibilities of implementing this type of cache replacement protocol. Overall, it tries to approximate the behaviour of LRU without incurring on the resource usage and greater latencies involved with actual LRU implementation. In the tree based implementation, the $\log_2(\#\text{associativity}) + 1$ most recently

used cache lines are safe from eviction [RGBW07], whereas an eviction victim is chosen dependent on the state of the tree from the $\#\text{associativity} - \log_2(\#\text{associativity}) - 1$ remaining cache lines. This implies that PLRU caches may be approximated to an equivalent LRU cache with $\#\text{associativity}' = \log_2(\#\text{associativity}) + 1$ and size $\text{cache-size}' = \#\text{associativity}' \times \text{line-size} \times \#\text{cache-sets}$.

By far the most employed in modern processors, when $\#\text{associativity} > 2$ is PLRU, for $\#\text{associativity} = 2$ then LRU is commonly used.

Once the cache set and the cache line where the referenced memory line is going to be placed, the evicted cache line may or may not be written to the lower memory level. This will largely depend on the write policy of the cache. The two most common are:

- **write-through:** all memory block modifications are added to the cache and in parallel written to lower memory levels;
- **write-back:** changed memory blocks in the cache are only written to lower memory-levels once an eviction occurs.

The write-through will have as a consequence that every memory store induces a change on main memory following said reference. This may greatly increase the traffic observed in the communication bus. On the other hand the write-back policy has the unfortunate consequence that when a write occurs which yields a cache miss then it can only be completed once the evicted cache is written onto lower levels. This situation only occurs in data caches, since generally instructions change (self-modifying code) is not perceived as an architectural use-case.

In the architectural model employed in this work we use two values to model the behaviour of cache hits and misses. A cache hit has a latency of T_{HIT} time units. In regular processors T_{HIT} is a constant value of between one or few processor clock cycles. The systems considered are composed of a single cache level, hence a cache miss occurrence triggers a memory fetch from main memory to the cache. A cache miss is assumed to have a latency T_{MISS}. Even though a cache miss will have a variable latency, T_{MISS} is conservatively assumed to be a constant, upper-bounding the maximum amount of time that a memory block takes to be transfered from main memory to the cache, which is one memory operation for write-through and two memory accesses for the write-back policy (in case the evicted line had been modified – dirty bit set). This previously holds true for data caches, whereas the T_{MISS} for instruction caches is set to one memory access time irrespective of the write policy. Furthermore we will restrict our model to systems for which $\#\text{associativity} = 1$ since it allows for easier explanation of the concepts and extensions to $\#\text{associativity} > 1$ for well behaved replacement policies as LRU are trivial to obtain.

If a task has several memory blocks in cache which may reuse in the future and its execution is interrupted by another task, some of these blocks may get evicted. When the task resumes execution the state of the cache will be distinct in relation to the point when it was pre-empted. Eventually the task as it executes requires the state of the cache to be re-established. The reconstruction procedure is subject to time penalties. In real-time systems, where timeliness is an essential property of the system, these penalties need to be carefully evaluated to ensure that all deadlines are met.

Generally, since the WCET is computed for each task in isolation, all the overheads attached to the scheduler pre-emptive behaviour are not present in the WCET value. The consequences of pre-emptive behaviour then has to be accounted for at later stages of the schedulability analysis. In order to avoid double accounting of overheads, it is important to have in mind the assumptions and procedures used in the WCET computation. For instance the cache misses considered in WCET should not be again considered as an effect of the pre-emptive behaviour.

Once an upper-bound of the overheads has been computed an integration into the schedulability tests is performed for the current workload such that the schedulability is guaranteed in any possible scenario. More on the subject of WCET computation, pre-emption delay assessment and schedulability conditions taking both parameters into account are described in Chapter 2.

1.5 Thesis Organization

The thesis put forward on this work grounded both on several state-of-the-art works and novel contributions presented in this document is clearly posed in the following wording:

Thesis– **Limited pre-emptive schedulers allow for significantly reducing pessimism in the pre-emption delay accounting in single-core and for enhanced schedulability when employing global fixed task priority schedulers.**

In this thesis the theory of limited pre-emptive scheduling is addressed and extended. Following this preliminary chapter(1), where the work is put into context and a broad depiction of the real-time area is provided, the related work is described (Chapter 2). The related work chapter covers the aspects of the worst-case execution time computation strategies. The assumptions taken during these steps greatly influence the considerations taken during the pre-emption delay assessment. An overview of the literature pre-emption delay assessment strategies for regular fully-pre-emptive scheduling is provided. Finally some further information is provided on the matters of limited pre-emptive theory in single-core and related to the temporal-isolation aspects.

The bulk content of the thesis is provided in the chapters succeeding the state-of-the-art. The thesis can be broadly organized in in four main areas content-wise:

- **Single Processor Limited Pre-emptive Theory Extension (Chapter 3)**:

The single-core limited pre-emptive theory for fixed task-priority scheduler is revisited. An series of methodologies are provided which, by resorting to information available at run-time, allow for extensions to the non-pre-emptive region length to be obtained. The properties of the algorithms with respect to pre-emptions is studied and upper-bounds on the number of pre-emptions are derived. Experimental results are presented showing that the proposed solution reduces the observed number of pre-emptions when compared to the state-of-the-art limited pre-emptive method.

A method to increase the schedulability in a similar way to the one achieved with fixed non-pre-emptive regions is also presented in this chapter. Unlike the more static limited pre-emptive schedulers (fixed non-pre-emptive) the more dynamic (floating non-pre-emptive) does not easily allow for schedulability increases. The presented ready queue locking mechanism prevents higher priority releases that occur after a prespecified time instant to interfere with lower priority pending work. An integration of this methodology with the well known pre-emption threshold is described as well. The derivation of the schedulability conditions is presented.

- **Pre-emption Delay estimation for the Floating Non Pre-emptive Region Model (Chapter 4)**: The usage of the floating non-pre-emptive region model in the state of the art works so far only allowed for a reduction on the number of pre-emptions quantified at design time. In this section the scheduling information is exploited in the analysis along with a further description of the workload to be executed and the evolution of the pre-emption delay cost as the execution progresses. A method which considers a function of the pre-emption delay per progress unit is described which yields an upper-bound on the pre-emption delay a task is subject to in the floating non-pre-emptive region model. Further extensions to this model are provided that take into account an additional function and reduce the pessimism taking into account assumptions from the WCET analysis phase.

- **Ensuring Temporal-isolation in Platforms with Non-negligible Pre-emption Overheads (Chapter 5)**: More often than not, temporal-isolation is overlooked in the real-time theory. The same cannot be said in industry where strong independence must be proven between separate workload modules in order for systems to be certified. This Chapter proves that the state-of-the-art temporal-isolation solutions are wasteful of resources and that in more complex models fail to provide the temporal-isolation property at all. An alternative method to encompass the temporal-isolation problematic in systems with non-negligible pre-emption delays is described. This method is presented for single-core scheduling but can be adapted to multi-core scheduling as well.

- **Derivation of the Limited Pre-emptive Theory for Global Multiprocessor Scheduling (Chapter 6)**:

  This Chapter delves on the multi-processor limited pre-emptive theory. The derivation of some models for limited pre-emptive scheduling (fixed non-pre-emptive) is described for global fixed task priority. The floating non-pre-emptive region model theory is further addressed as well. Even though most of the work was carried out assuming fixed global task priority schedulers it is shown that the limited pre-emptive scheduling mechanism allows for a schedulability increase in the GEDF case. Experimental results were obtained in order to assess whether the same patterns observed for single core would translate into global scheduling with respect to schedulability and overall estimation of pre-emptions.

The thesis is finalized by Chapter 7 where a summary of the thesis is provided and the a list of possible future directions is debated.

Chapter 2

Related Work

2.1 WCET Upper-bound Computation

Execution time bounds for task operation are an ubiquitous input in real-time systems theory. The value is computed in isolation or with the inherent assumption of non-pre-emptive execution. The computation of these upper-bounds may be divided into two distinct categories.

1. Measurement Based
2. Static Analysis
3. Hybrid

2.1.1 Measurement Based

Measurement based WCET estimation procedures rely on the end-to-end or partial execution of the task in order to extract an estimate of the upper-bound. A set of relevant input vectors is decided *a-priori* by the system designer. The main drawback of this approach is that it is often impossible to know *a-priori* which input vector will lead to the worst-case execution time. As a result the largest execution time observed can not be trusted as a safe upper-bound, and thus can not really be employed as an input for schedulability assessment in hard real-time systems. An additional concern is that measurement based WCET estimation does not easily allow for the quantification of the pre-emption delay associated to each task.

2.1.2 Static Analysis

A different approach for the execution time estimation is the static analysis of the task. This methodology relies on four specific steps:

- Control Flow Graph (CFG) extraction;
- Micro-architectural modeling;
- Value analysis;
  - Path analysis;
  - Constraint analysis;
- Value computation

In the first step the program is parsed at source or machine language level in order to extract valuable information for subsequent analysis steps. Using abstract interpretation a structural representation of the program is achieved [CC77]. This abstract representation is a directed graph denominated control flow graph (CFG) [CP01].

The set of nodes composing a CFG are of different types:

- Entry: has no predecessors and a single successor;
- Assignment: has exactly one predecessor and one successor. The functional parts of the code (i.e. the actual instructions) are associated to this node;
- Test: has a predecessor and two successors. The jump instructions are associated with such nodes;
- Junction: has more than one predecessor and a single successor. Every execution path joining will occur in this node type;
- Exit: has one predecessor and no successors.

A CFG is then a directed graph where nodes are termed basic blocks. An example is provided in Figure 2.1.

At this point the CFG is again parsed, at the value analysis step, in order to remove infeasible paths, and extraction of loop bounds. This is done either by knowing the set of possible inputs or by analyzing the semantics of the program.

The information present in the CFG may be exploited by several techniques to compute the WCET. As next step, the Micro-architectural modeling procedure is performed. This is constituted by analyzing each instruction in the assignment Basic Block (BB) of the CFG with respect to its latency. Another important aspect of this stage of WCET computation lies on the classification of the memory references with respect to both instruction and data caches.

In order to classify each memory instruction the CFG is parsed by two fixed point algorithms, the *must* and *may* analysis, with the knowledge of the cache, line size and associativity level. The *must* analysis computes, for each program point the set of memory lines which must be in the cache at each program point. The may analysis, in contrary, computes the set of memory lines which may be in the cache for every program point. Both must and may analysis sets are Abstract Cache Set (ACS) since they are the product of an analysis which does not consider a specific program path.

This in turn yields the following memory reference classification:

- always-hit (AH): at the given program point the memory address referenced by the instruction is present in the ACS_{must};
- always-miss (AM): at the given program point the memory address referenced by the instruction is not present in the ACS_{must} nor in the ACS_{may};
- not-classified (NC): the memory line referenced in the current instruction is not in the ACS_{must} but is part of the ACS_{may}.

For a system with a single cache level the memory references classified as AH will be treated as a cache hit, whereas the ones classified as AM and NC are treated as cache misses.

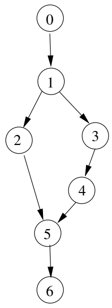


Figure 2.1: Example CFG

With the memory classifications available, the cost of executing each basic block is extracted.

As a last step the value computation is performed. Having the cost of each basic block execution and the set of feasible paths in the CFG, a technique called Implicit Path Enumeration is used. In this framework all possible paths from the CFG are encoded in a linear equation which is then maximized. The outcome of the maximization is the WCET for the given task.

A simpler mechanism may be used in the value computation stage. This method termed tree based analysis relies on the information provided by a syntax tree rather than a CFG. Though the procedure may be explained from a CFG perspective. Basic blocks belonging to the same path segment have their execution cost summed in order to create a fictitious node with the computed execution cost. In the case of a conditional path in the CFG the maximum between the two possible fictitious nodes is taken and another fictitious node is created to represent this conditional part of the program.

Overall the main drawback of static WCET computation is that it relies on an accurate model of the target platform. This, in order not to be optimistic, will have to rely on pessimistic values for the parameters used to model the behaviour of the execution platform. On the other hand measurement based mechanisms can never be guaranteed to be safe and since no cache analysis is performed will always lack information to compute an upper-bound on the CRPD.

2.1.2.1 Hybrid Approach

It is possible to create tools which consist not only of static analysis but also perform measurements on the program or segments of it on the target platform. These methods embody a category denominated as hybrid approach [Bet10]. The CFG information is extracted from the source or object code. Having the program logical structure available one can run the code segments in order to obtain execution upper bounds for each segment in the real platform. Before each segment is run a worst-case scenario can be constructed depending on architectural considerations. This can be for instance the flushing of all caches or running a specific piece of code which polutes the cache forcing subsequent writes to memory for every cache miss observed during the segment run. The segment execution times are then merged in order to obtain a safe upper-bound on the worst case execution time of the task. This tries to overcome the limitations from the previous approaches by joining the best of both worlds. Since static analysis is employed upper-bound on the pre-emption delay penalties can be extracted from the formal analysis of the object code.

2.2 Pre-emption Delay Estimation

As mentioned before, the WCET analysis inherently considers each task in isolation similar to what occurs in non-pre-emptive scheduling. As it turns out generally the system has a pre-emptive scheduler which in turn requires the effect of pre-emptions to be incorporated into the execution time of the tasks.

The pre-emption delay is present whenever some subsystem used in the task execution exhibits some inherent state. A set of states are assumed in the WCET computation. Whenever a pre-emption occurs, the pre-empting task may alter the state of several of the subsystems present in the execution environment. Whenever the pre-empted task resumes execution, if the subsystems it is using are in a different state in comparison to the time instant at which it was pre-empted, it will eventually suffer from pre-emption delay in future execution points. It is inherently assumed that the state changes for these subsystems have a non-negligible time penalty.

The subsystem considered most often in literature is the cache. This is due to the large size of these apparatus and the time penalty for a cache miss which is around two orders of magnitude greater than the latency of a cache hit in modern architectures [McK04, EEKS06, BDM09, RGBW07]. Cache Related Pre-emption Delay (CRPD) estimation has been a subject of wide study. Several methods have been proposed that provide an off-line estimation based on static analysis, for this inter-task interference value.

Lee et al. presented one of the earliest works on CRPD estimation for the instruction caches [LHS+98] where the notion of the set of memory blocks that might have been used in the past and may be used in the future, termed Useful Cache Blocks (UCB), was first introduced.

A UCB, for a given program point, is one which might have been used in any program path leading to this program point, which was not evicted, and that might be used in any future program point accessible from the current location.

The computation of UCB sets is carried out in an abstract interpretation framework. Lee et al. defines gen_b to be the set of memory accesses operated at basic block b mapped into their corresponding cache line [LHS+98]. For example consider a cache with $X = 4$ cache sets, if in BB_b memory addresses m_1, m_2 and m_4, which map to cache set $1,2$ and 4, are referenced then $gen_b = \{m_1, m_2, \perp, m_4\}$. Where $\perp$ signifies that there are no memory references in BB_b mapping to cache block 3. The following operator will aid us on the algorithm explanation task:

$$x \odot y \overset{\text{def}}{=} \begin{cases} y[i] & , \text{if } y[i] \neq \perp \\ x[i] & , \text{if } y[i] = \perp \end{cases} \tag{2.1}$$

Both x and y are an ACS. An ACS is a representation of all the memory blocks which may be in each cache set at a given time. In definition of the operator 2.1, the values $x[i]$ and $y[i]$ represents the contents of the i^{th} cache set in ACS x and y respectively.

There are two ACS computed per basic block, both obtained by a fixed point algorithm. The first, Reaching Memory Blocks (RMB), is computed by an algorithm starting from the first node of the CFG [LHS+98]. The RMB set computed in Algorithm 1, for each BB_b presents the information of all the memory blocks which might have been loaded into the cache on any CFG path leading to BB_b which have not been evicted since.

Algorithm 1: *RMB* set computation

```
foreach $b \in CFG$ do
    $RMB_b^{IN} = \emptyset$
    $RMB_b^{OUT} = gen_b$
$change = true$
while $change$ do
    $change = false$
    foreach $b \in CFG$ do
        $RMB_b^{IN} = \bigcup_{p \in pred(BB_b)} RMB_p^{OUT}$
        $prev = RMB_b^{OUT}$
        $RMB_b^{OUT} = RMB_b^{OUT} \odot gen_b$
        if $RMB_b^{OUT} \neq prev$ then
            $change = true$
```

The LMB [LHS⁺98] set computed in Algorithm 2, for a given BB_b contains the information of the first memory reference which maps to each cache block from any BB_j in all the paths of the CFG accessible from BB_b. The Algorithm for its computation is as forth:

Algorithm 2: *LMB* set computation

```
foreach $b \in CFG$ do
    $LMB_b^{IN} = gen_b$
    $LMB_b^{OUT} = \emptyset$
$change = true$
while $change$ do
    $change = false$
    foreach $b \in CFG$ do
        $LMB_b^{IN} = \bigcap_{s \in successor(BB_b)} LMB_s^{OUT}$
        $prev = LMB_b^{OUT}$
        $LMB_b^{OUT} = LMB_b^{IN} \odot gen_b$
        if $LMB_b^{OUT} \neq prev$ then
            $change = true$
```

Both sets RMB and LMB represent a possible set of memory blocks present in each cache line for every program point. Lee et al. considers the RMB and LMB to be a set of elements uniquivocally associated to each cache set. These represent all the possible memory blocks which may be present in each cache set [LHS+98]. When a union is performed between LMB and RMB sets this implies a union over all the elements which may reside in a given cache set.

For example, consider the abstract cache states $RMB_x^{OUT} = \{a,b,c,\perp\}$ and $RMB_y^{OUT} = \{a,f,\perp,h\}$. Further consider that BB_x and BB_y to be the predecessors of BB_b. The computation $RMB_b^{IN} = RMB_x^{OUT} \cup RMB_y^{OUT}$ has the output $\{a,b,c,\perp\} \cup \{a,f,\perp,h\} = \{a,b,f,c,h\}$. This example holds true for the LMB computation as the union operation over the LMB sets is defined in the same manner. The union of the sets provides an over-approximation on the number of memory blocks, which may be required in the future, that may be evicted if a pre-emption occurs in a given BB_b. The result of the union is the UCB set, which is formally defined in the following manner for each BB_b:

$$UCB_b \overset{\text{def}}{=} ones(RMB_b^{OUT} \bigcap LMB_b^{IN}) \tag{2.2}$$

The function $ones()$ returns a set of binary values, where for each cache line, if the set is empty then 0 is returned for the corresponding element, otherwise 1 is returned.

A slightly altenate manner to compute the UCB set is presented by Mitra et.al. [NMR03]. In this work, the computed RMB and LMB are multi-sets. On the basic blocks where CFG paths merged, the join function has as output a set of abstract cache states. These multi-set are termed leaving and reaching cache states respectively (LCS and RCS). This means that the join operation on the LMBs and RMBs is done in the following manner:

$$LMB_b^{OUT} = \uplus_{s \in successor(BB_b)} LMB_s^{IN} \tag{2.3}$$
$$RMB_b^{IN} = \uplus_{p \in pred(BB_b)} RMB_p^{OUT} \tag{2.4}$$

This leads to less pessimism in the CRPD analysis but at the cost of considerable complexity. While intersecting the LCS and RCS multi-sets at each basic block a multi-set of UCBs is obtained. The fundamental concept of UCB is still the same.

Another alternative for UCB computation is presented by Altmeyer [AB11]. The particularity of this work is the focus on the join operation defined over the LMB and RMB sets. Altmeyer observed that, since memory accesses which are not considered as cache hits are already accounted as cache misses in the WCET analysis, these should not be double accounted formally in the CRPD estimation. Hence only cache lines which must definitely be in the cache are to be considered in the computation of the UCB sets. In order to enforce this observation the join operation defined over the LMB and RMB sets is changed. This operation is then the intersection of the cache line sets between RMBs. Consider the previously referred example where $RMB_x^{OUT} = \{a, b, c, \perp\}$ and $RMB_y^{OUT} = \{a, f, \perp, h\}$. Assuming that BB_c and BB_y to be the predecessors of BB_b, the following computation $RMB_b^{IN} = RMB_x^{OUT} \cap RMB_y^{OUT}$ has the output $\{a, b, c, \perp\} \cap \{a, f, \perp, h\} = \{a, \perp, \perp, \perp\}$. The computation of LMBs is the same as the one defined in the work of Lee et al. [LHS+98]. The remainder of the algorithm for RMB computation is the same as Algorithm 1.

A fundamental assumption taken in the work of Lee et al. [LHS+98] is that the set of UCB is constant throughout a basic block. This assumption is only correct for instruction cache analysis, since by definition a basic block is a set of sequential instructions. Hence there can not be an instruction used in the beginning of a basic block which is later reused. For data caches this assumption no longer holds. This implies that the UCB set for data caches is not constant per basic block and hence has to be computed at each program point, or at least at every program point where a data reference occurs.

An alternative, present in the literature, regarding data cache pre-emption delay analysis was devised by Ramaprasad [RM06a]. In this work, rather than relying on UCB information at each program point, threads connecting memory references to the same address are created. At any program place the number of threads connecting memory references prior and later to the given point will constitute the pre-emption delay associated to an interruption of the program at the current execution point.

2.3 Pre-emption Delay Integration with Schedulability Assessment

Pre-emption delay estimation is of little value without its integration into the schedulability analysis of the task-set. Several approaches have been proposed in the literature to perform this analysis. Scheduling analysis by Lee et al. [LHS+98] is based on response-time analysis (RTA) by using the $PC_i(t)$ to denote the maximum pre-emption delay observed in the schedule until time t and incorporating that quantity into the response time of the task.

Let us define the following concept which allows for a simpler exposure of the theory at hand:

Definition 1 [level-i schedule]: *The schedule composed solely by the sub-set of $\mathscr{T}$ containing only task of priority equal or higher than τ_i.*

The concept of level-i schedule is henceforth defined for fixed task priority only, and is only employed in that context.

The RTA is defined as [LHS+98]:

$$R_i^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \times C_j + PC_i(R_i^{k-1}) \tag{2.5}$$

This is solved by aid of a fixed point algorithm, i.e. the algorithm terminated in the k^{th} iteration such that $R_i^k = R_i^{k-1}$. The function $PC_i(t)$ denotes the worst-case CRPD generated in the level-i schedule in a time interval of length t and is defined as:

$$\text{maximize } PC_i(t) \overset{\text{def}}{=} \sum_{j=2}^{i} g_j(t) \times f_j \tag{2.6}$$

The variable f_j encodes the maximum pre-emption delay which a job from τ_j can suffer as a consequence of a single pre-emption.

The function $g_j(t)$ represents the upper-bound on the cumulative number of pre-emptions which jobs from task τ_j may suffer in a time interval of length t. This variable is restricted by the two following inequalities.

1. $\sum_{k=2}^{j} g_k(t) \leqslant \sum_{b=1}^{j-1} \left\lceil \frac{R_i}{T_b} \right\rceil$
2. $g_j(t) \leqslant \left\lceil \frac{R_i}{T_j} \right\rceil \times \sum_{k=1}^{j-1} \left\lceil \frac{R_j}{T_k} \right\rceil$

Constraint number one dictates that the number of pre-emptions which jobs from task of the set $\{\tau_2, \cdots, \tau_j\}$ suffer in a time interval of length t is limited by the number of releases of jobs of tasks in the set $\{\tau_1, \cdots, \tau_{j-1}\}$. The second constraint limits the number of pre-emptions that jobs from task τ_j may suffer in a time interval of length t to the number of releases of jobs from task τ_j in the said interval times the maximum number of pre-emptions each single job from τ_j may suffer in the worst-case scenario.

Lee et al. [LHS+98] uses integer linear programming (ILP) to maximize the function $PC_i(t)$ (2.6). This yields an upper-bound on the pre-emption delay suffered by each task. The ILP problem is solved for every iteration of the response time fixed point algorithm (Equation (2.5)) where the objective function maximized is $PC_i(t)$. This approach is very complex due to the need for linear programming solving at every iteration.

An extension of this work by the same authors [LLH+01] exists. In this, the variable f_j ceases to be treated as constant value for each τ_j but rather a table of values which considers all possible combinations of higher priority tasks pre-empting task τ_j. A constraint on f_j is added to the previously described optimization problem formulation. The complexity of the approach is exacerbated since the ILP needs to be solved at every iteration of the RTA computation for each task with the added constraint complexity. Since then, the focus of the CRPD related literature has shifted from the ILP formulation to a rather less complex framework which explores particularities of the scheduler and the patterns of cache usages in order to compute CRPD upper-bounds.

Busquets et al. also used RTA [BMSO+96]. In this work the pre-emption delay is considered from the point of view of the pre-empting task. The concept of Evicting Cache Blocks (ECB) is used to model the maximum amount of information a pre-empting task may evict from the cache. The ECB is the $ones(RMB_{exit}^{OUT})$ set where the BB_{exit} is the terminating basic block of the task's CFG. The computation of ECB may be also written as:

$$ECB_j \stackrel{\text{def}}{=} \bigcup_{b \in CFG_j} UCB_b^j \tag{2.7}$$

Let us assume T_{MISS} to be a constant value representing the maximum latency associated with each cache miss. Busquets defines the following value:

$$\gamma_{i,j}^{bus} \stackrel{\text{def}}{=} T_{\text{MISS}} \times |ECB| . \tag{2.8}$$

The value $\gamma_{i,j}^{bus}$ encodes the maximum pre-emption delay which each job from task τ_j can induce on tasks in the set $\{\tau_{j+1}, \cdots, \tau_i\}$.

The response time equation is then written as:

$$R_i^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \times (C_j + \gamma_{i,j}^{bus}) \tag{2.9}$$

Mitra provided a simple mechanism to compute the maximum pre-emption delay which a pre-empting task τ_j may cause on a pre-empted task τ_i for a single pre-emption [NMR03]. The value $CRPD(\tau_j, \tau_i)$ is defined as:

$$CRPD_{j,i} \overset{\text{def}}{=} T_{\text{MISS}} \times \max_{b \in CFG} \{ \left| UCB_i^b \bigcap ECB_j \right| \}. \tag{2.10}$$

This means that the maximum pre-emption delay a τ_j may cause on τ_i for a single pre-emption occurs on the BB_b of task τ_i such that the intersection of the set UCB_i^b associated to the BB_b of τ_i has the maximum number of common elements with ECB_j of task τ_j.

The CRPD bound is then integrated with the schedulability test in the following way:

$$R_i^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \times (C_j + \sum_{l=i}^{j+1} CRPD_{j,l}) \tag{2.11}$$

The sum of the CRPD is performed over the set of lower priority tasks $\{j+1, \cdots, i\}$ in order to account for the chained pre-emptions where a task τ_j can evict useful cache lines from more than one lower priority task.

An extension of this work is presented by Tan, where chained pre-emption are accounted for [TM07]. Tan defines the maximum pre-emption delay a task τ_j may cause in the lower priority schedule to be:

$$\gamma_{i,j}^{tan} \overset{\text{def}}{=} T_{\text{MISS}} \times \left| \left(\bigcup_{l \in \{j+1, \cdots, i\}} MUMBS_l \right) \bigcap ECB_j^{WMP} \right|. \tag{2.12}$$

The maximum set of Useful memory blocks ($MUMBS$) as is a term coined by Tan which denominates a union of all the UCB sets of every program point of the analysed task.

$$MUMBS \overset{\text{def}}{=} \bigcup_{p \in P} UCB_p \tag{2.13}$$

Where P is the set of program points of a given task which have a UCB set defined for. Again for instruction caches it suffices to consider the CFG at BB granularity. The ECB^{WMP} set is in turn the largest set of memory references mapping to each cache line along a single path of the pre-empting task, which is the worst case $MUMBS$ path (WMP).

The safety of the approach considering a single path for the pre-empting task ECB relies on the over-approximation taken in the MUMBS definition.

Altmeyer has adapted Tan's work in order, dropping the ECB^{WMP} per path concept and considering a unified ECB i.e. enclosing all memory references occurring in any possible pre-empting task's path [AB11]. As for the UCB it only considers the UCB per program point [AB11]. This yields:

$$\gamma_{i,j}^{tan} \overset{\text{def}}{=} T_{\text{MISS}} \times \max_{\{b_i \in CFG_i, \cdots, b_{j+1} \in CFG_{j+1}\}} \left\{ \left| \left(\bigcup_{l \in \{j+1, \cdots, i\}} UCB_l^{b_l} \right) ECB_j \right| \right\}. \tag{2.14}$$

Both CRPD bounds may be used in the RTA present in Equation 2.9.

As a later contribution Altmeyer proposed a CRPD bound considering chained pre-emptions:

$$\gamma_{i,j}^{alt} \overset{\text{def}}{=} T_{\text{MISS}} \times \max_{k \in \{i, \cdots, j+1\}} \left\{ \left| \max_{b \in CFG_k} \left\{ UCB_k^b \cap \left(\bigcup_{h \in \{1, \cdots, j\}} ECB_h \right) \right\} \right| \right\} \tag{2.15}$$

Another less complex algorithm in comparison to Lee et al. [LHS$^+$98] resorting to RTA was presented by Petters and Färber [PF01]. In this work an iterative algorithm is used which considers the worst-case pre-emption delay generated in a level-i schedule but without resorting to the ILP formulation devised by Lee et al. [LHS$^+$98]. The fixed point algorithm for the RTA computation is nevertheless similar. The variable $\Delta_{i,j}(t)$ as defined by Petters encodes the maximum pre-emption delay task τ_i suffers from all jobs of task τ_j released in a given time interval of length t. First, Petters considers the tasks $\tau_l \in \{j-1, \cdots, i\}$ set which exhibits the highest pre-emption delay penalty. The pre-emption cost of task τ_k is then accumulated for the cumulative number of times jobs task τ_l may be pre-empted by jobs from τ_k in the interval of length t. The $\Delta_{i,j}(t)$ computation for each RTA iteration finishes when $\left\lceil \frac{t}{T_j} \right\rceil$ pre-emptions have been considered. Which means that the exact upper-bound on the number of releases of τ_j in the interval being analysed are considered. This is an inherently UCB based analysis. The RTA analysis is then written as:

$$R_i^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \times C_j + \Delta_{i,j}(R_i^{k-1}). \tag{2.16}$$

Staschulat and Ernst built upon Petters' work providing a solution for CRPD integration with schedulability test which incorporates the ECB information as well [SE04]. In this work varying pre-emption costs were considered. The RTA analysis is rewritten as:

$$R_i^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \times C_j + \Delta_{i,j}^{sta}(R_i^{k-1}). \tag{2.17}$$

Where the $\Delta^{sta}_{i,j}(t)$ parameter denotes the pre-emption delay caused by jobs from task τ_j in a given time window of length t defined as:

$$\Delta^{sta}_{i,j}(t) \overset{\text{def}}{=} T_{\text{MISS}} \times \sum_{l=1}^{n}(t) \left| \overset{l}{\max}(M) \right| . \tag{2.18}$$

The function $\max^h()$ returns the h^{th} biggest value in the set.

The function $n(t)$ states the maximum number of jobs from tasks in the set $\{j, \cdots, i-1\}$ which can be released in an interval of length t, i.e.:

$$n(t) \overset{\text{def}}{=} \sum_{l=j}^{i-1} \left\lceil \frac{t}{T_l} \right\rceil . \tag{2.19}$$

The variable M is defined as:

$$M \overset{\text{def}}{=} \uplus^{i}_{k=j+1} \uplus^{\left\lceil \frac{t}{T_k} \right\rceil}_{m=1} \widehat{\delta_{j,k}} \tag{2.20}$$

Where $\widehat{\delta_{j,k}}$ is a set containing exactly $\left\lceil \frac{R_k}{T_l} \right\rceil$ elements which are the highest CRPD values associated to consecutive pre-emptions by jobs from τ_j on a single τ_k job. Hence $\widehat{\delta_{j,k}}$ is defined by Staschulat as:

$$\widehat{\delta_{j,k}} \overset{\text{def}}{=} \left\{ \delta^1_{j,k}, \cdots, \delta^{\left\lceil \frac{R_k}{T_l} \right\rceil}_{j,k} \right\} \tag{2.21}$$

where $\delta^1_{j,k} = \max^1_{p \in CFG}\{UCB^p_k \cap ECB_j\}$, $\delta^2_{j,k} = \max^2_{p \in CFG}\{UCB^p_k \cap ECB_j\}$ and generically $\delta^h_{j,k} = \max^h_{p \in CFG}\{UCB^p_k \cap ECB_j\}$.

The complex formulation presented by Stachulat enables the consideration of variable pre-emption costs through the execution of each job, though no ordering of occurrences is considered.

All of the CRPD aware schedulability tests presented thus far are for the fixed task priority scheduling policy. Another work exists integrating CRPD into the schedulability analysis of EDF. This consist of demand-bound function based procedure proposed by Ju et al. [JCR07]. Ju, as Mitra did, considers the pairwise pre-emption delay $CRPD_{j,i}$ which encodes the maximum CRPD penalty a job from task τ_j may induce in a job from τ_i. Let us define the demand bound function of a given task (dbf) [Bar05]:

$$dbf(\tau_i, t) \overset{\text{def}}{=} \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \times C_i \tag{2.22}$$

Ju extends this definition by considering the C_i of the task and the maximum pre-emption delay it is subject to as an upper-bound for its workload, which yields:

$$dbf^{Ju}(\tau_i,t) \overset{\text{def}}{=} \left\lfloor \frac{t-D_i}{T_i} \right\rfloor \times (C_i + \sum_{j\in H_i} n_{i,j} CRPD_{j,i}). \tag{2.23}$$

In Ju's formulation H is the set of tasks of smaller relative deadline than τ_i, which are the only tasks which can in fact pre-empt jobs from τ_i and $n_{i,j} = \left\lceil \frac{D_i - D_j}{T_j} \right\rceil$ upper-bounds the maximum number of times jobs from τ_j may pre-empt a single job from τ_i.

EDF successfully schedules the task-set if the following condition is verified:

$$\forall t \in L, \sum_{\tau_i \in T} dbf^{Ju}(\tau_i,t) \leqslant t \tag{2.24}$$

Note that this schedulability test is sufficient but no longer necessary when CRPD is considered in the task-system due to the overestimation involved in the pre-emption delay estimation.

2.4 Limited Pre-emptive Scheduling

Non-pre-emptive scheduling has its benefits. Besides completely removing the problem of pre-emption delay, it schedules some task-sets that would not otherwise be schedulable under fully pre-emptive fixed priority (FP) [WS99] and enables considerable memory savings by allowing for the existence of a single stack of size equal to the maximum stack requirement by any task that compose the task-set. Nevertheless non-pre-emptive scheduling is not always a feasible solution to resort to.

Fully pre-emptive scheduling has its own drawbacks since there exists an inherent difficulty in assessing the worst-case number of pre-emptions in these policies. This may lead to overly pessimistic analysis.

A different type of scheduler which does not lie in either extremes (fully pre-emptive or non-pre-emptive) allows for the encompassing of the benefits observed in both extremes.

This can be achieved by thoughtfully restricting the pre-emptions, either by scheduler aid or by using specific pre-emption points in the code. The framework offers a viable ground over which to address the problem of pre-emption delay estimation. Scheduling the workload with such sort of policies enables more information to be present at the design time with respect to where the pre-emptions will occur in a given system execution.

The sole purpose of a scheduling algorithm is to determine which tasks execute on the target processor at each point in time. A scheduling algorithm can be, non-ambiguously, described through a function $\mathscr{A}$. $\mathscr{A}$ takes as an input the set of active tasks and knowledge of future task releases when available. The output of $\mathscr{A}$ is a partially ordered set of tasks. The m highest priority tasks are then executed upon the available processors.

The scheduling algorithms are generally classified according to the $\mathscr{A}$ behaviour with respect to the priority ordering.

1. **fixed task priority** each job is assigned the same fixed priority
2. **fixed job priority** each job is assigned a fixed priority which is not necessarily the same as previously or subsequently released jobs (from the same task)
3. **dynamic job priority** the priority of the job varies during its frame (between its absolute release time and absolute deadline)

It is easy to observe that algorithms of type 1 are a subset of type 2 which in turn are a subset of type 3. The algorithms of the subsets are associated to lower run-time complexity when compared to the ones on the complement set which yields the super-set.

Generally these algorithms tend to dispatch onto the execution platform the highest priority workload which is available to be executed at any instant in time.

Any scheduler for which a task pre-empts another running in the system immediately before a given time instant t when the pre-empting task had a priority increase that surpasses the pre-empted task can be termed as a fully pre-emptive scheduler.

In informal terms, limited pre-emptive schedulers can be seen as the product of slight modifications to the so called fully pre-emptive schedulers such that the pre-emptions do not generally occur synchronously with the release or increase in the priority level of some workload but in a more controlled manner after, for instance, a known time interval elapses. At run-time some pre-emptions which would generally occur in the fully pre-emptive schedule are now said to be deferred in the limited pre-emptive model.

The more common limited pre-emptive scheduling algorithms existing in literature, although being generally referred to as extensions to type 1 and 2 schedulers, can in fact be considered to be type 3 algorithms. The pre-emption deferral can in fact be seen as a temporary priority increase from the part of the executing task. Another analogy, which is more commonly referred to is that whenever a pre-emption is being deferred, the task which remains executing is in fact executing – for a limited time interval – in a non-pre-emptive manner.

The mechanism of pre-emption deferral has a number of advantages as has been pointed out in several works [YBB10, YBB09, BBM$^+$10]. These scheduling policies present a trade-off between the extremes of fixed priority non-pre-emptive and fully pre-emptive scheduling. Gang Yao et al. provide a comparison of all the available methods described so far in literature [YBB10].

Aside for enabling heightened sense of where pre-emptions occur, the limited pre-emptive model also allows for schedulability increases. An example is provided in task-set

	C	T
τ_1	2	10
τ_2	9	12

Table 2.1: Limited Pre-emptive Motivating Task-set

shown in table 2.1 where the task fails to be schedulable under fully pre-emptive fixed task priority but it is in fact schedulable by non-pre-emptive scheduling. For this task-set, the fully pre-emptive schedule is displayed in Figure 2.2 to the right. One possible limited pre-emptive schedule which ensures schedulability of the task-set is displayed in the same figure to the left. It is apparent that task τ_2 misses a deadline at $t = 12$ since it still has one unit of pending workload at this deadline. In the exemplified limited pre-emptive schedule all tasks meet their deadlines until $t = 24$ and hence it is concluded that the task-set is schedulable with this policy since at exactly $t = 24$ there is virtually no pending workload left to process in the system.

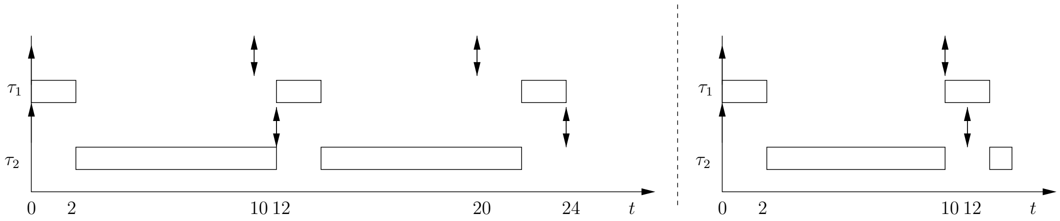


Figure 2.2: Limited Pre-emptive Schedule vs Fully Pre-emptive Schedule (task-set in Table 2.1)

The limited pre-emptive scheduling algorithms can be categorized into two broad groups depending on the nature and properties of the non-pre-emptive regions. This two groups are denominated as:

- Fixed non-pre-emptive;
- Floating non-pre-emptive.

2.4.1 Floating Non-pre-emptive Regions

In the regular floating non-pre-emptive region scheduling, each task τ_i has a maximum deferral time defined which will be denoted by the parameter Q_i. When a task τ_i is executing and τ_i is not the highest priority task in the ready queue, then it is said that a pre-emption deferral chain is occurring. A pre-emption deferral chain, as is shown in Figure 2.3 starts with a higher priority release, and lasts at most Q_i time units. At the end of a pre-emption deferral chain a pre-emption invariably happens. In Figure 2.3 task τ_i is executing when a job from τ_j is released, at some time in between a job from τ_k is released. The deferral chain ends exactly Q_i time units after the first higher priority release.

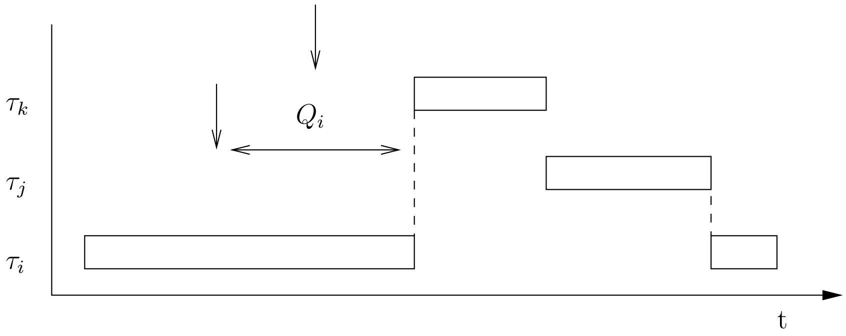


Figure 2.3: Floating Non-pre-emptive Region Scheduling Example

All the scheduling policies presented so far were created with the intention of enabling the floating non-pre-emptive regions usage. Effectively by extending the schedulability of fixed task priority the overall admissible blocking tolerance of higher priority tasks will increase in accordance. A practical consequence is that the allowed non-pre-emptive regions will increase in size in comparison to the limits considered by the schedulability test which does not contemplate the schedulability increase (i.e. the fully pre-emptive condition). Having bigger values for the non-pre-emptive regions is obviously advisable since this will inevitably allow for a reduction on the number of pre-emptions, and will enable less pessimism in the pre-emption delay computation [MNPP12a]. In the simple ready queue locking, and in the ready queue locking with pre-emption threshold maximum blocking times admissible for all the tasks are computed. This information alone enables the usage of the floating non-pre-emptive regions scheduling.

2.4.2 Fixed Non-pre-emptive Regions

The mechanism of restricting pre-emptions was first proposed by Burns et al. [Bur95] where only fixed non-pre-emptive regions are considered (*fixed pre-emption points*).

The fixed non-pre-emptive region model differs from the previously mention (floating) in that the pre-emption points are restricted to occurring at very well defined points in the task code.

In this work it is referred that dividing the tasks into a set of consecutive non-pre-emptive jobs would be a interesting way of increasing the schedulability of the task-sets. This initial work suffered from the shortcoming of only considering the first frame of a task in a synchronous release situation to account for the critical situation.

Keskin et al. discuss the theory of fixed non-pre-emptive region schedulability [KBL10]. The author deemed the available test [Bur95] optimistic, arguing that under no assumptions the worst-case response time for a job of task τ_i may no longer arise in the first critical region in a synchronous release situation but that it may show up in a job k of task τ_i in the level-i active period generated at a synchronous release situation.

Until this point, the answer to what the maximum length of the non-pre-emptive regions would be had not been found yet. A basic ingredient for this is identifying the amount of slack in the higher priority schedule. Slack computation was the subject of some attention in the past [DTB93, LRT92]. These works mainly deal with the detection of slack in the schedule that enables the execution of aperiodic tasks with low priorities to execute uninterrupted. This benefits the latency of those applications considerably. The drawback of these methods is that they either rely on off-line analysis and the periodic behaviour of hard deadline tasks [LRT92] or on the on-line computation of the slack using methods with variable execution (recursive method) time and high complexity [DTB93]. The previously mentioned works do not address the issue of pre-emption reduction nor consider slack stealing on a purely hard real-time system. A similar point of view is proposed by Dobrin et al. [DFP01]. In this work an off-line analysis parses the schedule identifying pre-emptions. It then tries to remove these by changing tasks priorities and the offsets without jeopardizing the schedulability. This method is solely applicable to periodic task-sets though. Slack computation in a schedule is the base theory for computing the maximum allowed blocking times for a given set of tasks. The maximum admissible blocking time depends on the task-set and the priority relation between the tasks and the scheduling policy. The first maximum admissible blocking time computations were proposed by Baruah [Bar05] for EDF scheduling policy, and later by Yao et al. [YBB09] for fixed task priority scheduling mechanism. In the initial work Baruah defined a maximum length for the non-pre-emptive chunks of each task. At the deadline of the first job of each task τ_j, in a synchronous release with all other higher priority tasks, the slack (β_j) is computed:

$$\beta_j \overset{\text{def}}{=} D_j - \sum_{h \in \{1,\cdots,j-1\}} dbf(D_j, \tau, h). \tag{2.25}$$

The value β_j denotes the maximum amount of blocking time the task would endure without missing a deadline. The maximum length of the non-pre-emptive regions of each task τ_i is referred to as Q_i which is defined as:

$$Q_i \overset{\text{def}}{=} \min_{j \in \{1,\cdots,i-1\}} \beta_j \tag{2.26}$$

This ensures that task τ_i never induces a bigger blocking time to higher priority jobs from task τ_j where $j \in \{1,\cdots,i-1\}$ such that they would miss a deadline. Note that in this situation tasks are ordered by increasing relative deadlines $\forall i \in \{1,\cdots n-1\} D_i \leqslant D_{i+1}$. The priority of the workload is a function of each jobs absolute deadline (EDF [LL73]).

This result was then further extended by Baruah and Bertogna in order to consider a variable length for the non-pre-emptive regions of each task [BB10]. In this work a monotonically decreasing function is considered $Q(t)$.

$$Q(t) \overset{\text{def}}{=} \min_{t' \in [0,t]} t' - \sum_{i \in \{1,\cdots,n\}} dbf(t', \tau_i) \tag{2.27}$$

Assume that at time t_2, a job from task τ_i which was released at time $t_1 < t_2$, is executing on the processor. Further assume that at time $t_2 - \varepsilon$ the job from τ_i is the highest priority active job in the system and that at time instant t_2 task τ_j releases a job. At this event the job from task τ_i will start executing non-pre-emptively for $Q(t_1 + D_i - t_2)$. It is easily shown that $Q_i \leqslant Q(t_1 + D_i - t_2)$, hence this extension enables bigger non-pre-emptive regions for each task at the cost of added run-time overhead. The authors propose the usage of a table to implement this mechanism and observe that the number of table elements is generally quite small.

A fixed priority scheduling method has been devised by Yao et al. [YBB09]. Yao defines an upper bound on the maximum admissible blocking time each task may suffer as:

$$\beta_i \overset{\text{def}}{=} \max_{t \in [0,D_i]} t - \sum_{j \in \{1,\cdots,i\}} rbf(t, \tau_j), \tag{2.28}$$

Where

$$rbf(t, \tau_i) \overset{\text{def}}{=} \left\lceil \frac{t}{T_j} \right\rceil \times C_j. \tag{2.29}$$

The maximum length of the non-pre-emptive regions is then defined as:

$$Q_i \overset{\text{def}}{=} \min_{j \in \{1,\cdots,i-1\}} \beta_j \tag{2.30}$$

The computed value is the maximum allowed for systems scheduled with floating non-pre-emptive regions but it is not optimal when fixed non-pre-emptive regions are considered. Still this only allows the scheduling of tasks that were schedulable under the fully pre-emptive model. The author proves that if the task-set is schedulable under fully pre-emptive fixed priority, the job of task τ_i with worst-case response time will still be the first job in the synchronous release situation.

Later the work was extended with fixed non-pre-emptive regions in mind [YBB11a]. Since the last part of τ_i executes non-pre-emptively for a maximum duration of Q_i time units. The maximum admissible blocking time per task is then rewritten as:

$$\beta_i \overset{\text{def}}{=} \min_{k \in \{0,\cdots,K\}} \left\{ \max_{t \in [k \times T_i, k \times T_i + D_i - Q_i]} \left\{ t - \sum_{j \in \{1,\cdots,i-\}} rbf(t, \tau_j) + C_i - Q_i \right\} \right\}, \tag{2.31}$$

where k upper-bounds the maximum number of jobs τ_i has in a level-i busy period blocked by the maximum admissible time.

Let us assume $L_i(B)$ to be the maximum length of the level-i busy period blocked for B time units,

$$L_i(B) \overset{\text{def}}{=} \min\{t|t-(B+\text{rbf}(\Gamma_i,t))=0\}. \tag{2.32}$$

Then K may be written as:

$$K \overset{\text{def}}{=} \left\lceil \frac{L_i(Q_i)}{T_i} \right\rceil, \tag{2.33}$$

since the $level-i$ busy period can never be blocked by more than Q_i time units.

In the initial work [YBB09] the regular fully pre-emptive schedulability test was employed. In this case the synchronous release of all higher priority tasks constitutes the worst possible case from the point of view of higher priority workload interference. As the work was extended to take into advantage the schedulability gains possible with the fixed non-pre-emptive region model, a new schedulability test was devised [YBB11a]. In this new test not only the first frame of each task in a synchronous release situation had to be checked for deadline misses. Instead all the frames in the busy period commencing with a synchronous release need to be checked to ensure all deadlines are met. This is tied to the fact that, with this analysis, task-sets which were not schedulable under fully pre-emptive fixed task priority might be so. Hence, due to the larger non-pre-emptive regions, some workload might be postponed to execute in a future time interval when compared to the interval observed with fully pre-emptive fixed task priority scheduling.

The pre-emption delay estimation problem using fixed non-pre-emptive region scheduling was presented by Bertogna et al. [BBM$^+$10]. In order to reduce CRPD, the usage of fixed non-pre-emptive areas of code is proposed. There exists an extensive set of possible pre-emption points. For each pair of points a pre-emption delay value is defined. Bertogna et al. propose in their model a mechanism to choose a set of points as actual pre-emption points such that, the worst case execution time distance between each chosen pair of points, considering pre-emption delay is never greater than Q_i time units, otherwise higher priority task's temporal requirements might not be met. This work may be used both with fixed task priority as well as with EDF, the Q_i parameter considered for each scheduling policy is the one defined in [YBB11a] and [Bar05] respectively.

An extension of this work was presented by Bertogna et al. in [BXM$^+$11], where the optimal set points is selected with the aim of minimizing the pre-emption delay the task may suffer overall. This has the limitation of requiring manipulation of the code of tasks and thus is not very amenable to system developers. Also, it is not straightforward to take into account tasks with complex control-flow graphs [BXM$^+$11]. Additionally it can not be easily applied in situations where the task-sets are subject to run-time changes, as the maximum distance between pre-emption points is defined by the higher priority workload.

A different mechanism to reduce pre-emptions was proposed by Express Logic [Lam] termed pre-emption threshold. In this work a task may only pre-empt another if its priority is bigger than that task's pre-emption threshold. Wang and Saksena provided an optimal priority assignment for pre-emption-threshold scheduling policy [WS99]. The pre-emption thresholds are computed by aid of a search algorithm that will test several possibilities until it either reaches a solution that ensures schedulability for the given task-set or fails. The pre-emption-threshold values are the sufficient for ensuring schedulability. Later the work was extended [SW00] by the same authors to assign the pre-emption threshold to the highest possible priority value which maintains schedulability.

2.5 Temporal Isolation Enforcement

Before a safety-critical system can be deployed and marketed, a certification authority must validate that all the safety-related norms are met. All the components comprising that system (the software, the hardware, and the interfaces) are scrutinized to ensure conformance to safety standards. Timing guarantees must be derived at design time and consequently enforced during run-time for the system to be certified. These timing guarantees are obtained through timing and schedulability analysis techniques, which are typically more accurate and simpler when *spatial and temporal isolation between tasks* is provided. This is because timing analysis techniques must thoroughly examine every shared resource and identify a worst-case interference scenario in which the analysed task incurs the maximum delay before accessing the shared resource[s]. Without a proper isolation between the tasks: First, the number of interference scenarios to be explored may be out of proportion, hence compelling the analysis techniques to introduce an undesired pessimism into the computation by over-approximating these scenarios. Secondly, having a high number of possible interference scenarios naturally increases the probability of encountering a "pathological" case, where the delay incurred by the analysed task in that particular scenario is far higher than in the average case. Since the analysis tools *must* capture the *worst-case scenario*, this pathological case will be retained and used in higher-level analyses (like schedulability analyses) which are built upon the results of timing analyses (thus propagating the pessimism all the way up to system-level analyses).

A first step in the certification process is to categorize each component (software and hardware) by its level of criticality and assign a unitary safety integrity level[1] (SIL) to all components. When integrated in the same platform the components of different SILs share low-level hardware resources such as cores, cache subsystems, communication buses, main

[1] A Safety Integrity Level (SIL) is defined as a relative level of risk-reduction provided by a safety function, or to specify a target level of risk reduction. In simple terms, a SIL is a measurement of performance required for a safety instrumented function.

memory, etc. To provide the required degree of “sufficient independence” between components of different SILs, Industry and Academy have been seeking solutions for many years to (1) render the components of a specific SIL as independent and isolated as possible from the components with different SILs and (2) upper-bound the residual impact that components of different SILs may have on each other after the segregation step, with the primary objective of certifying each subset of components at its own SIL.

Generally the sufficient separation is achieved in regular system model by resorting to specific reserves which strictly dictate the rate of access to a given platform by the workload. Reservation-based systems are quite a mature topic in real-time literature. Sporadic servers [AB04, SSL89] are proposed in real-time literature to ensure temporal isolation in fixed task priority systems. Each server reserves a budget for the task execution. A task can only execute if the server budget is not depleted. This ensures that the interference generated by a task in the schedule cannot exceed what is dictated by its server parameters and servers are the scheduled entities.

Solutions for temporal isolation in Earliest Deadline First (EDF) also exist employing the sporadic server concept, namely constant bandwidth server (CBS) [AB04] and Rate Based Earliest Deadline First (RBED) [LKPB06]. Each server has an absolute deadline associated to it which acts as the server priority. On top of the temporal isolation properties these frameworks also employ budget passing mechanisms which enhance the average-case response time of tasks in the system without jeopardizing the temporal guarantees [NP08, LB05].

The previously employed execution models assume that the interference between workload occurs solely on the CPU. As it turns out, if other architectural subsystems are shared in the execution platform which present some state with non-negligible state transition times (e.g. caches), interference between task will be created (commonly referred to as pre-emption delay). The maximum pre-emption delay any task may endure may still be integrated into the task’s budget[2], this solution is shown in this thesis to be subject to *heavy pessimism*. *When minimum inter-arrival times of tasks cannot be relied upon* (i.e. tasks release jobs at a faster rate than computed at design time), the previously mentioned frameworks *fail* to ensure temporal isolation between concurrent tasks in the system.

In order to ensure temporal isolation cache partitioning may be employed [BCSM08]. This technique has the disadvantage of decreasing the usable cache area available to each task, and as a consequence impacting its performance. Other architectural subsystems exist (e.g. TLB, dynamic branch predictors, etc.) which cannot be partitioned in order to remove the interference source between tasks. Recently an approach has been proposed where, when a task starts to execute it stores onto the main memory all the contents of the

[2] we assign one server per task, the task and server term is used interchangeably in this document

cache lines it might potentially use [WA12]. After the pre-empting task terminates its execution it loads back from memory all the memory blocks that it has stored in the cache at its release. This indeed ensures temporal isolation among different applications but has several drawbacks. It unnecessarily moves all the memory blocks to main memory which reside in cache lines it might use even if the actual execution does not access them. This mechanism significantly increases memory traffic which may be troublesome in multicore partitioned scheduling due to increased contention on the shared memory bus. In comparison our approach only passes budgets between servers and hence this budget is only used if it is required. As a last limitation of [WA12] it cannot cope with scenarios where a given task does not *respect the minimum inter-arrival contract part*.

As a last resort non-pre-emptive scheduling policies may be employed to ensure temporal isolation in platforms with non-negligible pre-emption delay. By nature, these are not subject to any pre-emption delay overhead. As discussed previously, this brings the potential of severely decreasing the ability to schedule a large portion of task-sets.

Chapter 3

Extensions to the Limited Pre-emptive Model

This chapter delves into the extended theory on limited pre-emptive scheduling. Firstly a mechanism to extend the floating non-pre-emptive region length exploiting the knowledge of the workload demand available at run-time is presented in section 3.1. This approach was presented in a prior work [MP11]. Furthermore, a work extending the schedulability of fixed task priority for single core is described. A concept termed ready-queue locking is introduced in section 3.4. This scheduling policy is integrated with pre-emption threshold and with floating non-pre-emptive regions. This was previously published in [MPB12].

3.1 On-line FTP Floating Non-Pre-emptive Region Extension

Pre-emptive schedulers, compared to non-pre-emptive ones [Bur95, GMR00, WS99], introduce time-overheads during the execution of the system, due to context switches and the loss of working sets in the caches and similar architectural sub-systems, generated by some pre-empting task [SE04, AMR10, RM06b]. Pre-emption delays are generally considered, in the pure real-time scheduling theory, to be null or negligible whereas in actual systems this assumption does not hold. The main motivation for resorting to a simpler model is that it allows for easier reasoning and presentation of the issues related to the core interest of these works, which is that of the scheduler behaviour and properties. Depending on the execution platform the pre-emption delay overheads may turn out to be quite significant [DD10]. The actual incorporation of these quantities in the schedulability test can be associated to large over-estimations. The main contributor to this exaggeration in its accounting is associated to the lack of information on the actual worst-case scenarios. The increased over-estimations lead to task-sets being classified as unschedulable where, in fact, no deadline could be missed. There is an inherent complexity of estimating the CRPD and also tightly bounding the number of pre-emptions in fully pre-emptive systems. A way to ease the task of quantifying pre-emption overhead is to introduce restrictions on the pre-emptions. Two methods may be exploited in this manner. The first, *fixed non-pre-emptive regions*, relies on specific pre-emption points inserted into the task's code. This has to be performed at design-time, relying on WCET estimation tools [WEE$^+$08] that can partition the task into non-pre-emptible sub-jobs [BBM$^+$10]. As previously stated, this approach is highly restrictive since these points cannot be easily replaced. As a consequence, scenarios where task-set changes are foreseeable (e.g. mode-changes [MRNP11]) cannot cope with the fixed non-pre-emptive model. Also code reutilization would require a recertification of its properties. Aside from these more mundane considerations a more fundamental one lies in the difficulty of pre-emption point placement into non-trivial control-flow graphs. Since the temporal distance between consecutive pre-emption points must be smaller than a given value even in the worst-case scenario the average number of pre-emptions during actual system execution may be similar to the fully pre-emptive scenario in platforms where a big variation between the worst and the average-case behaviour exists.

The second limited pre-emptive model, *floating non-pre-emptive regions*, is implemented by disabling pre-emptions for a certain amount of time promptly decided during system execution. This approach solely relies on the computation of the maximum time each task can execute non-pre-emptively. These can be changed at run-time if the task-set changes. The presented work will only address the *floating non-pre-emptive regions* model. This is the only model suited for use in open-systems concept [DL97] or multi-mode scenarios [SKKC00], where task-sets may vary at run-time. It also has the distinct advantage that no code changes are required in the application in order to implement it.

In this section the reduction of the observed number of pre-emptions by dynamically delaying pre-emptions is investigated. This work exploits the maximum admissible blocking times of tasks (alternatively termed as maximum admissible pre-emption deferral) and takes advantage of the existing relative task release phasing at run-time. The maximum admissible blocking time is defined for every task τ_j and is denoted by $\beta_j(t)$. It represents a lower bound on the time span for which a job from a task τ_j can be blocked by lower priority workload without incurring a deadline miss. This quantity, termed $\beta_j(t)$, is both a function of the scheduling policy (FTP in this work) and the priority ordering of the task-set. Additionally the quantity $\beta_j(t)$ is also a function of the currently pending higher priority jobs. In other words $\beta_j(t)$, as it is computed in this work, takes into the account the knowledge about arrival phasing of the higher priority jobs. The times tasks execute non-pre-emptively are,as a consequence, larger than for mechanism relying on pre-computed non-pre-emptive region lengths [BBM$^+$10].

In this section $r_i^l(t)$ represents the absolute time instant of the last release of a job of task τ_i with pending workload before (or at) time t. If there is no pending workload from task

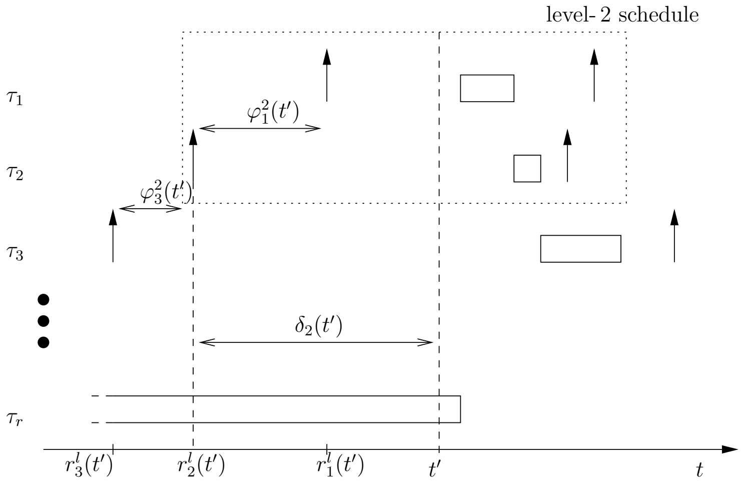


Figure 3.1: Model Notation

τ_i at time t then $r_i^l(t) = t$. The value $\delta_i(t)$ denotes the amount of time elapsed since $r_i^l(t)$, i.e., $\delta_i(t) \stackrel{\text{def}}{=} t - r_i^l(t)$. Then, let us define by $\varphi_j^i(t) = r_j^l(t) - r_i^l(t)$ the *task-relative* offset of task τ_j (in relation to task τ_i at time t), and by $\boldsymbol{\varphi}^i(t)$ the set of all $\varphi_j^i(t)$ such that $\tau_j \in hp(i)$. $\boldsymbol{\varphi}^i(t)$ represents the vector of the offsets of higher priority releases of tasks in relation to τ_i. To clarify, this means that all the offsets are considered in relation to $r_i^l(t)$ which is the time instant of the last release of a job from task τ_i. If $r_j^l(t) < r_i^l(t)$ then, at time instant t, the last release of τ_j preceded the last release of τ_i. If there is no pending workload from task τ_j at time t, this implies $\varphi_j^i(t) = \delta_i(t)$ since there is no knowledge about future releases of jobs from τ_j. Hene a worst-case scenario is considered, which equates to τ_j relesasing a job at the current time instant $t = r_i^l(t) + \delta_i(t)$ The stated notations are clarified in Figure 3.1.

3.1.1 Admissible Pre-emption Deferral

Every task can endure a pre-emption deferral which solely depends on the amount of higher priority workload that will need to be executed in the future, as will be shown later in this section. Firstly, a few concepts taken from related work are introduced.

Definition 2 [**level-i schedule**]: *The schedule composed of jobs from task τ_i and jobs from tasks with higher priority is denominated as* level-i *schedule [KBL10].*

The reader may find an example for a level-i in use in the graphical representation provided in Figure 3.1. The amount of idle time that will exist in the level-i schedule is computed as a function of the known previous higher priority releases that are still deferring.

$$W_i(t,\boldsymbol{\varphi}^i(t)) \overset{\text{def}}{=} C_i + \sum_{j \in hp(i)} \text{rbf}(\max(t-\varphi^i_j(t),0),\tau_j) \tag{3.1}$$

where

$$\text{rbf}(t,\tau_j) \overset{\text{def}}{=} \left\lceil \frac{t}{T_j} \right\rceil \times C_j. \tag{3.2}$$

Equation (3.1) [LSD89b] gives us the amount of pending workload in the level-i schedule that was released up until time instant t and is being deferred (if higher priority constraints enable so). By computing the difference between Equation (3.1) and the time progression line for every point in the given time interval $[0,\delta_i(t)]$, chosing the maximum of such values, the amount of idle time in the schedule for the specified time interval is obtained. This may be formaly written as

$$\beta_i(\delta_i(t),\boldsymbol{\varphi}^i(t)) \overset{\text{def}}{=} \max_{t' \in [0,\delta_i(t)]} (D_i - t' - W_i(D_i - t', \boldsymbol{\varphi}^i(t))). \tag{3.3}$$

The intuition behind the Equation (3.3) is depicted in Figure 3.2 for a level-3 schedule. Figure 3.2 is composed by three plots, the beginning of the referential is $r^l_3(t)$ which is the time instant of the release of the job of τ_i considered in this example. Without loss of generality $r^l_3(t)$ may equate to 0 (i.e. $r^l_3(t) = 0$). The top graph depicts Equation (3.1) together with the time progression line. The maximum difference between the time line and the function defined by Equation (3.1) at every interval $[0,\delta_i(t)]$ (where $t \in [r^l_i, r^l_i + D_i]$) is equal to the amount of time when there was no pending workload in the system to be processed in that cumulative interval. This is apparent by observing the schedule chart in the middle of Figure 3.2.

The bottom part plot in Figure 3.2 displays the amount of idle time available in the level-3 schedule at the release time of a job from τ_3 and its evolution from when $\delta_i(t) = 0$ as it progresses towards $\delta_i(t) = D_i$, as the workload of task τ_3 gets deferred and the higher priority workload shifted for $\delta_i(t)$ time units as well. From the point of view of the current active job of task τ_3, as it is deferring its pre-emption in time, the function $B_3(\delta_3(t),\boldsymbol{\varphi}^3(t))$ in (3.3) denotes the lower bound imposed by task τ_3 on the blocking amount that the $level-3$ schedule can withstand at time t. In the same figure a situation where no higher priority releases have occurred prior to $r^l_3(t)$ is displayed intentionally in order to provide a clearer example of the computation of Equation (3.3) and its evolution with time. In this specific scenario it is easily perceivable that if all higher priority workload is shifted to the right in conjunction with τ_i's job the available idle time will itself "shift" in the same manner, hence the evolution with time of the bottom plot.

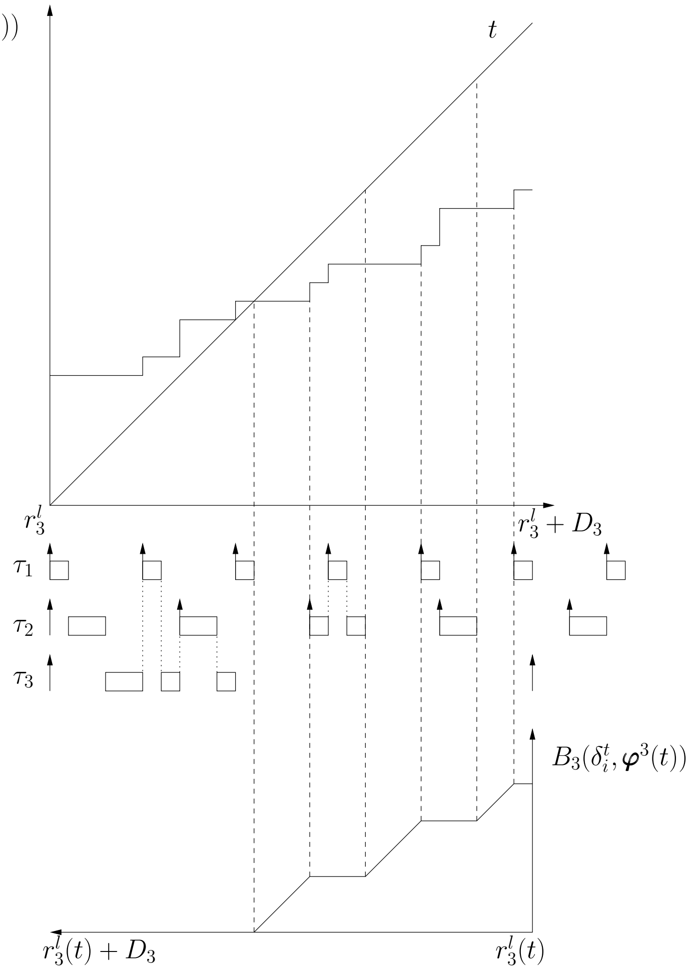


Figure 3.2: Function 3.3 with Unknown Prior Higher Priority Releases

Theorem 1. *After the release of a job of τ_i at time $r_i^l(t)$, if some lower priority task is executing the pre-emption may be safely deferred for $\beta_i(\delta_i(t), \boldsymbol{\varphi}^i(t))$ time units, without jeopardizing τ_i's deadline.*

Proof. At time instant $r_i^l(t)$ there will be at least $\beta_i(D_i, \boldsymbol{\varphi}^i(t))$ time units of idle time in the level-i schedule up until $r_i^l(t) + D_i$. If the level-i pre-emption is deferred for ε time units then $\beta_i(\delta_i(t), \boldsymbol{\varphi}^i(t)) - \varepsilon$ time units of idle time would be available for the level-i schedule at time $> r_i^l + \varepsilon$. At the earliest time instant t'' which makes $\beta_i(\delta_i^{t''}, \boldsymbol{\varphi}^i(t)) = 0$ no more idle time will be available in the level-i schedule until $r_i^l(t) + D_i$. From $r_i^l(t)$ to t'' the task may be safely delayed since there will always be enough time to execute completely before its deadline even if fully pre-emptive schedule would be carried out from this point onwards. □

The previous theorem handles a generic case of what was shown in [BB04b] for the synchronous arrival of higher priority workload situation. Note that Theorem 1 only refers to the amount of time a job of task τ_i may be deferred so that it does not miss its deadline. The same reasoning has to be applied to all jobs currently deferring the pre-emption so that no deadline is missed in the system. At every instant in time, while there are jobs deferring their pre-emptions $\beta_i(\delta_i(t), \boldsymbol{\varphi}^i(t)), \forall i \in S$ is computed, where $S = lep(j) \cap hp(p)$ In this case τ_j denotes the highest priority task blocked at time t, τ_p the task currently running and $lep(j)$ is the set of task of lower or equal priority in relation to task τ_j. At the time instant when there exists an i such that $\beta_i(\delta_i(t), \boldsymbol{\varphi}^i(t)) = 0$, a pre-emption occurs and all previously deferring jobs cease to defer, at which point the blocked job with highest priority is scheduled to execute. Normal fixed priority scheduling is carried out until there is a release of a task of higher priority than the currently running task. Implementing a scheduling policy following this exact methodology is clearly unrealistic since it implies a high complexity algorithm to operate at every instant in time. Some approximations may be used though. The simplifications rely on the observations described in the following text.

3.1.2 Practical Usage of Equation (3.3)

A straightforward way to exploit the knowledge provided by Equation (3.3), is to trigger a non-pre-emptive execution region for the job of task τ_l currently running, whenever a release from a higher priority job occurs. In the next step the situations where no higher priority job releases are present in the system is considered, so all the elements in vector $\boldsymbol{\varphi^i}(t)$ will be equal to the amount of time elapsed since $r_i^l(t)$. Equation (3.3) may then be rewritten as,

$$\beta_i(\delta_i(t), \boldsymbol{\varphi}^i(t)) = \beta_i(\delta_i(t)), \tag{3.4}$$

since $\forall j \in hp(i), \varphi_j^i(t) = \delta_i(t)$. The non-pre-emptive region should have a duration equal to $\min_{i \in hp(l)}(\beta_i(0))$. This approach is the one presented by Yao et al. [YBB09]. A simple low complexity build up on the previous approach would be to still consider $\beta_i(0)$ as the time a job from task τ_i may defer its pre-emption, and create a set of rules that enable the scheduler to perform better or equally well whenever the schedule is not in the worst-case scenario (synchronous release of higher priority tasks).

Property 1. *Whenever a job of task τ_i is released, if it has higher priority than the running task and no other job is already deferring its pre-emption, the scheduler may safely delay its pre-emption by $\beta_i(0)$).*

The Property is a direct consequence of the definition of the $\beta_i(\delta_i(t))$ function (Equation (3.4)). By definition if at $\delta_i(t) = 0$ there exists no pending higher priority workload, then the level-i schedule can be blocked for $\beta_i(0)$ time units. So task τ_3 may be safely delayed for that amount and always complete before the deadline as stated in Theorem 1.

Property 2. *If one or more tasks are already deferring their pre-emption and no job has higher priority than the job from τ_i, the timer is set to $timer = \min(timer, \beta_i(0))$.*

The variable “timer” denotes time instant (t’) when the highest priority blocked task is poised to pre-empt the currently running task. This variable can be defined for every time instant t as $timer = t' - t$. Henceforth the term “timer” is used to denote both this quantity as well as the architectural device responsible for automatically updating it at the rate of time progression.

The previously deferring jobs (if any exist) already set the timer in order not to miss a deadline. In case the highest priority task at time t is executing upon the platform then $timer = \infty$. If the lower priority tasks are to complete execution before deadline the minimum amount of time that all jobs may be deferred for has to be taken into consideration in this situation.

Property 3. *If in the previous situation there is at least one higher priority job already deferring its pre-emption, the timer is set to $\min(timer, \max(\beta_i(0) - (r_i^l(t) - t_0), 0)$, where t_0 is the instant in time when the first job, from the set of jobs currently blocked, was released.*

This is due to the fact that $\beta_i(0)$ represents the amount of time a job may be deferred if no other higher priority job is deferring at that instant in time. If at time $r_i^l(t)$ there is higher priority workload deferring pre-emption it was released mandatorily after t_0 (i. e. $t_0 < r_i^l(t)$). This implies that at time instant t_0 no job with higher priority than the current job of task τ_i was deferring its pre-emption. If the current job of task τ_i had been released at time t_0 it could defer its pre-emption until $t_0 + \beta_i(0)$ without missing its deadline, as a consequence if the job arrives at a time instant after t_0 it can still defer its pre-emption until the same point in time ($t_0 + \beta_i(0)$).

3.1.3 Sufficient Schedulability Condition for Proposed Framework

The presented framework (properties 1 to 3) cannot schedule all task-sets which would be schedulable with the fully pre-emptive or regular limited pre-emptive schedulers. When trying to leverage these three properties caution must be exercised in order to make sure that the following overload situation does not occur. For some task-sets, some arrival patterns may lead to a task being released in a situation where the admissible pre-emption deferral would in fact be negative. In this situation a deadline may be missed. During the schedulability test a synchronous release situation is considered. This was proven to be the situation leading to the worst-case response time of a task in fixed priority scheduling [BLV07a]. In restricted pre-emption fixed priority scheduling policy the synchronous release of higher priority tasks situation may not lead to the largest response time of a task [BLV07a].

Lemma 1. *Not more than one additional job from every higher priority task may appear (in comparison to the synchronous release case) in the time interval bounded by a release and a deadline of a task.*

Proof. In the synchronous release situation $\left(\left\lfloor \frac{D_i}{T_j} \right\rfloor + 1\right) \times C_j$ units of higher priority workload per higher priority task may be considered. If some lower priority task defers the start of execution of a higher priority task then no more than $\left\lceil \frac{T_i + D_i}{T_j} \right\rceil \times C_j$ units of workload need to be considered. If a middle priority task τ_i had a release more than T_j time units after the release of a job from task τ_j then, considering the constrained deadline task model, the workload of task τ_j would have been concluded at the time of release of task τ_i, hence not interfering in its response time. □

As a consequence of Lemma 1 a sufficient schedulability test ensuring that no such situations can occur is presented in Equation (3.5).

Theorem 2. *A task-set is schedulable when using the presented framework (properties 1 to 3) if, for all tasks, the following inequality is respected:*

$$D_i \geqslant C_i + \sum_{j \in hp(i)} \left\lfloor \frac{D_i}{T_j} \right\rfloor \times C_j + \sum_{j \in hp(i)} \left(\min(C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor \times T_j \right) \tag{3.5}$$

Proof. From Lemma 1 a bound on the maximum number of jobs a higher priority task can release in a given time window is provided. By summing over all higher priority tasks considering the over-provisioning stated in Lemma 1 an upper-bound on the interference any task τ_i suffers in the interval $[0, D_i]$ is obtained. Note that the lower priority blocking is inherently considered by the Lemma 1 since the additional higher priority job considered in the $[0, D_i]$ is a direct consequence of the blocking. Hence the sufficient schedulability test is proven safe. □

The condition stated in inequality (3.5) reflects the fact that at most one additional job of every higher priority task may be present in the time interval bounded by a release and deadline of a middle priority job. Equation (3.5) takes into account the workload of every higher priority task that executes entirely until completion in a time interval of length D_i and then sums two additional workloads or the length of the interval $D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor \times T_j$, whichever is smaller, as additional higher priority workload. For task-sets that miss the sufficient schedulability test a safety mechanism needs to be added to the protocol, which prevents deadlines from being missed. At run-time the sum of the worst-case execution times of all tasks with higher priority than the running task is computed. If at a release this value becomes greater than $\min_{j \in hp(i)}(\beta_j)$ then the timer is set to $t - t_0 + \min_{j \in hp(i)}(\beta_j)$. The previously described set of rules takes into account the existence of job release phasing in relation to the worst-case (synchronous release of higher priority) during the normal run of the schedule and enables better decisions when sporadic behaviour is present (commonly denominated as minimum inter-arrival time). The set of rules may be enhanced by considering that if no higher priority workload is available then the job may be even further delayed. This fact motivates the following theorem.

Theorem 3. *A job of task τ_i may defer its pre-emption for $D_i - \mathrm{WCRT}_i$ time units while no job with priority higher than τ_i is released in the time interval $[r_i^l(t), r_i^l(t) + \Delta_i]$.*

Proof. The amount of pre-emption deferral time is referred to by Δ_i defined as $\Delta_i \overset{\text{def}}{=} D_i - \mathrm{WCRT}_i$. The quantity WCRT_i is defined in the usual way, $\mathrm{WCRT}_i = R^k$, where k is the smallest value that satisfies $R^k = R^{k-1}$. The R_k value is iteratively computed by the following equation [Bur95],

$$R^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R^{k-1}}{T_j} \right\rceil \times C_j \tag{3.6}$$

and choosing $R^0 = C_i + \sum_{j \in hp(i)} C_j$ as the initial value in the iteration for R. If no higher priority job is released in the interval $[r_i^l(t), r_i^l(t) + \Delta_i]$, the job from task τ_i meets its deadline if it pre-empts at $t = r_i^l(t) + \Delta_i$ irrespective of the pre-emption deferral admissible for all the higher priority workload that may be released after or at instant $r_i^l(t) + \Delta_i$, due to the definition of WCRT_i. □

Using the previous theorem in the protocol generates a situation requiring special consideration: Assume one task is deferring, having set the timer on arrival to its corresponding Δ according to the previous theorem. A higher priority task arriving may not use the past value in the timer, but needs now to take into account the $B(0)$ of the highest priority job already deferring.

An example of a problematic situation is displayed in Figure 3.3. If some middle priority job of task τ_i is released at time $r_i^l(t) = 0$, followed by a release of a lower priority job from task τ_l, $\varphi_l^i(r_l^l(t))$ time units after, the situation depicted in Figure 3.3 may arise. The hollow arrows in the picture represent timer values that tasks contended with in the timer setting procedure and set the timer (i.e. were the minimum value at the contending time).

According to previous rules the job from task τ_l sets the timer since its admissible pre-emption deferral is smaller than the value τ_i loaded into the timer, $\max(B_l(0) - r_l^l(t)), 0) < \Delta_i - r_l^l(t)$. At time instant $r_j^l(t)$, when a job from task τ_j was released (where j is of higher priority than i) τ_i has again to check if its admissible deferral time is the minimum among all deferring jobs. When it was released it tried to set the timer until Δ_i, since higher priority workload is available and no knowledge about Equation (3.3) exists at this stage apart from $\beta_i(0)$, the scheduler has to check if this would yield the minimum value for the timer if it was used at time $r_i^l(t)$. When this scenario occurs the following relation is true $\max(B_l(0) - r_j^l(t), 0) > \max(\beta_i(0) - r_j^l(t), 0)$. The timer has to be loaded at this time with

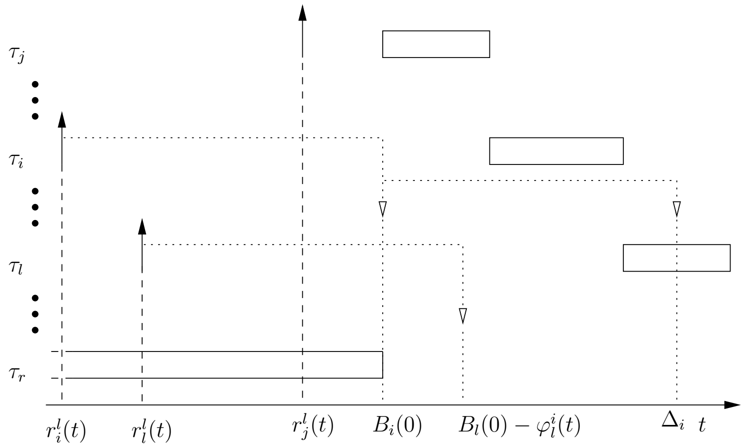


Figure 3.3: Scenario Motivated by Theorem 2

$\max(\min(B_l(0) - \varphi_j^i, \beta_i(0) - \varphi_j^i, \Delta_j), 0)$. The information on the highest priority job deferring its pre-emption has to be stored then, along with the time instant of its release in order for the mechanism to work, since when some other higher priority workload is released its admissible deferral has to mandatorily be re-evaluated.

3.1.4 Admissible Deferral Approximation

It is clear that a trade-off between memory usage and efficiency may be exploited in order to better use the knowledge presented by Equation (3.3). Using only its initial point is too restrictive. Two efficient approaches for having a lower bound on $\beta_i(a)$ (Equation (3.4)) are presented, which may be used at run-time to achieve longer pre-emption deferrals. The following two strategies are employed at the time instant t if task τ_i was the highest priority task blocked at time $t - \varepsilon$ and at this time instant (t) an even higher priority task is released.

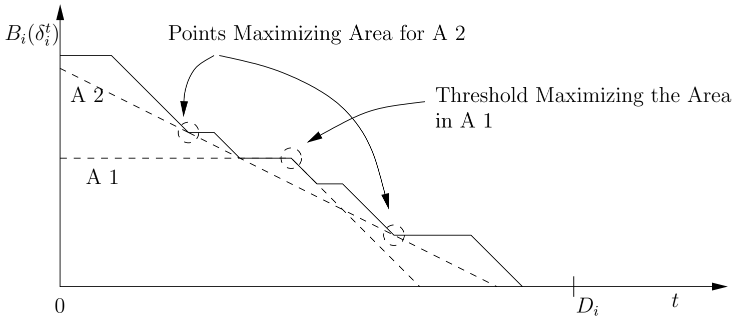


Figure 3.4: Outline of the Devised Approximations for Equation 3.3

A 1 One may consider the usage of another point of $\beta_i(a)$ function to improve deferral decisions. Based on the notion that $\beta_i(a)$ is monotonically decreasing, and that $\frac{d\beta_i(a)}{dt} = 0$ or $\frac{d\beta_i(a)}{dt} = -1$ the following method may be devised. The value set on the timer is $\beta_i(\text{threshold})$, i.e. any job of task τ_i may have its pre-emption deferred for $\beta_i(threshold)$ if no *threshold* time units have elapsed since its release when a higher priority release occurs. After *threshold* time units have elapsed since the release, the value is set to the timer is $\beta_i(threshold) - (threshold - timer)$. This approximation is made possible by the specificities of Equation (3.3) previously referred.

A 2 a linear equation which is a lower bound of $\beta_i(a)$ in the interval $[0, \Delta_i]$ may be created. This linear function is always smaller or equal to $\beta_i(a)$ and tangent to the convex hull defined by $\beta_i(a)$. At the time of release of a higher priority task relative to the previously highest deferring task the deferral is computed by $\frac{y_2-y_1}{x_2-x_1} \times a + (y_1 - \frac{y_2-y_1}{x_2-x_1} \times x_1)$. Both quantities $\frac{y_2-y_1}{x_2-x_1}$ and $y_1 - \frac{y_2-y_1}{x_2-x_1} \times x_1$ are computed off-line.

Both methods define a lower bound on $\beta_i(a)$ function which takes up little memory space and may be exploited quite efficiently. Without any prior knowledge on the arrival pattern of higher priority workload and possible phasings a rule of thumb stating that both areas should be maximized to achieve better performance. For the first approach (A 1), the point that maximizes area $= a \times \beta_i(a) + \frac{a^2}{2}$ is the one chosen. The second one (A 2) chooses the two adjacent points defined by (x_1, y_1) and (x_2, y_2) of the convex hull defined by function $\beta_i(a)$ that maximize area $= \frac{y_1^2}{m} + 2 \times y_1 \times x_1 - m \times x_1^2$, where $m = \frac{y_2-y_1}{x_2-x_1}$. Both approximation are used in a scenario where the previous highest priority job deferring its pre-emption faces a release from a higher priority job.

3.1.5 Implementation Overhead

All of the methods presented so far have small time complexity, having at most three comparisons when setting the timer. At every release a maximum of four values have to be compared in order to chose the minimum. The last approximations rely on a limited number of computations, as was shown in A 1 and A 2 description. These computations are cheap in comparison to the overall savings that they provide.

3.1.6 Tighter Bound on the Number of Pre-emptions

The maximum number of pre-emptions per task may be upper bounded both in the state of the art [YBB09] as well as in our methods by $\left\lfloor \frac{C_i}{Q_i} \right\rfloor$, where $Q_i = \min_{j \in hp(i)}(\beta_j(0))$. This bound is implicitly stated in the work of Yao et al. [YBB09]. For our method it suffices to state that whenever a job from task τ_i executes on the processor it will do so for at least Q_i time units uninterrupted by a higher priority workload. Observe that in all the presented protocols, whenever a single higher priority task τ_j release occurs, the lower priority task τ_i will execute non-pre-emptively for at least $\beta_j(0)$. If subsequent higher priority releases occur, for which the corresponding tasks have smaller $B_k(0)$ (i.e $B_k(0) < \beta_j(0)$) in the worst case scenario then $B_k(0)$ time units would be counted since t_0 (the start of the deferral chain). If $B_k(0) = \min_{j \in hp(i)}(\beta_j(0))$, then $Q_i = B_k(0)$. Hence the same bound applies. Suppose the following taskset presented in table 3.1

	C_i	D_i	T_i	Q_i
τ_1	2	2	5	∞
τ_2	3-ε	15	7	3
τ_3	2	15	15	ε

Table 3.1: Example Task-set Denoting a Pre-emption Corner-case

The value denoted by ε is an arbitrarily small value. This yields the following relation,

$$\lim_{\varepsilon \to 0} \left(\left\lfloor \frac{C_i}{\varepsilon} \right\rfloor \right) = \infty. \tag{3.7}$$

The bound provided, considering the previous bound for jobs of task τ_3 would be overly pessimistic in cases where Q_i is small. Bear in mind that this is an extreme case to motivate the fact that, in specific situations, the number of higher priority releases in a given interval is itself a tighter bound on the number of pre-emptions for a given task. All the higher priority jobs that arrive $\text{WCRT}_i - Q_i$ time units after the job of τ_i started first to execute are guaranteed not to pre-empt and hence may be disregarded in the maximum number of pre-emption computation. This indicates that a suitable bound should then be

$$\min \left(\left\lfloor \frac{C_i}{Q_i} \right\rfloor, \max \left(\sum_{j \in hp(i)} \left\lceil \frac{\text{WCRT}_i - Q_i}{T_j} \right\rceil, 0 \right) \right). \tag{3.8}$$

A less pessimistic pre-emption quantification may be obtained by posing the following trivial optimisation problem, which exploits the notion that some higher priority releases will lead to extended non-pre-emptive regions. The intent is to maximize the number of pre-emption taking as constraints: 1) the maximum number of job releases that each task τ_j produces in a interval of length D_i; 2) the summation of the non-pre-emptive regions considered cannot be greater than C_i, since in the worst-case after each execution resume from τ_i a higher priority release occurs, triggering a subsequent non-pre-emptive region.

$$
\begin{aligned}
\text{maximize} \sum_{j=1}^{i-1} a_j \\
&\text{subject to :} \\
&a_j \leqslant \left\lfloor \frac{D_i}{T_j} \right\rfloor, j \in \{1, \cdots, i-1\} \\
&\sum_{j=1}^{i-1} a_j \times \min_{k \in \{1,\cdots,j-1\}} \{\beta_k\} \leqslant C_i
\end{aligned}
$$

A solution to this problem is easily derived by noting that $\min_{k\in\{1,\cdots,j-1\}}\{\beta_k\}$ increases monotonically as the values of j decrease. This motivates Algorithm 3 which computes an upper-bound on the number of pre-emptions a task τ_i is subject to.

Algorithm 3: Pre-emption Upper-bound

```
preempt ← 0
exec ← 0
$j \leftarrow i-1$
**while** exec $\leqslant C_i$ & $j \geqslant 1$ **do**
    preempt ← preempt + min $\left( \left\lfloor \frac{D_i}{T_j} \right\rfloor, \frac{C_i - \text{exec}}{\min_{k\in\{1,\cdots,j-1\}}\{\beta_k\}} \right)$
    exec ← exec + $\left\lfloor \frac{D_i}{T_j} \right\rfloor \times \min_{k\in\{1,\cdots,j-1\}}\{\beta_k\}$
    $j \leftarrow j-1$
```

The upper-bound on the number of pre-emptions is contained on the variable pre-empt. The algorithm considers the a_j coefficients to take the greatest possible value allowed by its constraint (the maximum number of job releases by τ_j in an interval of length D_i) in a decreasing priority order. This maximizes the objective function since the quantity $\min_{k\in\{1,\cdots,j-1\}}\{\beta_k\}$ decreases monotonically as the priority decreases.

3.2 Evaluation

In this section comparative results on all previously mentioned approaches are demonstrated. The approaches are denominated from first to fourth with the following meaning:

- *first approach* – implements Properties 1 to 3 ;
- *second approach* – implements the method described by the Theorem 3 on top of the *first approach*;
- *third approach* – implement the approximation A1 of Equation (3.3) on top of the *second approach*;

- *fourth approach* – implement the approximation A2 of Equation (3.3) on top of the *second approach*.

3.2.1 Discussion

With greater non-pre-emptive execution length, the likelihood of several higher priority jobs becoming blocked increases. These situations are actually benign and intended, since these will reduce the amount of pre-emptions generally. Take for instance an illustrative scenario where task τ_i is executing and a release from τ_j occurs at time instant t followed by a release of τ_h at $t+\delta$ and $i > j > h$. In this case at time t τ_j becomes blocked. If τ_j pre-empted τ_i before $t+\delta$ then τ_j might be pre-empted in the future by the job released by τ_h at $t+\delta$. If the non-pre-emptive region of τ_i is in this case larger than δ then τ_h will pre-empt only τ_i and never the job from τ_j. In broad terms one can state that larger non-pre-emptive region length promote priority ordered schedules from dispatch and hence reduce the number of required pre-emptions. Parallel to that mechanism, delaying further also enables higher priority workload to wait for a lower priority job to finish its execution, hence reducing the number of pre-emptions as well. Both these facts contend with a contrary effect. By further delaying some middle priority jobs, situations where hypothetical higher priority jobs arrive and cannot be deferred for the same amount of time will generate pre-emptions where none should have existed if all jobs were deferred for the same amount of time as a function of the priority of the running task (Yao et al. approach [YBB09]). Our claim is that the first two effects generally dominate over the third one. This is supported by the experimental data presented in the following subsection. The possibilities of increasing the number of pre-emptions in relation to Yao et al. only stems from the fact that higher priority workload is being moved in the schedule, which does not change any of the off-line guarantees in terms of pre-emptions for each task.

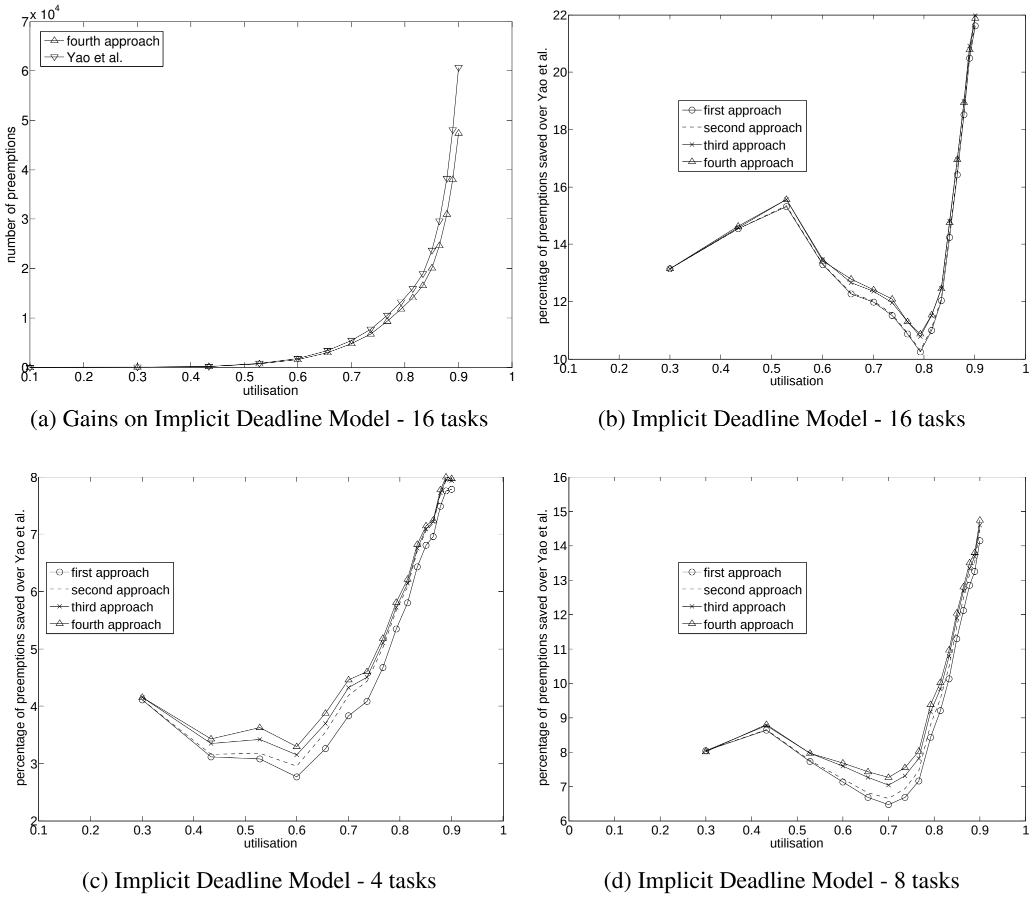
3.2.2 Simulations

The three system models were evaluated using simulations. In each model all tasks are generated using the unbiased task-set generator method presented by Bini and Buttazzo (UUniFast) [BB04a]. Tasks are randomly generated for every utilization step, their maximum execution requirements (C_i) were uniformly distributed in the interval $[50, 500]$.

In the first situation the task-set behaves in a fully periodic manner with implicit deadlines ($D_i = T_i$). In the second situation constrained deadlines are investigated. The constrained deadline model was implemented by randomizing the period of the tasks in relation to their deadlines. For this data run the periods are constructed in the following manner $D_i = T_i - S$, where S is a random variable with uniform distribution in the interval $[0, 0.2 \times T_i]$. In the sporadic model the consecutive release of a task is $T_i + A$ units separated from the last release of the same task. A is taken from a uniform distribution in the interval $[0, 0.5 \times T_i]$.

For every utilization step, the schedule is simulated for 10000 task-sets which respect the sufficient schedulability test from Theorem 2. The observed pre-emptions are quantified and its average across all runs is computed. Utilization steps are non-uniformly distributed and the points are given by the function $\text{step}(k) = 1.1 - \frac{1-k\cdot4}{k\cdot4}$ where $k \in [0, 16]$. This enables us to get a better concentrations of data at higher utilisations.

3.2.2.1 Implicit Deadlines Periodic Model



(a) Gains on Implicit Deadline Model - 16 tasks

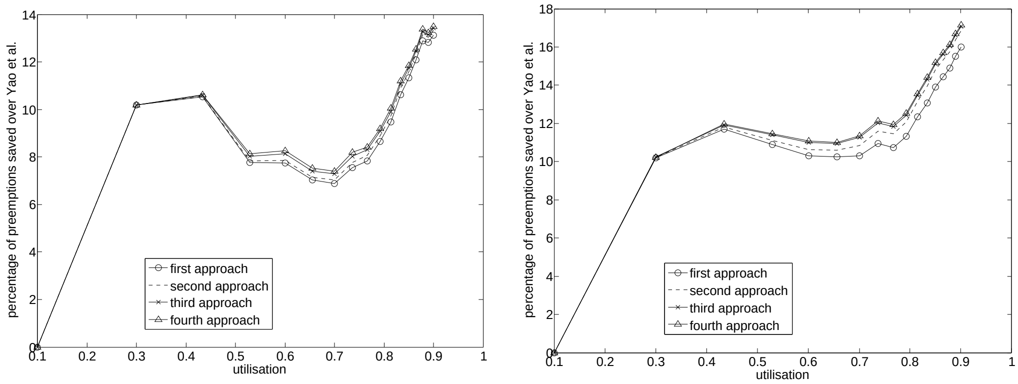
(b) Implicit Deadline Model - 16 tasks

(c) Implicit Deadline Model - 4 tasks

(d) Implicit Deadline Model - 8 tasks

Figure 3.5: Implicit Deadline Task Model Experimental Results

In Figure 3.5a the results in number of average pre-emptions for task-sets with 16 tasks are presented showing the state of the art algorithm and our fourth approach. Both lines display an exponential behaviour, the fourth approach has a brief offset, which implies greater pre-emption savings as is shown in the following plots. It is generally observable



(a) Constrained Deadlines Model - 8 tasks

(b) Sporadic Model - 8 tasks

Figure 3.6: Constrained and Sporadic Task Model Results for n=8

that the approaches proposed in this work outperform the state of the art in the number of avoided pre-emptions in particular at higher utilisations as is show in Figures 3.5b to 3.5d.

As the number of tasks increase it can also be observed in Figures 3.5b to 3.5d that the gains of our approach in comparison to the Yao et al. method become even more evident. This is tied to the fact that with more tasks there will be more situations where the gains, in terms of admissible deferral time, allowed by task phasing appear. It is worth noting that there is a considerable number of pre-emptions that cannot be avoided, the higher priority jobs that are released while some lower priority workload is executing and that have their deadline inside the response time of the lower priority workload will always have to pre-empt it. The displayed data does not make a distinction between unavoidable pre-emptions and the ones that might possibly be avoided by delaying pre-emptions a bit more in a feasible way. For the implicit deadline case with no sporadicity at the biggest utilisation point tested when the task-set is composed of 16 tasks, roughly 21% of the total number of pre-emptions yielded by the state-of-the-art-methods are saved.

3.2.2.2 Constrained Deadlines

By introducing constrained deadlines the off-line assumptions about maximum deferral time are drastically reduced and hence the opportunities for online phasing exploitation are increased, in particular for lower utilisations. In Figure 3.6a the gains of the four approaches are compared against the state of the art method. While only the results for the task set with 8 tasks are shown, the other task-set sizes expose similar tendencies.

3.2.2.3 Sporadic Behaviour

Similar to the constrained deadlines model, a shift towards gains at lower utilisations can be observed in Figure 3.6b. This can be explained with the reduced actual workload due to sporadicity and the increased scope for exploiting online information. Somewhat counter-intuitively these additional gains are not apparent at very high utilisations. Here only the results for 8 tasks are depicted, as the other task-set sizes were also exposing similar trends.

3.3 Floating Non-pre-emptive Schedulability Increase

Similarly to the non-pre-emptive region case, it is intuitive to assume that the floating non-pre-emptive region methodology would itself allow for a schedulability increase with respect to the fully pre-emptive fixed task priority scheduler.

This scheduling policy (floating non-pre-emptive fixed task priority) is inherently non-predictable. This means that situations where higher priority task request a smaller amount of workload or where consecutive jobs are release further apart than the minimum inter-arrival time may constitute situations of higher strain. These worst-case scenarios are complex in their definition with acute formal detail.

In order to motivate this finding a graphical depiction of the previously mention subject is in order. Consider first the trivial task-set composed of two tasks:

	C_i	D_i	T_i
τ_1	2	5	5
τ_2	5	5	5

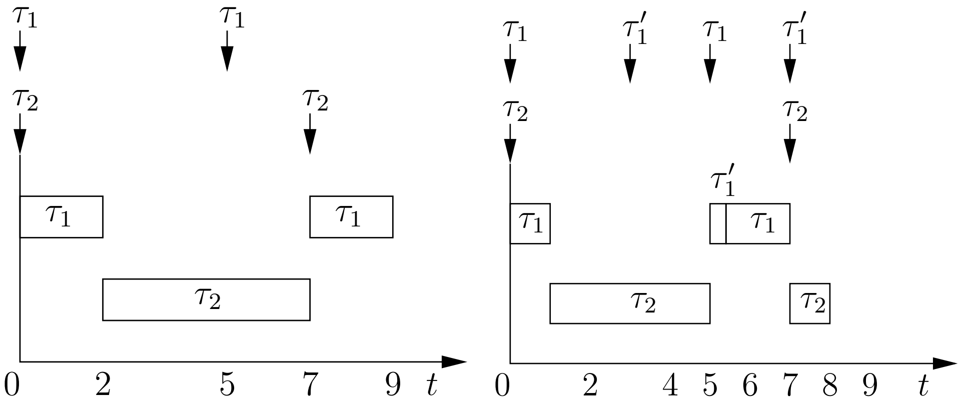
Table 3.2: Floating Non-Pre-emptive Increased Schedulability Task-set

It can be observed from analysing the Table 3.2 that each job from task τ_1 has its execution deferred for a total of 2 time units without missing deadlines. This in turn allows each job of task τ_2 to always complete its execution before its deadline – Refer to Figure 3.7a for a graphical representation of the schedule. It is also apparent from analysing the Task-set 3.2 that under fully pre-emptive FTP scheduling task τ_2 would be subject to an interference from task τ_1 totalling 4 time units in the worst-case in a window of 7 time units. Hence the Task-set 3.2 is not schedulable under fully pre-emptive FTP.

Consider in turn the task-set in Table 3.3 which is a slight modification to the task-set in Table 3.2. In this task-set task τ_1 is divided into two other tasks with the same period such that the sum of worst-case execution times of each new task is equal to the original task in task-set in Table 3.2. Observe that both tasks τ_1 and τ_1' can still have their jobs deferred for at most 2 time units and still meet all deadlines. By constructing a scenario where τ_1' is not released synchronously with task τ_1 (as is shown in Figure 3.7b), task τ_2 will face a deadline miss. Note that the task-sets in tables 3.2 and 3.3 have the exact same total utilization.

	C_i	D_i	T_i
τ_1	1.5	5	5
τ_1'	0.5	5	5
τ_2	5	7	7

Table 3.3: Floating Non-Pre-emptive Unschedulable Task-set



(a) task-set in Table 3.2 Schedule

(b) task-set in Table 3.3 Schedule

Figure 3.7: Task-sets' Floating Non-Pre-emptive Schedules

The examples provided in this section are constructed in order to show that indeed the floating non-pre-emptive FTP has a greater schedulability than fully pre-emptive. It is nevertheless hard to analyse this scheduling policy for task-sets which are not already fully pre-emptive schedulable. Besides lacking a method to characterize the critical situation with respect to higher priority job arrivals, scenarios can be drawn where the execution of some higher priority jobs for less than its WCETS leads to deadline misses of lower priority tasks. The same holds for sporadic arrival of higher priority jobs.

Facing the mentioned difficulties in the analysis of floating non-pre-emptive scheduling a similar scheduling policy, termed ready-q locking, is devised which enables the scheduling of task-sets otherwise unschedulable with FTP while still maintaining a low run-time overhead and allowing the floating non-pre-emptive region scheduling scheme to be employed.

3.4 Ready-Q locking concept

Even though the *floating non-pre-emptive regions* allows considerably more flexibility than *fixed non-pre-emptive regions* it cannot be easily exploited in order to increase the schedulability of fixed task-priority scheduling policy for a sporadic task model.

The possible improvements on the schedulability of task-sets by limiting the interference suffered by lower priority tasks is investigated [MPB12] in this section. This is done by introducing a new task parameter termed ready-queue locking time instant. If a job from a task has pending workload at its ready-queue locking time instant then the ready queue is locked, preventing higher priority workload from being inserted into the ready queue and hence interfering with its execution. As a consequence, the upper bound on the number of pre-emptions each job might suffer is reduced in comparison to the regular fixed task-priority policy. A new pre-emption-threshold scheduling policy is provided so that ready-queue locking can be used together with the aforementioned mechanism. The proposed methods may straightforwardly be used in conjunction with the *floating non-pre-emptive regions* which further helps on the reduction of pre-emptions during workload execution.

The properties of the solutions provided in this work allow for on-line changes in the task-set, since they only require an update of the relative ready-queue locking time instant of every task. The complexity of the proposed solutions is much lower than optimal uniprocessor scheduling policies. Namely, running the ready-queue locking mechanism has a timing complexity of $O(1)$ associated to it. This adds to the complexity of sorting the ready queue in FTP scheduling which is of $O(N)$. These complexity values delineate one of the motivations for the ready queue locking since these are much better than the complexity of sorting the ready queue using the EDF policy – $O(n \times \log(n))$.

In this section each task is characterized by the four-tuple $\langle C_i, D_i, T_i, \mathrm{RQL}_i \rangle$. The first three parameters maintain the same definition as has been used throughout the document. The last task parameter (RQL_i) states the instant in time, relative to a job release, at which the job of task τ_i locks the ready queue if it still has pending workload. While the ready queue is locked job releases are not inserted into the ready queue. Fully pre-emptive and floating non-pre-emptive fixed priority scheduling policies are considered. In the fully pre-emptive case, at every time instant, the job with highest priority in the ready queue is executing in the processor, in the later model the pre-emption from the highest priority task in the ready queue is deferred for the maximum allowed time that still guarantees the task's correct temporal behaviour.

The ready-Q locking mechanism is introduced as a means to limit the amount of interference a task may suffer. It enables a job from a task to request that, after a certain time instant until its current workload completes, no other job is inserted into the ready queue. By preventing higher priority workload releases after a certain point in time, the maximum interference a task may suffer is reduced. Each task τ_i has a ready-Q lock time

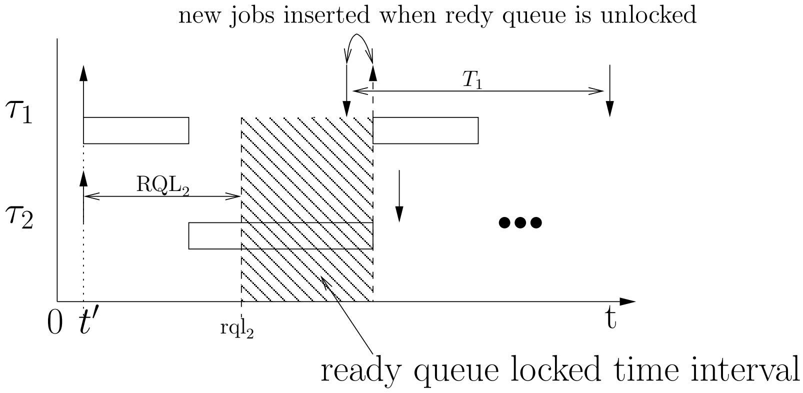

new jobs inserted when redy queue is unlocked
T_1
τ_1
RQL_2
τ_2
0
t'
rql_2
t
ready queue locked time interval

Figure 3.8: Ready-Q Locking Example.

instant defined (RQL_i). The RQL_i time instant is relative to the release of the current job and $\mathrm{RQL}_i \leqslant D_i$. This translates into each job having a rql_i absolute time instant for which $\mathrm{rql}_i \leqslant d_i$, where d_i is the absolute deadline of the job. Since a constrained deadline task model is considered, at any time t there can only be at most one active job from each task in the system.

In Figure 3.8 an example is provided showing the benefits of the ready-queue locking mechanism. Consider the following taskset, composed of two tasks in an implicit deadline task model. The first task has $C_1 = 4$, $T_1 = 10$ and the second task has $C_2 = 7$, $T_2 = 12$. Assume a situation where these two tasks are synchronously released at a given time instant t', as is displayed in Figure 3.8. In fully pre-emptive fixed priority scheduling task τ_2 would suffer an interference of 6 time units from t' to $t' + T_2$, which would then leave only 6 time units for task τ_2 to execute its workload. Since the ready queue was locked by τ_2 at $\mathrm{rql}_2 = t' + \mathrm{RQL}_2 = t' + 6$ it is then only subject to 4 time units of interference. Hence task τ_2 is able to complete its workload before the deadline, consequently releasing the lock on the ready queue.

3.4.1 Ready-Q Lock Implementation Considerations

The scheduler will manage a list of rql_i time instants for all active jobs (i.e. all the jobs in the ready queue at any time instant) from now onwards referred to as rlist. The list has at most $n-1$ valid entries at any time. Each element in the rlist is composed by a time instant and the task to which it belongs.

In Figure 3.9 a depiction of an example rlist evolution over time is shown. At each relevant time instant an arrow points to the current rlist data structure. The solid black triangles represent rql_i time instants relative to each job depicted in the figure.

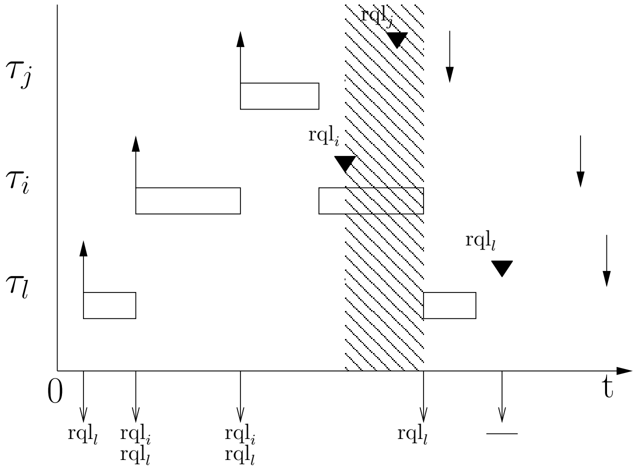


Figure 3.9: rlist List Evolution over Time

In Algorithm 4 a pseudo-code description of the ready-Q locking management mechanism is presented. The mechanism is described as a set of callback procedures for the events of interest. When a job is dispatched for the first time (i.e. no workload has executed yet) the scheduler will get the rql$_i$ from the task control block and evaluate its insertion into the rlist. In this situation, it holds true that the job being dispatched is the highest priority job in the ready queue. It also holds true that there can only be valid entries in the rlist belonging to jobs of lower or equal priority than the one being dispatched. If this would not hold then some higher priority jobs would be in the ready queue which implies that the current job could not be dispatched at this time. As a consequence of this observation, if two tasks τ_i and τ_j have jobs in the ready queue at some time t and $i > j$ then the response time of the job from τ_j will be smaller than the one from τ_i.

At time of first dispatch of a job from task τ_i the scheduler compares rql$_i$ with the value on the top of the list. Two situations may then be observed.

1. rql$_i$ is smaller than the value on top of the list, which leads to the insertion of rql$_i$ in the rlist as the top element

2. rql$_i$ is greater or equal than the value on top of the list, which leads to rql$_i$ being discarded.

In the Situation 1 there exist no job that will lock the ready queue before rql$_i$ and since the job from task τ_i may need to lock the queue in order to complete its workload the value rql$_i$ has to be considered. In the later Situation 2 there exists a job from a lower priority task τ_l which will lock the ready queue before the job from task τ_i requires it. If by the time rql$_i$ the job from τ_i still has pending workload, then the job from τ_l has pending workload as well. Which means that at time rql$_i$ the ready queue is locked by the job from τ_l (rql$_i$ > rql$_l$).

Algorithm 4: Ready Queue Locking Management

```
on event: release job from task $\tau_i$:
rql$_i$ ← release + RQL$_i$
if QueueIsLocked then
    $\tau_l$ ← GetTaskLockingReadyQ()
    append job from $\tau_i$ to $\tau_l$ list of blocked tasks

on event: first dispatch of job from task $\tau_i$:
if rql$_i$ < rlist[0] then
    rlist.push(rql$_i$)

on event: t == rlist[0]:
assign the lock of the ready queue to the task which set rlist[0]
rlist.pop()

on event: terminate execution of $\tau_i$ job:
insert all tasks blocked by $\tau_i$ into the ready queue
```

The job from τ_i will then proceed to complete its workload without requiring to lock the ready queue since a lower priority task conducted that procedure on its behalf. The ready queue remains locked until the job from τ_l finishes its execution.

At any time instant t, if rlist is not empty, there exists a timer which will fire at time $t' = \text{rlist}(0)$, the time instant at the top of rlist. When the timer fires, all the higher priority releases that occur after t' and before rlist(1) (if such value exists) are inserted into the ready queue once the job from the task that sets rlist(0) finishes its workload. In the timer event handler the head of rlist is popped. A task locks the ready queue between the time instant t' (at which a timer set by it fires) and a time instant t'' when either it completes execution or the timer set by some other task fires. Any job released in the interval $[t', t'']$ is inserted into a separate buffer which stores a pointer to all the tasks which are currently blocked. Once task τ_i finishes its workload the scheduler will check whether there were tasks blocked by task τ_i, in case there were, these are then moved from the separate queue into the ready queue with their correct priority.

In an extreme case RQL$_i$ could be set to 0. This would constitute effectively non-preemptive scheduling, where once jobs from τ_i start to execute they would not suffer any further interference. Which could jeopardize the timing properties of tasks with priority greater than τ_i. The value of RQL$_i$ has then to be decided upon considering the higher priority workload timing guarantees; i.e. no higher priority task can miss a deadline due to a lower priority one. These considerations are developed in the following sections.

When the ready queue is locked, the tasks released in the meantime are inserted into a separate circular buffer. The task τ_i that currently holds the ready queue lock has a pointer for the first task to be released while it held the lock, and another pointer to the last task to be released while it held the lock. Once task τ_i terminates the tasks in the circular buffer between the start and the end pointer are inserted into the ready queue. Notice that the complexity of operating this mechanism is reduced in comparison to EDF.

It is important to stress that all operations on the rlist have $O(1)$ complexity. A ready-queue lock is only enforced at the release of some higher priority task, and a new element can only be added to the rlist at the time of the first dispatch of a job. In the same manner an element in rlist is only removed when the job responsible for its insertion terminates.

3.4.2 Maximum Interference Computation

The computation of the maximum interference a job might suffer in a ready queue locking framework is presented in this subsection. This computation is carried out on the maximum busy period of the level-i priority for task τ_i. The worst-case level-i busy period value represents the biggest amount of workload from tasks with priority bigger or equal than i may execute continuously assuming that all the level-i priority tasks are blocked by the maximum allowed time (B). Let us define Γ_i as the set of tasks with priority greater or equal to i. The largest of the level-i busy period occurs when all tasks in Γ_i are release synchronously and are blocked by the longest time possible(B) [KBL10].

The worst-case level-h busy period is defined as:

$$L_i(B) \overset{\text{def}}{=} \min\{t | t - (B + \text{rbf}(\Gamma_i, t)) = 0\} \tag{3.9}$$

Let us assume that each task locks the ready queue at time RQL_i. A bound on the higher priority interference is required. This bound consists on all the higher priority releases, prior to the release of a job from task τ_i, which have not yet completed at the release time instant and all of the higher priority releases that occur since the release of τ_i and RQL_i.

Theorem 4. *The maximum interference any job from task τ_i is subject to is generated in the level-i busy period where the first job of task τ_i in the busy period is released ϕ time units after a synchronous release of all higher priority tasks, where $0 \leqslant \phi \leqslant L_{i-1}(Q_i)$.*

Proof. Assume that a synchronous release of all higher priority tasks occurs at time $t = 0$ and that a release from τ_i occurs at $t = 0$ as well. In this scenario the interference the current job from task τ_i can suffer is I_0. Let us assume now that $\phi \neq 0$. For this case the interference the job from task τ_i is $I_0 - \phi$, provided that the number of higher priority releases in the interval $[0, \text{RQL}_i]$ is the same as for the interval $[0, \text{RQL}_i + \phi]$. In a situation where the number of higher priority releases in the interval $[0, \text{RQL}_i + \phi]$ is greater than the count from $[0, \text{RQL}_i]$, then the interference suffered by the job of task τ_i is $I_0 - \phi + W$, where W is the additional workload released in the interval $[\text{RQL}_i, \text{RQL}_i + \phi]$ which will interfere with τ_i until D_i. The ϕ for which the interference is greater constitutes the worst-case scenario for a job from task τ_i.

After $L_{i-1}(Q_i)$ time units have elapsed since the previous synchronous release of all higher priority tasks has elapsed, mandatorily an idle time has occurred in the processor. If for some $\phi > L_{i-1}$ a situation of bigger amount of higher priority workload is generated then the critical scenario would not start with a synchronous release of all higher priority workload, which would contradict the first part of the proof. Since higher priority tasks cannot endure a larger blocking time than Q_i then the same value is at most the maximum admissible blocking time for tasks of priority higher or equal to i. Hence the Theorem is proven. □

In order to find the maximum amount of interference one has then to find the correct higher priority synchronous release offset ϕ. The value ϕ is extracted by looking at the frame considering all the possible ϕ values in the interval $[0, L_{i-1}(Q_i)]$

Each frame of the task τ_i, in the busy period which starts with a synchronous release of the tasks in the level $i-1$ priority level, has to be checked for its temporal behaviour. The deadline is met in q^{th} frame if the slack in the frame is greater or equal to 0. Let us assume Γ_i to be the subset of tasks with higher priority that task τ_i.

Let us define the following variable in order to ease the discussion:

$$r_i^q(\phi) = (q-1) \times T_i + \phi \tag{3.10}$$

The variable $r_i^q(\phi)$ encodes the value of the release of the q^{th} job from task τ_i relative to an absolute time instant, with a given ϕ offset.

Definition 6 [job frame]: *The time interval between a job release and its deadline is termed job frame.*

All jobs in the level-i busy period have to be checked for deadline violations. In order to compute the number of jobs in the busy period, a maximum possible blocking for the level-i has to be considered. The level-i schedule can never be blocked by more than Q_i time units, since the tasks of priority greater than τ_i could otherwise suffer deadline misses. On the other hand τ_i can never be blocked by more than $D_i - C_i$, hence an upper bound on the number of jobs from τ_i in the busy period can be obtained considering $\min(Q_i, D_i - C_i)$ as the workload initially blocking the level-i schedule.

$$K = \left\lceil \frac{L_i(\min(Q_i, D_i - C_i))}{T_i} \right\rceil \tag{3.11}$$

The slack in each frame q may be expressed by:

$$\beta_i^q = \min_{\phi}\{\max(\max_{t \in \left[r_i^{q-1}(\phi), r_i^{q-1}(\phi)+\mathrm{RQL}_i\right]}\{t - (\mathrm{rbf}(\Gamma_i, t) + q \times C_i)\},$$
$$(r_i^{q-1}(\phi) + D_i) - (\mathrm{rbf*}(\Gamma_i, r_i^{q-1}(\phi) + \mathrm{RQL}_i) + q \times C_i))\} \tag{3.12}$$

Where

$$\mathrm{rbf}(\Gamma_i, t) \overset{\mathrm{def}}{=} \sum_{j \in \Gamma_i} \left\lceil \frac{t}{T_j} \right\rceil \times C_j \tag{3.13}$$

and

$$\mathrm{rbf*}(\Gamma_i, t) \overset{\mathrm{def}}{=} \sum_{j \in \Gamma_i} \left(\left\lfloor \frac{t}{T_j} \right\rfloor + 1 \right) \times C_j. \tag{3.14}$$

Notice that equations 3.13 and 3.14 only differ in value in multiples of T_j. The usage of both request bound function forms is tied to reducing equation display complexity. The Equation (3.12) is composed by two terms. In the first one the maximum idle time in the schedule is searched for in the interval between the release of a job from task τ_i and the ready queue locking time instant of the said job. In the second term the maximum idle time for the given frame is searched in the interval between the ready queue locking time instant of the job and its deadline, but since higher priority jobs released in this interval do not interfere with task τ_i it suffices to compute the idle time at the deadline of the job. For a given ϕ the maximum idle time is then the maximum value given by the two mentioned terms. Then for all the possible ϕ the minimum of the idle times is stored in β_i^q.

Considering all the frames the maximum amount of blocking that a job from task τ_i can endure is then defined as:

$$\beta_i = \min_q\{\beta_i^q\} \tag{3.15}$$

A simple depiction of the schedulability condition is provided in Figure 3.10. In this example the taskset is composed of three tasks. All tasks are implicit deadline tasks with parameters (T_i, C_i). Task τ_1 has $(5,1)$, and τ_2, τ_3 have $(7,2)$ and $(16,4)$ respectively. It is implicitly assumed that $\mathrm{RQL}_i = D_i$ to simplify the visualization. The subject of schedulability analysis is in this case τ_3. Two frames from τ_3 are present in the figure. One can observe that $\beta_3^1 = 4$ and $\beta_3^2 = 8$. Since both frames present a non-negative slack value, then the τ_3 is deemed schedulable. Note that the maximum blocking time admissible for τ_3 is then 4 time units. This then implies that the level-3 busy period terminates at time instant 24, hence no more frames from task τ_3 require to be checked.

The maximum admissible blocking time for task τ_i is denoted by β_i. The task τ_i is schedulable if $\beta_i > 0$, when task τ_i is the task with lowest priority in the system. In order

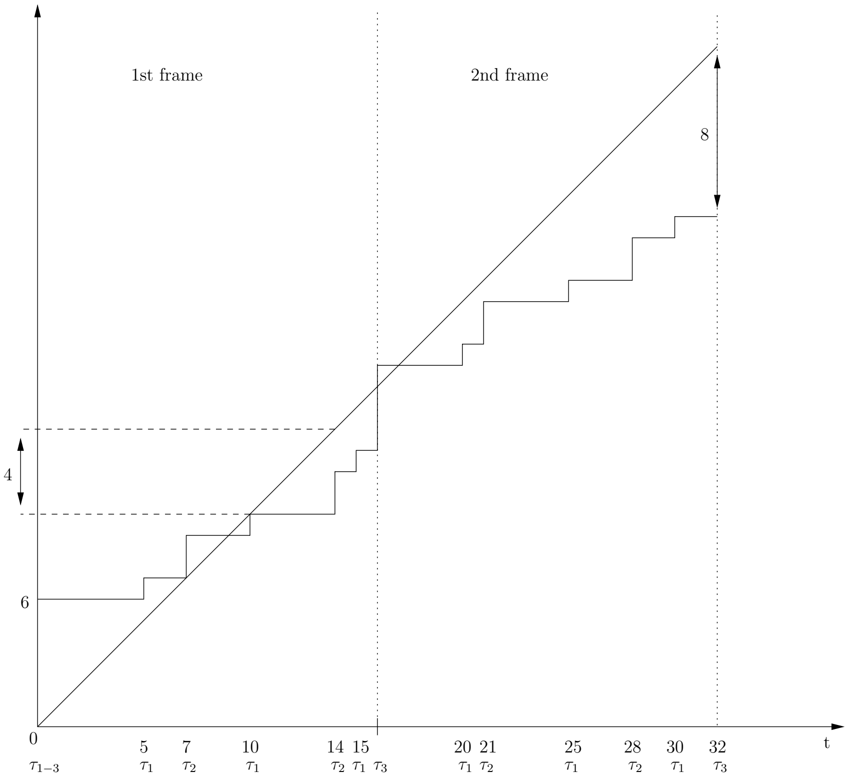


Figure 3.10: Schedulability Condition

for the task-set to be schedulable the RQL$_i$ values for all the tasks have to be set so that the higher priority tasks temporal guarantees are still met.

For every q^{th} frame only a subset of the $\phi \in [0, L_{i-1}(Q_i)]$ needs to be checked. The higher priority workload increment for each frame occurs only at the boundary of the minimum inter arrival time of each higher priority task. Since only offsets up to the maximum busy-level period of the $i-1$ priority level needs to be checked as has been shown in Theorem 4, only the set of possible higher priority releases in the interval $[0, L_{i-1}(Q_i)]$ need to be checked. The discrete set of offsets considered are the ones smaller than $L_{i-1}(Q_i)$ (as per Theorem 4) and such that the ready-queue locking rime instant coincides with the arrival of a new higher priority job. This set is defined as:

$$\Phi_i^q \stackrel{\text{def}}{=} \{0\} \cup \bigcup_{j\in hp(i)} \left(\left\{ \left\lceil \frac{r_i^{q-1}(0)+\text{RQL}_i}{T_j} \right\rceil, \quad \ldots, \left\lfloor \frac{L_{i-1}(\beta_i)+r_i^{q-1}(0)+\text{RQL}_i}{T_j} \right\rfloor \right\} \times T_j - (r_i^{q-1}(0)+\text{RQL}_i) \right) \tag{3.16}$$

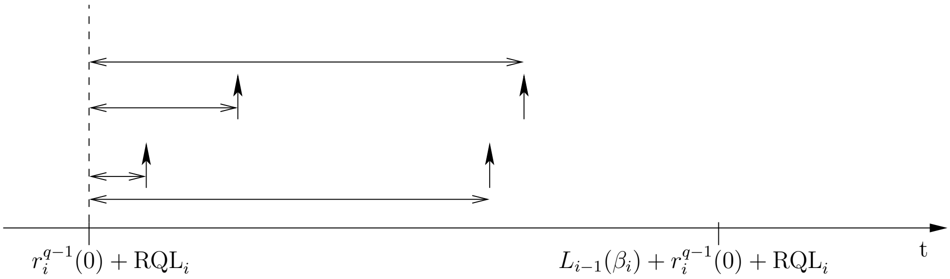

$r_i^{q-1}(0) + \mathrm{RQL}_i$
$L_{i-1}(\beta_i) + r_i^{q-1}(0) + \mathrm{RQL}_i$
t

Figure 3.11: Set of Relevant Offsets for Frame q

In Figure 3.11 the set of relevant offsets for the given frame is graphically represents. The offsets are intuitively shown as the distance between the higher priority releases in the $\left[r_i^{q-1}(0) + \mathrm{RQL}_i, L_{i-1}(\beta_i) + r_i^{q-1}(0) + \mathrm{RQL}_i\right]$ interval. Of course $\phi = 0$ still has to be tested since the sysnchronous release of all the tasks in the $level - i$ priority band might still generate the critical interference scenario for task τ_i.

3.4.3 Ready-queue Locking Time Instant

The RQL_i value has to be such that τ_i has enough time to carry out its workload and the higher priority task are not blocked by more time than it is admissible.

Each task in the system may endure a maximum blocking time without endangering its timing guarantees [YBB11a]; i.e. all tasks may be blocked by a lower priority task up to a certain limit without missing any deadline. The task with highest priority in the system can always sustain a maximum blocking time of $\beta_1 = D_1 - C_1$.

Let us assume that $\mathrm{RQL}_2 = D_2 - Q_2$. A hypothetical release from task τ_1 arriving at time instant RQL_2 would see its ready queue insertion delayed by task τ_2 by at most Q_2 time units. Since, by definition, $Q_2 \leqslant \beta_1$ this job from task τ_1 would still meet the deadline. The concept may be generalized to n tasks by using the following relations:

$$Q_i \overset{\text{def}}{=} \begin{cases} 0 & \text{,if } i = 1 \\ \min_{j \in hp(i)}(\beta_j) & \text{,if } 1 < i \leqslant n \end{cases} \tag{3.17}$$

Equation (3.17) defines the maximum blocking time (Q_i) that all task of higher priority than i may endure without missing a deadline.

Assuming $\mathrm{RQL}_i = D_i - Q_i$ ensures then that, after locking the ready queue a job from task τ_i can only maintain it blocked by at most Q_i time units, hence not jeopardising higher priority task's temporal behaviour. Even if a synchronous release of all tasks in the $i - 1$

priority level occurs after a ready-queue lock, at time instant rql_i then the queue will not be locked by more than Q_i.

In a scenario where $C_i < Q_i$ then $\mathrm{RQL}_i = D_i - C_i$. Since values are only inserted into the rlist at time of the first dispatch of a job, it might be the case that the first dispatch of the job occurs after rql_i, and hence the job might suffer more interference than it is admissible. By setting $\mathrm{RQL}_i = D_i - C_i$ it is ensured that the first dispatch of the job will always occur before rql_i. For each task in the system the RQL_i is then set in the following manner:

$$\mathrm{RQL}_i = D_i - \min(Q_i, C_i) \tag{3.18}$$

Theorem 5. *The ready-queue locking scheduling policy dominates over fully pre-emptive fixed task priority.*

Proof. The maximum interference (UI_i) a task τ_i is subject to is smaller or equal to the interference the same task may endure without the ready-queue locking mechanism. This fact is trivially proved by noting that for the same arrival pattern a job can be interfered by jobs released in a smaller time interval. Hence all the task-sets scheduled by fully pre-emptive fixed priority are schedulable with ready-queue locking. Since the UI_i may at times be smaller than the maximum interference, in fully pre-emptive scheduling, there exist task-sets schedulable by ready-queue locking which fail to be in fully pre-emptive scheduling. □

It is worth noting that setting $\mathrm{RQL}_i = D_i - \min(Q_i, C_i)$ is not necessarily the optimal RQL_i time instant assignment. This is driven by the fact that an earlier locking will be possible in many situations, caused by a worst-case response time of a task which is shorter than RQL_i. Even more, the higher priority task τ_j which limits Q_i, may have a worst-case phasing such that an earlier RQL_i will not lead to a reduction of direct interference and thus have no negative effect on the higher priority task and a later than worst-case release would mean more progress for the locking task. However, deriving the optimal RQL_i is non-trivial and beyond the scope of this work.

3.4.4 Ready-q Locking with Pre-emption Threshold

Another adaptation of FTP targeted at reducing the number of pre-emptions is termed pre-emption threshold. In this framework tasks are characterized by a base priority and a pre-emption threshold. When tasks are in the ready queue and not executing they conserve their base priority. When a task commences execution on the processor its priority is increased to its pre-emption threshold value. Effectively, this task can only be pre-empted by tasks with base priority exceeding its pre-emption threshold value. The initial pre-emption-threshold assignment algorithm has been presented in [WS99]. The main drawback of the pre-emption-threshold assignment method is that it does not easily allow for a maximum blocking time per task computation. With this in mind the algorithm is rethought. As opposed to the solution proposed in [WS99] the taskset is parsed from the highest priority task to the lowest priority one.

Let us assume that the pre-emption threshold (π_i) of a task τ_i is:

$$\pi_i = \min(j | C_i \leqslant \min_{k \in \{j, \cdots, i-1\}} \beta_k).$$

Upon release a job from task τ_i is inserted into the ready queue (in case the ready queue is not locked) with priority i. At the time of its first dispatch onto the processor its priority is elevated to π_i. The set Γ_i^h denotes the set of tasks of higher priority than π_i, whereas the set Γ_i^l denotes the set of tasks with priority higher than i but lower than or equal to π_i. In order to ensure schedulability of the system, for each task, an initial upper-bound on the maximum blocking tolerance is provided. Initially $\beta_i = \min(D_i - C_i, Q_i)$, which is the maximum value that the blocking time could potentially take. The $level - i$ busy period is then parsed and checked for deadline misses.

Let us first define t_s^q as the worst-case time instant for the first dispatch of the q^{th} job from task τ_i. This time would be similar to computing the length of the $level - i$ busy period without considering the q^{th} job from task τ_i. Where t_s^q is the worst-case instant in time when the q^{th} job of task τ_i starts to execute considering a blocking of β_i time units in the beginning of the busy period.

$$t_s^q = \min\{t | t - (\beta_i + rbf(\Gamma_i, t) + (q-1) \times C_i) \geqslant 0\} \tag{3.19}$$

Notice that if $t_s^q > r_i^q(0) + \text{RQL}_i$ then the q^{th} job from task τ_i would miss a deadline since $D_i - \text{RQL}_i \leqslant C_i$.

A deadline is missed in the q^{th} frame when there does not exist any idle time instant for the $level - i$ busy period in the interval $[t_s^q, r_i^q(\phi) + D_i]$. This condition may be written in the following form:

$$\nexists t \in [t_s^q, r_i^q(\phi) + \text{RQL}_i] | t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i) \geqslant 0 \tag{3.20}$$

$$\wedge$$

$$D_i - (\beta_i + \text{rbf*}(\Gamma_i^h, t_s^q) + \text{rbf*}(\Gamma_i^l, r_i^q(\phi) + \text{RQL}_i) + q \times C_i) \leqslant 0 \tag{3.21}$$

The condition expressed in Equation (3.20) relates to the idle time occurrence in the interval between the release time of the q^{th} job of task τ_i and the instant in time at which it locks the ready queue. The second condition in Equation (3.21) relates to the search of idle time after the ready-queue is locked by task τ_i, in an interval where higher priority workload with higher priority than τ_i's pre-emption threshold do not interfere with the execution of τ_i.

If both conditions are met simultaneously, then for the given β_i a deadline may be missed in the q^{th} frame of task τ_i In this case, the β_i parameter has to be decreased.

$$\begin{aligned}\beta_i^- \overset{\text{def}}{=} \min\Bigg(& \min_{t\in A}\{t-(\beta_i + rbf(\Gamma_i,t) + (q-1)\times C_i)\},\\ & \min_{t\in[t_s^q, r_i^q(\phi)+\mathrm{RQL}_i]}\{t-(\beta_i + rbf(\Gamma_i^h,t_s^q) + rbf(\Gamma_i^l,t) + q\times C_i)\},\\ & D_i - (\beta_i + \mathrm{rbf*}(\Gamma_i^h,t_s^q) + \mathrm{rbf*}(\Gamma_i^l, r_i^q(\phi)+\mathrm{RQL}_i) + q\times C_i)\Bigg)\end{aligned} \tag{3.22}$$

where A is the set of time instants when higher priority releases occur in the time interval $[r_i^q(\phi), t_s^q]$:

$$A \overset{\text{def}}{=} \{0\} \cup \bigcup_{j\in hp(i)} \left(\left\{\left\lceil \frac{r_i^{q-1}(\phi)}{T_j}\right\rceil, \cdots, \left\lfloor \frac{t_s^q}{T_j}\right\rfloor\right\} \times T_j\right) \tag{3.23}$$

In Equation (3.22) there are three terms present. The first one relates to the t_s^q value, and both the second and the third relate to the idle time available in the q^{th} frame of task τ_i for the given ϕ parameter. In the second and third terms the minimum amount of β_i reduction required to compute all the workload in the frame is computed. In the first parameter the minimum β_i decrease which ensures that the t_s^q time instant occurs before one higher priority job is computed. The β decreased computed in the first term aims at decreasing the t_s^q such that the q^{th} job from τ_i starts earlier potentially suffering smaller interference from tasks of higher priority which is still lower than the τ_i pre-emption threshold. The minimum value between the three parameters is then chosen to be the value of β_i^- for the current algorithm iteration. Finally the current maximum blocking time allowed by task τ_i is obtained by :

$$\beta_i^k = \beta_i^{k-1} - \beta_i^{-\ \ (k-1)} \tag{3.24}$$

The iterative nature of the procedure described in Equation (3.24) is patent on the super scripts usage. These are omitted from the remainder discussion so as to not overload notations unnecessarily.

If the deadline is met for the q^{th} frame with $\phi = 0$, using ready-q locking still requires all the possible offsets to be tested. As is the case for the ready-q locking mechanism only a subset of all possible ϕ values has to be tested, this subset Φ_i^q is defined in Equations 3.23.

When the following condition is met

$$\begin{aligned} &\forall q \in \{0, \cdots, K\}, \forall \phi \in [0, L_{i-1}(\beta)], \exists t \in [r_i^q(0), r_i^q(0) + D_i] : \\ &t - \beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i \geqslant 0 \end{aligned} \tag{3.25}$$

then no deadlines are missed for task τ_i. Ensuring that the condition is met for all the tasks in the taskset will ensure the schedulability of the taskset.

Algorithm 5: Pre-emption Threshold Assignment with Ready-q Locking

```
Input   : $T$
$\beta_1 = D_1 - C_1$
$\pi_1 = 1$
for $i = \{2, \cdots, n\}$ do
    $\pi_i = \min(j | C_i \leqslant \min_{k \in \{j, \cdots, i-1\}} \beta_k)$
    $Q_i = \min_{j \in hp(i)} \{\beta_j\}$
    $\mathrm{RQL}_i = D_i - \min(Q_i, C_i)$
    $\beta_i = \min(D_i - C_i, Q_i)$
    $K = \left\lceil \frac{L_i(\beta_i)}{T_j} \right\rceil$
    for $q \in \{1, \cdots, K\}$ do
        for $\phi \in \Phi_i^q$ do
            $t_s^q = \min\{t | t - (\beta_i + rbf(\Gamma_i^h, t) + rbf(\Gamma_i^l, t) + (q-1) \times C_i) \geqslant 0\}$
            while
            $\nexists t \in [t_s^q, r_i^q(\phi) + \mathrm{RQL}_i] | t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i) \geqslant$
            $0 \wedge D_i - (\beta_i + \mathrm{rbf*}(\Gamma_i^h, t_s^q) + \mathrm{rbf*}(\Gamma_i^l, r_i^q(\phi) + \mathrm{RQL}_i) + q \times C_i) \leqslant 0$ do
                $\beta_i^- = \min(\min_{t \in A}\{t - (\beta_i + rbf(\Gamma_i, t) + (q-1) \times$
                $C_i)\}, \min_{t \in [t_s^q, r_i^q(\phi) + \mathrm{RQL}_i]}\{t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times$
                $C_i)\}, D_i - (\beta_i + \mathrm{rbf*}(\Gamma_i^h, t_s^q) + \mathrm{rbf*}(\Gamma_i^l, r_i^q(\phi) + \mathrm{RQL}_i) + q \times C_i))$
                $\beta_i = \beta_i - \beta_i^-$
                if $\beta_i < 0$ then
                    return UNSCHED
                $t_s^q = \min\{t | t - (\beta_i + rbf(\Gamma_i^h, t) + rbf(\Gamma_i^l, t) + (q-1) \times C_i) \geqslant 0\}$
return SCHED
```

The insertion of the rql_i value into the rlist data structure is considered at the first dispatch of a job, which coincides with the time instant of priority promotion of the job to π_i, hence the priority ordering between jobs with valid entries in the rlist is never changed after the insertion.

Notice that in a situation where $\text{RQL}_i = D_i$ the provided schedulability test and pre-emption-threshold assignment technique is still valid for the regular pre-emption-threshold mechanism [Lam] . Contrary to the method presented in [WS99] which assigns π_i the highest value that ensures schedulability (if such exists), Algorithm 5 assigns π_i the smallest possible value.

Similarly to the Ready-q locking assigning, $\text{RQL}_i = D_i - \min(Q_i, C_i)$ is not optimal. Given the RQL_i value for each task, the schedulability test provided in Algorithm 5 is necessary and sufficient.

3.4.5 Pre-emption Upper Bounds

In this section a brief comparison between the pre-emption upper bound guarantees of the proposed solutions is provided. The value $WCRT_i$ denotes the worst-case response time of task τ_i. For fully pre-emptive

$$\sum_{j \in hp(i)} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \tag{3.26}$$

The simple ready-queue locking scheduling policy ensures that pre-emptions may only occur in the time interval between release and the locking of the ready-queue. Hence with ready queue locking the worst-case number of pre-emptions will never be greater than in the fixed task priority scheduling.

$$\sum_{j \in hp(i)} \left\lceil \frac{\min(WCRT_i, \text{RQL}_i)}{T_j} \right\rceil \tag{3.27}$$

The regular pre-emption threshold mechanism ensures that only tasks with higher priority than the pre-emption threshold priority may in fact pre-empt.

$$\sum_{j \in hp(\pi_i)} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \tag{3.28}$$

For the pre-emption threshold with ready queue locking scheduling policy, the pre-emption upper-bound for each task is guaranteed to be smaller or equal than the value for any other policy described in this work. In the worst-case scenario a job from task τ_i would start execution immediately after release. In this situation assuming it executes for the worst-case execution time, suffering the worst possible interference it can be at most pre-empted during $\min(WCRT_i, \mathrm{RQL}_i)$ time units by the tasks with priority greater than π_i.

$$\sum_{j \in hp(\pi_i)} \left\lceil \frac{\min(WCRT_i, \mathrm{RQL}_i)}{T_j} \right\rceil \tag{3.29}$$

Obviously with floating non-pre-emptive regions the maximum number of pre-emptions each job would suffer is upper bounded by:

$$\left\lfloor \frac{C_i}{Q_i} \right\rfloor \tag{3.30}$$

As as been shown in [MP11], for small enough values of Q_i this bound is worse than the ones dictated by the higher priority releases. Hence for all the scheduling policies provided the upper bound on the number of pre-emptions is given by the

$$\min(\text{fully pre-emptive bound}, \left\lfloor \frac{C_i}{Q_i} \right\rfloor)$$

.

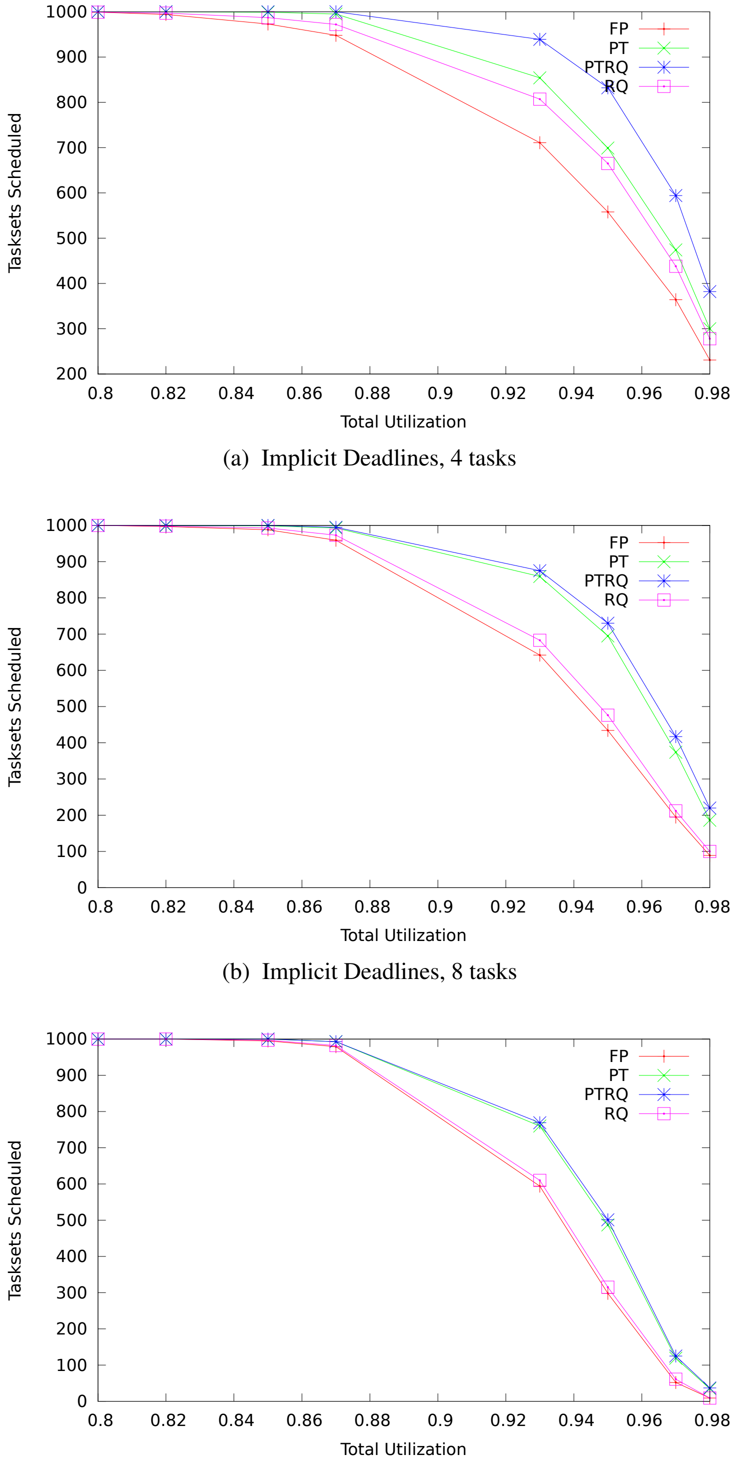
3.4.6 RQL Evaluation

In this section the proposed solutions are evaluated in terms of schedulability performance against fully pre-emptive fixed task-priority and regular pre-emption threshold.

Both proposed scheduling policies were evaluated with respect to schedulability. In each model all tasks are generated using the unbiased task-set generator method presented by Bini (UUniFast) [BB04a]. Tasks are randomly generated for every utilization step in the set $\{0.8, 0.82, 0.85, 0.87, 0.93, 0.95, 0.97, 0.98\}$, their maximum execution requirements (C_i) were uniformly distributed in the interval $[20, 400]$. For every utilization step 1000 tasksets are trialled and checked whether the respective algorithm considers it schedulable. Task set sizes of 4, 8, and 16 tasks have been explored.

In the second situation constrained deadlines are investigated. The constrained deadline model was implemented by randomizing the period of the tasks in relation to their deadlines. For this data run the relative deadlines are constructed in the following manner $D_i = T_i - S$, where S is a random variable with uniform distribution in the interval $[0, 0.2 \times T_i]$. The results of these are put into juxtaposition with the implicit deadlines results in Figures 3.13a to 3.13c.

The data relative to regular fixed priority is tagged with FP. Pre-emption threshold is shown with tag PT. The simpler method of ready queue locking is addressed by RQ and the simple ready queue locking used together with pre-emption threshold is tagged with PTRQ.

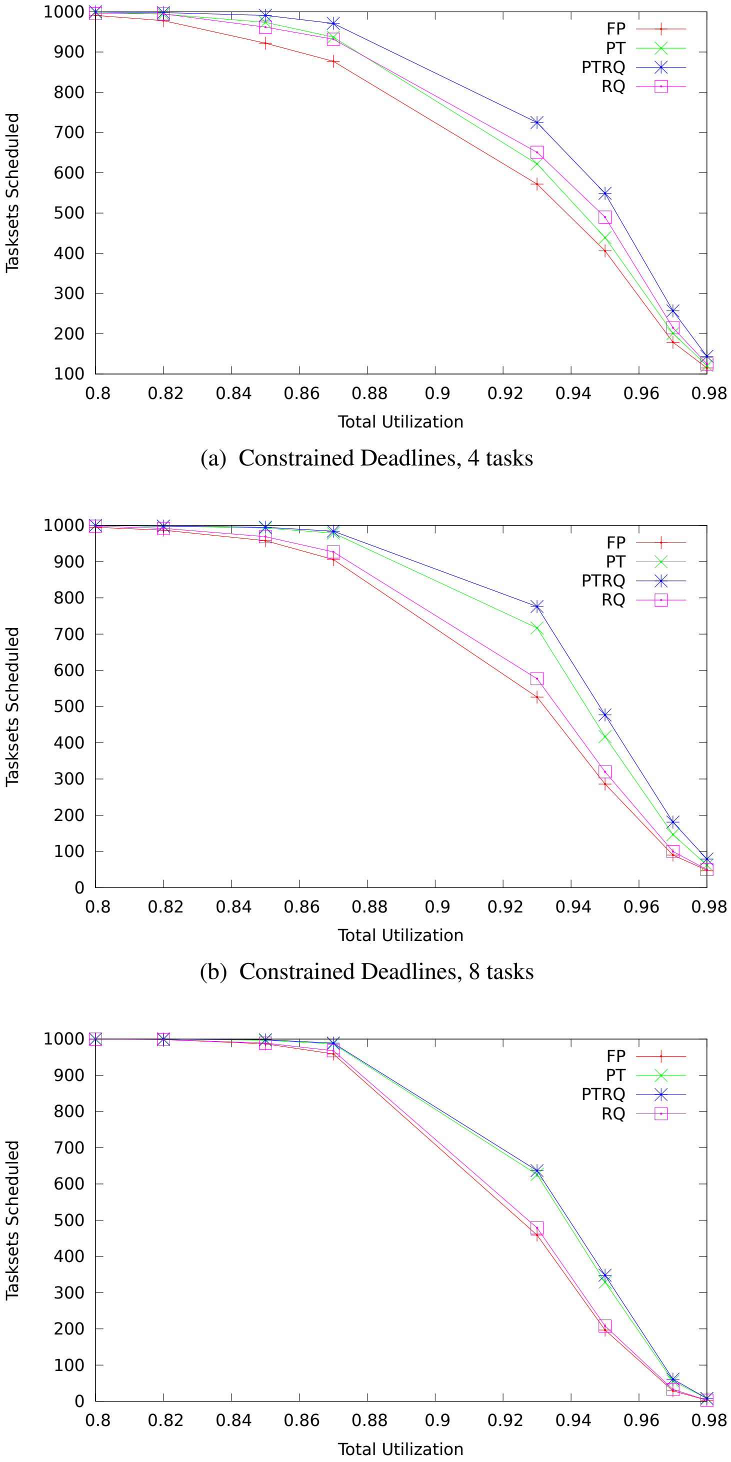


(a) Implicit Deadlines, 4 tasks

(b) Implicit Deadlines, 8 tasks

(c) Implicit Deadlines, 16 tasks

Figure 3.12: Simulation Results for the Implicit Task Model



(a) Constrained Deadlines, 4 tasks

(b) Constrained Deadlines, 8 tasks

(c) Constrained Deadlines, 16 tasks

Figure 3.13: Simulation Results for the Constrained Task Model

3.4.7 Discussion

The pattern present in the results is clear. The simple ready-queue locking mechanism outperforms regular fixed priority as expected. However, it only rarely outperforms pre-emption threshold and that comparison deteriorates with increasing task set sizes and only gives it an overall gain for 4 tasks in the constrained deadline case. It is however noteworthy that there is no clear dominance relationship between PT and ready-queue locking, as some tasks sets are deemed schedulable with one, but not the other. This lack of dominance holds both for implicit and constrained deadlines models as well as for the different task set sizes investigated. The PTRQ solution performs always better than the simple pre-emption-threshold mechanism or simple reaqdy queue locking. Though again the benefits of PTRQ dilute with the increase of the taskset size.

Chapter 4

Pre-emption Delay Upper-bound for Limited Pre-emptive Scheduling

As was previously addressed in the initial chapters of this thesis (1 and 2), the pre-emptive nature of the scheduler gives rise to a phenomenon termed pre-emption delay. This is a by-product of the methodology for the WCET quantification. The worst-case execution requirement is assessed assuming execution in isolation (please refer to Chapter 2 for further detail). Once a pre-emption occurs, the state of the several architectural systems accessed by the task may be altered. These state changes were not accounted for in the temporal analysis and have to be upper-bounded in order to guarantees the temporal properties of each task. In this Chapter, an upper-bound on the pre-emption delay a given task suffers, when floating non-pre-emptive region scheduling is employed, is presented. The related theory has been published and presented previously in [MNPP12a, MNPP12b].

This chapter addresses the computation of the pre-emption delay in systems using pre-emption triggered floating non-pre-emptive regions which was previously not covered in the literature. In order to perform this computation, the variability of the pre-emption delay throughout the task execution is modelled by aid of a function $f_i(t)$, and an upper-bound $G_i(t)$ on the pre-emption delay in the task execution referential is also considered.

The first subsection focuses on determining the pre-emption delay function $f_i(t)$ of a task τ_i, that defines the maximum pre-emption delay should τ_i be pre-empted t time units after starting. The determination of $f_i(t)$ directly comes from [MNPP12a]. The computation of $f_i(t)$ requires the CRPD when pre-empted at every basic block (BB_b) to be known (see section 4.1). Then the set of basic blocks that may be executed at time t has to be computed (see section 4.2). Function $f_i(t)$ can then be computed as detailed in section 4.2.1. The concept of extrinsic cache miss, that will be used to tighten the estimation of the CRPD, is introduced in section 4.3.1.

4.1 CRPD Estimation

The CRPD when pre-empting a task at a given basic block is estimated using the method proposed by Negi et al in [NMR03]. The method relies on the fixed-point computation, for every basic block of:

- Reaching Cache States (RCS). The Reaching Cache States at a basic block BB_b of a program, denoted as RCS_b, is the set of possible cache states when BB_b is reached via any incoming program path. This notion captures the possible cache content when the task is pre-empted at BB_b.

- Live Cache State (LCS). The LCS at BB_b, denoted as LCS_b, is the set of memory blocks that may be referenced in the future in any outgoing program path from BB_b. This notion captures the potential reuse of memory blocks after a pre-emption point occurring at BB_b.

Cache Utility Vectors (CUV) as defined in [NMR03] are then computed, and correspond to the cache blocks that may be in the cache (in RCS) and may be reused (in LCS).

By definition, the pre-emption delay upper-bound function $f_i(t)$ holds the worst-case pre-emption penalty associated to each progress point t. This penalty value is a function of the CUV at that particular point and the cache sets used by the pre-empting tasks. In order not to consider multiple $f_i(t)$ functions (distinct functions per higher priority task), just a single ECS could be assumed for all higher priority tasks. This ECS would be computed through the union of the ECS sets of all tasks with priority greater than the considered task. In order not to overload notation, and without loss of generality, the CRPD at BB_b is then simply the delay to reload all the cache blocks in CUV_b, without considering cache usage in the pre-empting task(s) as done by Negi et al. [NMR03] and Altmeyer et al. [AB11].

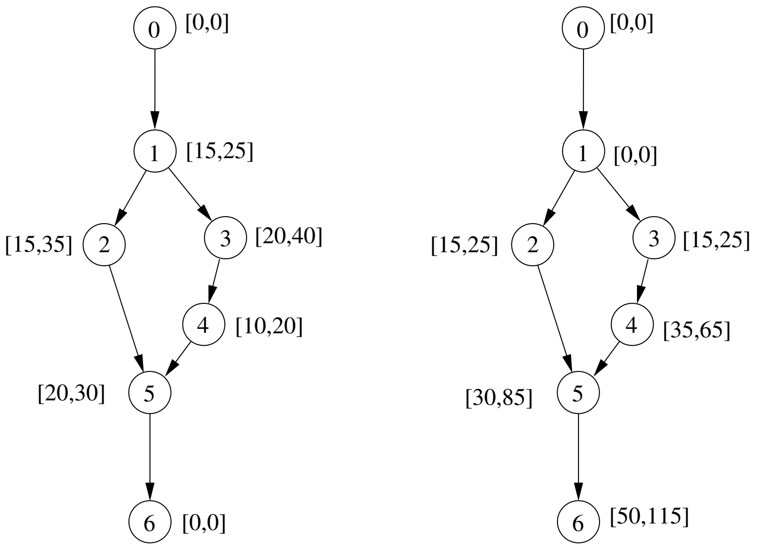
4.2 Computing Execution Intervals

Computing $f_i(t)$ for every task τ_i, represented by its control-flow graph (see left part of Figure 4.1), requires the identification of the corresponding time interval during which every basic block b of τ_i might execute.

The minimum and maximum execution time of every basic block b, noted respectively e_b^{min} and e_b^{max} have to be known. The cache analysis method used to classify every memory access uses the following categories:

- AH (Always Hit) when the access will always result in a hit

- AM (Always Miss) when the access will always result in a miss



(a). CFG with Execution Time Intervals per BB

(b). CFG with Computed Earliest and Latest Start Offsets per BB

Figure 4.1: Example of CFG for loop-free code.

- FM (First Miss) the access could neither be classified as hit nor miss the first time it occurs but will result in cache hits afterwards (this category is used for code inside loops)

- NC (Not Classified) when no precise categorization can be achieved

Given the classification of every reference, respective values of e_b^{min} and e_b^{max} are easily generated by considering the lower and higher delays allowed by the category (e.g. hit delay and miss delay for the category NC). The first iteration of every loop is virtually unrolled before applying the analysis in order to ease the analysis and still allow for the limitation of the number of references classified FM and NC.

Then, computing execution intervals on loop-free code requires to know for every basic block b its earliest and latest start offsets s_b^{min} and s_b^{max}. This can be done by a breadth-first traversal of the CFG, applying to every traversed basic block b the following formulas:

$$s_0^{min} = s_0^{max} = 0 \tag{4.1}$$

$$s_b^{min} = \min_{x \in pred(b)} (s_x^{min} + e_x^{min}) \tag{4.2}$$

$$s_b^{max} = \max_{x \in pred(b)} (s_x^{max} + e_x^{max}) \tag{4.3}$$

with *pred(b)* the direct predecessor(s) of a basic block b in the CFG, and the task entry basic block (BB_0). In the formulas, e_x^{min} (resp. e_x^{max}) represent the minimum (resp. maximum) execution time of basic block b; such values can be produced by standard WCET estimation tools (variations between e_x^{min} and e_x^{max} come for example from memory references that cannot be determined statically as hitting or missing the cache).

The right part of Figure 4.1 shows for every basic block its earliest and latest start offset after applying the above formulas. Then, the time interval within which every basic block b may execute is $[s_b^{min}, s_b^{max} + e_b^{max}]$.

The method was generalized to code with natural loops and function calls as follows. A fundamental assumption in the WCET analysis is that there exist definite bounds to the number of iterations of each loop. Every loop can then be unrolled (i.e. an equivalent CFG can be derived which does not contain any loop). The computation of execution time intervals is done on every loop starting from the innermost one, and then considering every loop as a single node with known earliest and latest start offsets. Similarly, in case of function calls, each function is analysed for every call context, starting from the leaves in the acyclic call graph.

4.2.1 Computation of $f_i(t)$

Knowing the possible execution interval $[s_b^{min}, s_b^{max} + e_b^{max}]$ of every basic block b, the set of basic blocks that may execute at a time instant t, noted $BB(t) = \left\{ \cup_{\{b | s_b^{min} \leqslant t \leqslant s_b^{max} + e_b^{max}\}} BB_b \right\}$ is known. For each basic block b in this set, $f_i(t)$ can then be computed as follows, with $CRPD_b$ the CRPD paid when pre-empting the task at basic block BB_b:

$$f_i(t) = \max_{\forall b \in \text{BB}(t)} \{\text{CRPD}_b\} \tag{4.4}$$

4.3 Determination of Pre-emption Delay Upper-bounds

As stated previously, when floating non-pre-emptive region scheduling is employed, a task will always execute non-pre-emptively for at least Q_i time units before a pre-emption occurs, unless it completes before the end of the non-pre-emptive region.

In the present discussion, the term “progression point“ is employed referring to a specific instruction in the tasks’ code. Program points are loosely associated to a time value in the task execution time frame. This means that a program point can have a minimum access time and a maximum access time but the instant at which it is reached may vary between jobs. In this framework if $p_k < p_a$ then the program point p_k precedes p_a. For clarity purposes a precedence relation is assumed between any pair of program points.

A naïve thought to upper-bound the cumulative pre-emption delay over a task’s execution (say τ_i) might be to select from f_i the maximum number of progression points p_k (each

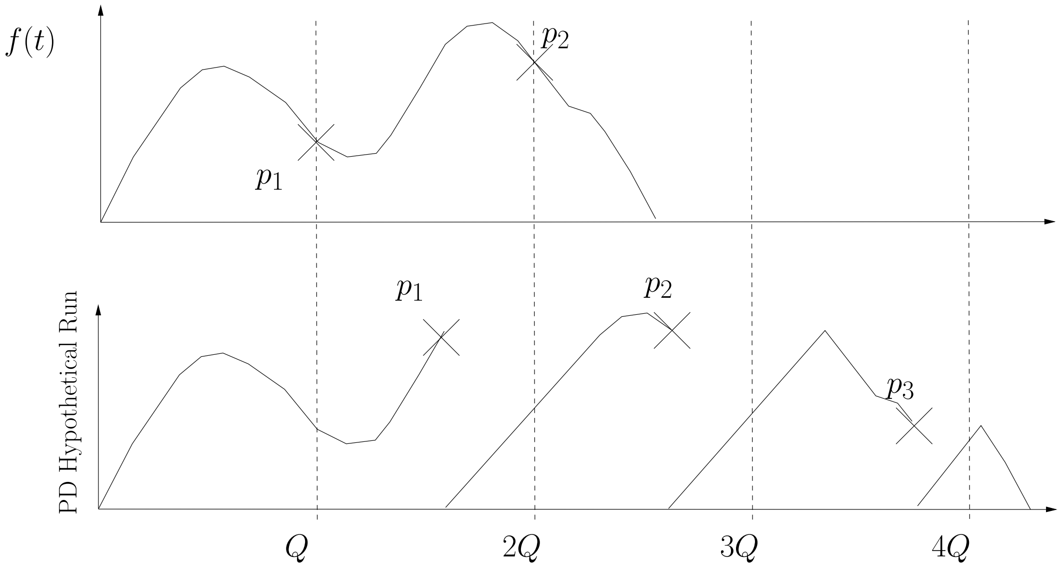


Figure 4.2: Comparison between Function f_i and the Run-time Pre-emption Delay

distanced from every other by at least Q_i time units) such that the sum $\sum_{\forall p_k} f_i(p_k)$ is maximized. However, the simple example depicted in Figure 4.2 shows that this solution is not correct. As one can see, on the top plot where f_i is depicted, there are at most two points that may be selected (since no three points could be distanced by at least Q_i time units in time). The bottom plot presents a hypothetical run of task τ_i, where the run-time pre-emption delay cost is presented. At run-time, since time is spent paying pre-emption delay after each pre-emption, more points can be selected (see the bottom plot), hence providing a higher cumulative pre-emption delay.

A pessimistic, but correct, solution to upper-bound the execution time C'_i of a task τ_i while taking into account all the possible pre-emption delays that τ_i might suffer during its execution, is simply to multiply the maximum number of pre-emptions that can occur during τ_i's execution (i.e., $\left\lfloor \frac{C_i}{Q_i} \right\rfloor$, this is discussed in more detail in the previous section) by the maximum delay of one pre-emption (i.e., $\max_{t \in [0,C_i]} f_i(t)$). Given the increase in the WCET due to this cumulative overhead, the maximum number of pre-emptions that can occur eventually increases as well. Therefore, this computation has to be performed iteratively, in the style of the well-know task response-time computation, i.e., $C_i'^{(0)} = C_i$ and

$$C_i'^{(k)} = C_i + \left\lfloor \frac{C_i'^{(k-1)}}{Q_i} \right\rfloor \times \max_{t \in [0,C_i]} f_i(t) \tag{4.5}$$

The pessimism of this computation comes directly from the fact that it considers a constant cost for every possible pre-emption, and this constant cost is assumed to be the maximum possible cost. That is, this approach is not sensitive to the pre-emption cost pattern of the task. As it was claimed in the abstract, using this additional information (the tasks

Algorithm 6: Upper-Bound the Pre-emption Delay

```
Input  : $f_i()$: pre-emption delay function of task $\tau_i$
         $Q_i$: length of the non-pre-emptive region
Output: total_delay: cumulative pre-emption delay suffered by $\tau_i$

prog ← 0 ;
total_delay ← 0;
$\text{delay}_{\max} \leftarrow 0$ ;
$p_{\text{next}} \leftarrow Q_i$ ;
/* While the next progression is not beyond $C_i$                */
while $p_{\text{next}} < C_i$ do
    /* Update time, progression and delay                         */
    prog ← $p_{\text{next}}$ ;
    /* Compute the next progression step and the next delay
       to account for                                             */
    $p_\cap \leftarrow \min\{p_x\}$ such that
            $p_x \in [\text{prog}^{(k)}, \text{prog}^{(k)} + Q_i]$
            and $f_i(p_x) = -p_x + \text{prog}^{(k)} + Q_i\}$ ;
    if $p_\cap$ = null then $p_\cap \leftarrow \text{prog} + Q_i$;
    $p_{\max} \leftarrow \arg\max_{p_x \in [\text{prog}, p_\cap]} \{f_i(p_x)\}$;
    $\text{delay}_{\max} \leftarrow f_i(p_{\max})$;
    $p_{\text{next}} \leftarrow \text{prog} + Q_i - \text{delay}_{\max}$;
    $\text{total\_delay} \leftarrow \text{total\_delay} + \text{delay}_{\max}$ ;
return total_delay ;
```

pre-emption cost pattern) enables us to derive a more accurate upper-bound. This second technique is described in Algorithm 6, and a detailed explanation is provided below.

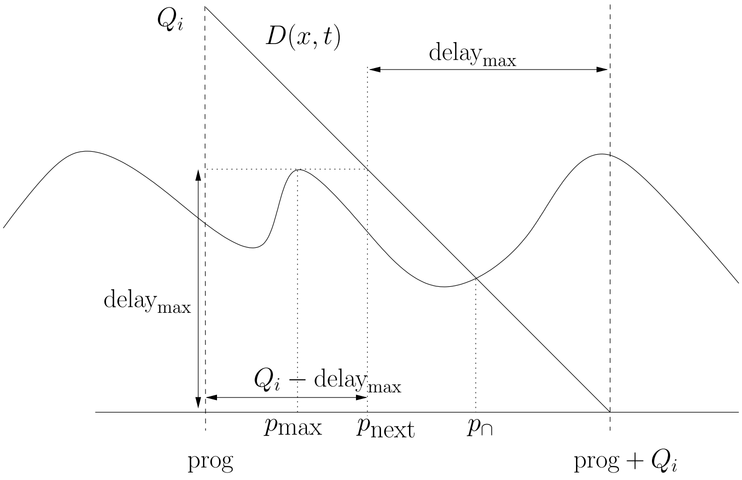


Figure 4.3: Algorithm iteration sketch

Description of Algorithm 6. Initially an explanation of the intuition behind the approach is provided on Figure 4.3 before presenting the actual algorithm. In Figure 4.3 the grey curve is the f_i function. Suppose that prog is the current progression in the task execution. Considering the next pre-emption point, the approach is looking for the lower bound on the progression which will be achieved within the next Q_i time units in any pre-emption scenario. For this, function f_i is investigated from the current prog to prog $+Q_i$. On the ordinate also at length Q_i a line $D(x,t)$ is drawn to prog $+Q_i$. The point $p_\cap$ where f first crosses $D(x,t)$ limits the range of values which need to be considered. A pre-emption past this value would lead to a situation where this point would again be considered in a subsequent iteration, since then prog would not pass this point in the current iteration. Within the interval, $\text{delay}_{\text{max}}$ is determined. That means in an interval Q_i under any pre-emption scenario at least $Q_i - \text{delay}_{\text{max}}$ progress in program execution will be achieved. It is a conservative bound as a later pre-emption means that also the non pre-emptible region will only start then. This point prog $+Q_i - \text{delay}_{\text{max}}$ will serve as new starting point.

Returning to the Algorithm 6: Lines 1–4 initialise the variables. The variable prog records the current progression in the task's operations while total_delay records the cumulative pre-emption delay accounted for up to the current progression point. As the task τ_i executes, it accounts for progressing in its execution (and the variable prog is increased) and for the pre-emption delay (which updates the variable total_delay). The algorithm is iterative, and at each iteration the variables $\text{delay}_{\text{max}}$ and p_{next} (lines 3 and 4) are the pre-emption delay taking place only in the current iteration and the next progression point in τ_i's execution at which the next iteration will start, respectively. Lines 1–4 can be seen as the first iteration of the algorithm. $\text{delay}_{\text{max}}$ is set to 0 and p_{next} to Q_i, because no pre-emption can occur during the first Q_i time units of τ_i's execution.

The algorithm starts iterating at line 5, and it iterates as long as the next computed progression point p_{next} does not fall beyond τ_i's execution boundary. Line 6 shifts the current progression point of τ_i to the computed value p_{next}. Then, lines 12 and 13 compute the next progression point p_{next} and the maximum delay that τ_i could suffer while progressing in its operations from its current progression point to p_{next}. Finally, line 14 adds this maximum delay to the current cumulative delay accounted so far.

In the following Theorem 6, it is proven that the value returned by Algorithm 6 is an upper-bound on the cumulative pre-emption delay that the given task τ_i might suffer during its execution. This implies that the WCET of τ_i (while taking into account all the possible pre-emption delays that τ_i might suffer during its execution) is given by

$$C_i' \overset{\text{def}}{=} C_i + \text{total\_delay} \tag{4.6}$$

where total_delay is the value returned by Algorithm 6.

Theorem 6. *Algorithm 6 returns an upper-bound on the pre-emption delay that a given task τ_i can suffer during the execution of any of its jobs.*

Proof. Algorithm 6 computes the maximum cumulative pre-emption delay iteratively, by progressing step by step through the execution of the task τ_i. Hereafter, the notation $\text{prog}^{(k)}$ is employed to denote the progression through τ_i's execution at the beginning of the k^{th} iteration of the algorithm. Similarly, $\text{total\_delay}^{(k)}$ will be used to denote the cumulative pre-emption delay that τ_i has suffered until it reached a progression of $\text{prog}^{(k)}$. In this proof, it is shown that at each iteration $k > 0$, $\text{total\_delay}^{(k)}$ provides an *upper-bound* on the cumulative pre-emption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ in its execution. The proof is made by induction.

Basic step. Algorithm 6 first considers that τ_i progresses by Q_i time units in its execution without suffering any pre-emption delay (since it cannot get pre-empted during these first Q_i time units). This first step is devised as the first iteration of the algorithm. That is, straightforwardly, $\text{total\_delay}^{(1)} = 0$ is an upper (and even exact) bound on the cumulative pre-emption delay that τ_i may suffer before reaching a progression of Q_i time units in its execution.

Induction step. It is assumed (by induction) that $\text{total\_delay}^{(k)}$, $k > 1$, is an upper-bound on the cumulative pre-emption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ time units in its execution.

During the k^{th} iteration, Algorithm 6 computes $\text{prog}^{(k+1)}$ and $\text{total\_delay}^{(k+1)}$ as follows:

$$\begin{aligned}\text{prog}^{(k+1)} &= \text{prog}^{(k)} + Q_i - \text{delay}_{\max} && (4.7)\\ \text{total\_delay}^{(k+1)} &= \text{total\_delay}^{(k)} + \text{delay}_{\max} && (4.8)\end{aligned}$$

where

$$\begin{aligned}\text{delay}_{\max} &= f_i(p_{\max}) && (4.9)\\ p_{\max} &= \underset{p_x \in [\text{prog}^{(k)}, p_\cap]}{\arg\max} \{f_i(p_x)\} && (4.10)\\ p_\cap &= \min\{p_x\} \text{ such that} && (4.11)\\ &\quad p_x \in [\text{prog}^{(k)}, \text{prog}^{(k)} + Q_i] \\ &\text{and } f_i(p_x) = -p_x + \text{prog}^{(k)} + Q_i\end{aligned}$$

Equations 4.7 and 4.8 can be interpreted as follows. During the k^{th} iteration, Algorithm 6 assumes that τ_i executes for Q_i time units during which τ_i progresses by $Q_i - \text{delay}_{\max}$ units of time in its execution and suffers from a delay of $\text{delay}_{\max}$; The algorithm assumes that τ_i gets pre-empted when its progression reaches $p_{\max}$ given by Equation (4.10). Below we show that choosing any other pre-emption point $p_{\text{other}} \neq p_{\max}$ would ultimately, when τ_i's execution will be completed, result in a cumulative pre-emption delay lower than the one returned by Algorithm 6, thus showing that the value returned by Algorithm 6 is an upper-bound. Only two cases are possible: $p_{\text{other}} > p_{\text{next}}$ or $p_{\text{other}} \leqslant p_{\text{next}}$.

Case 1: $p_{\text{other}} > p_{\text{next}}$. This means that τ_i progresses in its execution until it reaches p_{next} *without being pre-empted*, i.e., from a progression of $\text{prog}^{(k)}$, τ_i reaches a progression of p_{next} by being executed only for $(p_{\text{next}} - \text{prog}^{(k)})$ time units, and with an unchanged cumulative pre-emption delay of $\text{total\_delay}^{(k)}$. On the other hand, in the execution scenario built by Algorithm 6, τ_i's execution reaches a progression of $\text{prog}^{(k+1)} = p_{\text{next}}$ by being executed for Q_i time units, and with a cumulative pre-emption delay of $\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \geq \text{total\_delay}^{(k)}$. In other words, Algorithm 6 manages to progress slower in τ_i's execution while suffering from a greater pre-emption delay. Furthermore, p_{other} is still a candidate pre-emption point for a further iteration of Algorithm 6.

Case 2. $p_{\text{other}} \leqslant p_{\text{next}}$. *After executing τ_i for Q_i time units*, the following facts hold

1. the delay of the pre-emption that occurs when τ_i's progression reaches p_{other} has been totally accounted for (since $p_{\text{other}} < p_{\text{next}} \leqslant p_{\cap}$).

2. the progression of τ_i in this scenario becomes

$$\begin{aligned} \text{prog}_{\text{other}} &= \text{prog}^{(k)} + Q_i - f_i(p_{\text{other}}) \\ &\geqslant \text{prog}^{(k)} + Q_i - f_i(p_{\max}) \\ &\geqslant \text{prog}^{(k+1)} \end{aligned} \tag{4.12}$$

3. the cumulative pre-emption delay becomes

$$\begin{aligned} \text{total\_delay}_{\text{other}} &= \text{total\_delay}^{(k)} + f_i(p_{\text{other}}) \\ &\leqslant \text{total\_delay}^{(k)} + f_i(p_{\max}) \\ &\leqslant \text{total\_delay}^{(k+1)} \end{aligned} \tag{4.13}$$

Thus, after executing τ_i for Q_i time units Algorithm 6 progressed less in the execution of τ_i (Inequality 4.12) while suffering from a higher pre-emption delay (Inequality 4.13). As a consequence of Cases 1 and 2, it holds at each iteration of Algorithm 6 that choosing to pre-empt the task when it reaches a progression of $p_{\max}$ ultimately leads to an upper-bound on its cumulative pre-emption delay. □

4.3.1 Extrinsic Cache Miss Function

Until now the pre-emption delay is assumed to be paid immediately following a pre-emption. By analysing the task code it is possible to extract information on when the pre-emption delay may be paid. This is captured through the concept of the *Extrinsic Cache Misses* or "Miss Function".

Definition 3 [Extrinsic Cache Miss]: *A memory access resulting in a cache miss due to the prior eviction of the requested cache line by code not belonging to the current task is termed extrinsic cache miss.*

The extrinsic cache miss function $G_i(t)$ is an upper bound on the number of extrinsic cache misses a task might suffer, multiplied by the maximum time penalty to service a cache miss, in the interval $[0,t]$. $G_i(t)$ is then an upper-bound on the pre-emption delay that might have been paid in the same interval. Using the $G_i(t)$ information it is possible to provide pre-emption delay estimations in the presented framework for situations where $Q_i \leqslant \max_t(f_i(t))$, whereas in the methodology provided in Algorithm 6 this is not possible. This subsection is devoted to the explanation of the procedure to compute this $G_i(t)$ function.

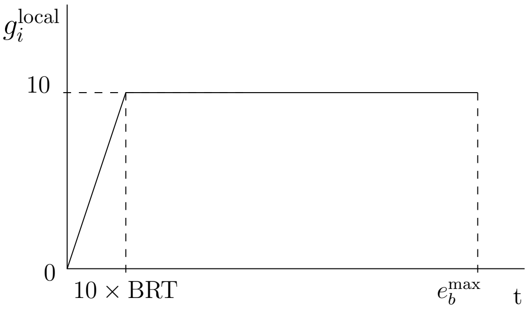


Figure 4.4: Example g_b^{local} Function Where BB$_b$ Execution May Generate at Most 10 Extrinsic Cache Misses

Each basic block has a fixed number of memory requests which may lead to extrinsic cache miss during its execution. For each BB$_i$ this number is given by

$$\text{extrinsic-miss}_b = |RCS_b \cap gen_b| \tag{4.14}$$

Recall that gen_b is defined as the set of memory blocks accessed during the execution of BB$_b$ and RCS_b is the set of memory blocks that may be in the cache while BB$_b$ is being executed. The intersection of both sets at each BB$_b$ effectively yields the upper bound on the number of extrinsic cache misses that may occur while BB$_b$ is executed after a pre-emption.

For each BB$_b$ a function can then be constructed. This function g_b^{local}, has the memory requests that may lead to an extrinsic cache miss in BB$_b$. These requests are assumed to occur as early as possible and at the maximum rate at which they can occur, as is displayed in Figure 4.4. This function is defined in the interval $[0, e_b^{\max}]$, which is the maximum length execution time interval for BB$_b$.

Let us define the following constant BRR modeling the block reload rate:

$$\text{BRR} \overset{\text{def}}{=} \frac{T_{\text{HIT}} + T_{\text{MISS}}}{T_{\text{HIT}}}$$

where T_{HIT} is the time to serve a cache hit, and T_{MISS} is the time to serve a cache miss.

The constant BRR imposes a restriction on the maximum rate in comparison to progression that pre-emption delay may be paid. Assuming BRR $\neq \infty$ ensures that while paying pre-emption delay progression is still occurring, albeit at a slower pace. Intuitively BRR is an upper bound on the maximum rate at which cache miss penalties may be generated while executing the program.

Formally the g_b^{local} function is defined per basic block as:

$$g_b^{\text{local}} \overset{\text{def}}{=} \begin{cases} \text{BRR} \times t & ,\text{if} 0 \leqslant t \leqslant \frac{\text{extrisic-miss}_b}{BRR} \\ \text{extrisic-miss}_b & ,\text{if} \frac{\text{extrisic-miss}_b}{BRR} < t \leqslant e_b^{\max} \end{cases} \tag{4.15}$$

A task-wide g^{local} function is constructed starting from the first BB$_1$ of the CFG. An initial function g_1^{in} is fed into the CFG. Where $g_1^{\text{in}}(t) = 0, \forall t$. The input functions for each BB$_b$ are then merged together with g_b^{local} as is shown in Figure 4.5

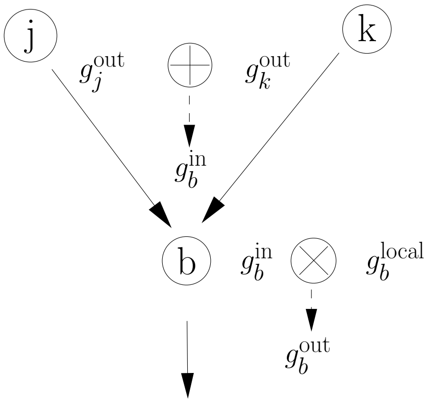


Figure 4.5: g_b^{out} Computation for BB$_b$

The two merge operations used to conduct the computation of the task-wide extrinsic cache misses function are defined next. Each block outputs a function g_b^{out}. Accordingly each basic block has one input function g_b^{in}. This input function is a product of the merging of all the output functions of the BB$_b$ parent nodes. The input merging operation is defined by:

$$g_j^{\text{out}} \oplus g_k^{\text{out}} \overset{\text{def}}{=} \max_{t \in [0, C_i]} (g_j^{\text{out}}(t), g_k^{\text{out}}(t)) \tag{4.16}$$

The input function is then merged with the corresponding node-specific g_b^{local} function using the merge-at-node operation defined in the following way:

$$g_b^{\mathrm{local}} \otimes g_b^{\mathrm{in}} \overset{\mathrm{def}}{=} \begin{cases} g_b^{\mathrm{in}}(t) & , t \leqslant s_b^{\mathrm{min}} \\ g_b^{\mathrm{in}}(t) + g_b^{\mathrm{local}}(t - s_b^{\mathrm{min}}) & , s_b^{\mathrm{min}} < t < C_i \end{cases} \tag{4.17}$$

This assumes the knowledge of the earliest time the node may be accessed (s_b^{min}).

The g_n^{out} where BB$_n$ is the last basic block in the CFG (return block) is the task-wide G_i function. In the case that several return blocks exist then the task-wide G_i function is obtained by combining the g_n^{out} functions of all the return blocks using the merge-at-edge operator.

$$G_i \overset{\mathrm{def}}{=} \bigoplus_{j \in \mathrm{RET}} g_j^{\mathrm{out}} \tag{4.18}$$

The procedure described is graphically explained in Figure 4.5. In this figure only one BB$_b$ is portrayed for clarification. The procedure repeats for all the BB$_b$ of the CFG.

Theorem 7. *The Function $G_i(t)$ is an upper-bound on the extrinsic cache misses occurring during the execution of task τ_i at any time t.*

Proof. The function g_1^{in} is an upper bound on the number of extrinsic cache misses for the time instant 0 since there can occur zero extrinsic cache misses until this time instant. For each basic block all the memory accesses which may generate an intrinsic cache miss are considered to occur as early as possible (s_b^{min}), at the maximum possible rate (BRR). Hence g_b^{local} is an upper bound on the extrinsic cache misses occurring in BB$_b$. Then, the function g_1^{out} is an upper-bound on the number of extrinsic cache misses until the execution of the task exist BB$_1$. The same function g_1^{out} is then the g_b^{in} for all the BB$_b$ child nodes of BB$_1$.

Since the merge-at-node operation integrates the maximum number of extrinsic cache misses occurring in BB$_b$ at the earliest time that these could occur (s_b^{min}) and at the maximum rate (BRR) Then it holds true that merge-at-node operation carried out in each of BB$_1$ child preserves the property that each g_b^{out} is itself an upper-bound on the number of extrinsic cache misses occurring from the start of task execution until it took the path leading to BB$_b$.

If a node BB$_b$ has more than one parent then g_b^{in} is constructed using the merge-at-edge operator over all the node predecessors output functions (g_j^{out}). Since the merge-at-node operation takes the maximum value for all the $t \in [0, C_i]$ of the parents g_j^{out} output functions then it holds true that g_b^{in} is still an upper-bound on the extrinsic cache misses that could occur from the start of task execution until it took the path leading to BB$_b$.

This reasoning holds true for all nodes in the CFG. Lastly to compute the $G_i(t)$ function the merge-at-edge operator is applied across all the return edges of the CFG. Hence the $G_i(t)$ function is an upper bound on all the extrinsic cache misses occurring for all the possible execution paths in task τ_i □

4.3.2 Pre-emption Delay Computation using Extrinsic Cache-miss Function

For the upper bound computation of pre-emption delay, two functions ($f_i(t)$ and $G_i(t)$) are considered. The $f_i(t)$ function represents an upper bound on the pre-emption delay a task may face at any point in time, whereas the G_i function yields the upper bound on the amount of pre-emption delay in the time interval $[0,t]$.

Using the knowledge provided by $f_i(t)$ function alone prevents the application of the method in scenarios where $Q_i \leqslant \max_t(f_i(t))$. This was one of the major limitations of the method presented in the solution provided in Algorithm 6. This constraint is now relaxed in Algorithm 7. This fact arises from a situation where no progression can be performed since there is always more pre-emption delay to be paid than the length of the considered non-pre-emptive execution region.

As previously stated a task will always execute non-pre-emptively for at least Q_i time units before a pre-emption occurs, unless it completes before the end of the non-pre-emptive region.

Description of the Pre-emption Delay Estimation Algorithm 7. As before for Algorithm 6, the intuition behind the approach on Figure 4.6 is explained before presenting the actual algorithm. Suppose that prog is the current progression in the task execution. Considering the next pre-emption point, the approach is looking for the lower bound on the

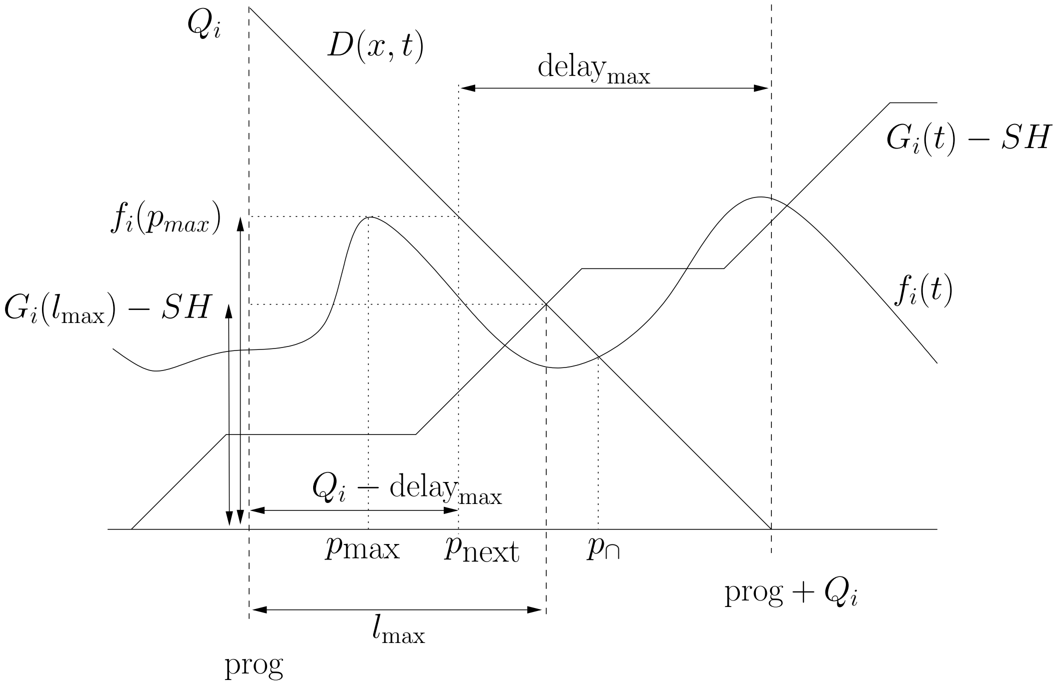


Figure 4.6: Algorithm Iteration Sketch

Algorithm 7: Upper-Bound the Pre-emption Delay

```
Input  : $f_i(t)$: pre-emption delay function of task $\tau_i$.
         $G_i(t)$: extrinsic cache misses function of task $\tau_i$.
         $Q_i$: length of the non-pre-emptive region
Output: total_delay: cumulative pre-emption delay suffered by $\tau_i$
total_delay $\leftarrow 0$;
delay$_{\max} \leftarrow 0$ ;
$p_{\text{next}} \leftarrow Q_i$ ;
$SH \leftarrow 0$ ;
/* While the next progression is not beyond $C_i$ */
while $p_{\text{next}} < C_i$ do
    /* Update time, progression and delay */
    prog $\leftarrow p_{\text{next}}$ ;
    /* Compute the next progression step and the next delay
       to account for, based on the $f_i(t)$ function */
    $p_\cap \leftarrow \min\{p_x\}$ such that
        $p_x \in [\text{prog}, \text{prog} + Q_i]$
        and $f_i(p_x) = -p_x + \text{prog} + Q\}$ ;
    if $p_\cap = $ null then $p_\cap \leftarrow \text{prog} + Q_i$;
    $p_{\max} \leftarrow \arg\max_{p_x \in [\text{prog}, p_\cap]} \{f_i(p_x)\}$;
    $l_{\max} = \min((t | G_i(t) - SH = -t + \text{prog} + Q_i), \text{prog} + Q_i)$;
    if $G_i(l_{\max}) - SH \leqslant f_i(p_{\max})$ then
        delay$_{\max} \leftarrow G_i(l_{\max}) - SH$;
        $SH = G_i(l_{\max})$;
    else
        delay$_{\max} \leftarrow f_i(p_{\max})$;
        $SH = SH + f_i(p_{\max})$;
    delay$_{\max} \leftarrow \min(f_i(p_{\max}), G_i(l_{\max}))$;
    $p_{\text{next}} \leftarrow \text{prog} + Q_i - \text{delay}_{\max}$;
    total_delay $\leftarrow$ total_delay + delay$_{\max}$ ;
return total_delay ;
```

progression which will be achieved within the next Q_i time units in any pre-emption scenario. For this, functions $f_i(t)$ and $G_i(t)$ are investigated from the current prog to prog $+Q_i$. On the ordinate also at length Q_i a line $D(x,t)$ is drawn to prog $+Q_i$.

Two intersection points with the $D(x,t)$ function are obtained for $f_i(t)$ and $G_i(t)$ with an offset of SH units. The variable SH represents the pre-emption delay currently considered from the $G_i(t)$ function. It ensures that double accounting of pre-emption delay does not occur. One is the point $p_\cap$ where $f_i(t)$ first crosses $D(x,t)$. It limits the range of values which need to be considered for the $f_i(t)$ function as previously stated (Algorithm 6). This intersecting point limits the pre-emption delay values to be considered since the assuming the worst-case pre-emption penalty to be paid no further progression past the intersection point would be possible.

Within this interval, $\text{delay}_{\text{max}}$ is determined. That means in an interval Q_i under any pre-emption scenario at least $Q_i - \text{delay}_{\text{max}}$ progress in program execution will be achieved. It is a conservative bound as a later pre-emption means that also the non pre-emptible region will only start then. This point $\text{prog} + Q_i - \text{delay}_{\text{max}}$ will serve as new starting point.

The second point is l_{max}. At l_{max} the function $G_i(t) - \text{SH}$ intersects $D(x,t)$. At this point $G_i(l_{\text{max}}) - \text{SH}$ cache misses have occurred in the $[\text{prog}, l_{\text{max}}]$ interval. If only considering the $G_i(t)$ information, a progression of $\text{prog} + Q - G_i(l_{\text{max}}) - \text{SH}$ would be assumed, where in fact a comparatively smaller quantity of $\text{prog} - G_i(l_{\text{max}}) - \text{SH}$ time units is spent reloading cache content.

Returning to the Algorithm 7: Lines 1–4 initialise the variables. The variable prog memorizes the current progression in the task's operations while total_delay records the cumulative pre-emption delay accounted for up to the current progression. As the task τ_i executes, it accounts for progressing in its execution (and the variable prog is increased) and for the pre-emption delay (which updates the variable total_delay). The algorithm is iterative, and at each iteration the variables $\text{delay}_{\text{max}}$ and p_{next} (lines 2 and 3) are the pre-emption delay taking place only in the current iteration and the next progression point in τ_i's execution at which the next iteration will start, respectively. Lastly in line 4 the variable SH, which represents the offset of the $G_i(t)$ function, is initialised with zero. Lines 1–4 can be seen as the first iteration of the algorithm. $\text{delay}_{\text{max}}$ is set to 0 and p_{next} to Q_i, because no pre-emption can occur during the first Q_i time units of τ_i's execution.

The algorithm starts iterating at line 5, and it iterates as long as the next computed progression point p_{next} does not fall beyond τ_i's execution boundary. Line 6 shifts the current progression point of τ_i to the computed value p_{next}. In lines 7 – 11 the $f_i(t)$ pre-emption delay computation for the current iteration is carried out. Subsequently in lines 11 and 12 the pre-emption delay computation is carried out with the $G_i(t)$ function information.

From line 13 to 19 a decision is carried out on the amount of pre-emption delay to consider in the current iteration. This value is the minimum between the one estimated with the $G_i(t)$ function and the $f_i(t)$ function, since both functions hold an upper bound on the pre-emption delay for a given interval.

The next progression point p_{next} is computed with the knowledge of the maximum delay that τ_i could suffer while progressing in its operations from its current progression point to p_{next}. Finally, line 21 adds this maximum delay to the current cumulative delay accounted so far.

In the following Theorem 8, it is proven that the value returned by Algorithm 7 is an upper-bound on the cumulative pre-emption delay that the given task τ_i might suffer during its execution. This implies that the WCET of τ_i (while taking into account all the possible pre-emption delays that τ_i might suffer during its execution) is computed through Equation (4.6) where total_delay is the value returned by Algorithm 7.

Theorem 8. *Algorithm 7 returns an upper-bound on the pre-emption delay that a given task τ_i can suffer during the execution of any of its jobs.*

Proof. Algorithm 7 computes the maximum cumulative pre-emption delay iteratively, by progressing step by step through the execution of the task τ_i. In this proof, it is shown that at each iteration $k > 0$, $\text{total\_delay}^{(k)}$ actually provides an *upper-bound* on the cumulative pre-emption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ in its execution. The proof is made by induction.

Basic step.

Algorithm 7 first considers that τ_i progresses by Q_i time units in its execution without suffering any pre-emption delay (since it cannot get pre-empted during these first Q_i time units). Similarly to Algorithm 6 this is the first step as the first iteration of the algorithm. That is, straightforwardly, $\text{total\_delay}^{(1)} = 0$ is an upper (and even exact) bound on the cumulative pre-emption delay that τ_i may suffer before reaching a progression of Q_i time units in its execution.

Induction step.

It is assumed (by induction) that $\text{total\_delay}^{(k)}$, $k \geqslant 1$, is an upper-bound on the cumulative pre-emption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ time units in its execution.

During the k^{th} iteration, Algorithm 7 computes $\text{prog}^{(k+1)}$ and $\text{total\_delay}^{(k+1)}$ as follows:

$$\text{prog}^{(k+1)} = \text{prog}^{(k)} + Q_i - \text{delay}_{\max} \tag{4.19}$$

$$\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \tag{4.20}$$

where

$$\text{delay}_{\max} = \min(f_i(p_{\max}), G_i(l_{\max}) - SH) \tag{4.21}$$

$$p_{\max} = \operatorname*{arg\,max}_{p_x \in [\text{prog}^{(k)}, p_\cap]} \{f_i(p_x)\} \tag{4.22}$$

$$\begin{aligned} p_\cap = {} & \min\{p_x\} \text{ such that} \\ & \quad p_x \in [\text{prog}^{(k)}, \text{prog}^{(k)} + Q_i] \\ & \text{and } p_x = \text{prog}^{(k)} + Q_i - \min(f_i(p_x), G_i(p_x) - SH) \end{aligned} \tag{4.23}$$

$$l_{\max} = \min((t | G_i(t) - SH = -t + \text{prog} + Q), \text{prog} + Q) \tag{4.24}$$

Equations 4.19 and 4.20 can be interpreted as follows. During the k^{th} iteration, Algorithm 7 assumes that τ_i executes for Q_i time units during which τ_i progresses by $Q_i - \text{delay}_{\max}$ units of time in its execution and incurs a delay of $\text{delay}_{\max}$; The algorithm assumes that τ_i gets pre-empted when its progression reaches $p_{\max}$ given by Equation (4.22). Below we show that choosing any other pre-emption point $p_{\text{other}} \neq p_{\max}$ would ultimately[1] result in a cumulative pre-emption delay lower than the one returned by Algorithm 7, thus showing that the value returned by Algorithm 7 is an upper-bound. Two cases may arise: $p_{\text{other}} > p_{\text{next}}$ or $p_{\text{other}} \leqslant p_{\text{next}}$.

Case 1: $p_{\text{other}} > p_{\text{next}}$. This means that τ_i progresses in its execution until it reaches p_{next} *without being pre-empted*, i.e., from a progression of $\text{prog}^{(k)}$, τ_i reaches a progression of p_{next} by being executed only for $(p_{\text{next}} - \text{prog}^{(k)}) \leqslant Q_i$ time units, and with an unchanged cumulative pre-emption delay of $\text{total\_delay}^{(k)}$. On the other hand, in the execution scenario built by Algorithm 7, τ_i's execution reaches a progression of $\text{prog}^{(k+1)} = p_{\text{next}}$ by being executed for Q_i time units, and with a cumulative pre-emption delay of $\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \geqslant \text{total\_delay}^{(k)}$. In other words, Algorithm 7 manages to progress slower in τ_i's execution while suffering from a greater pre-emption delay. Furthermore, p_{other} is still a candidate pre-emption point for a further iteration of Algorithm 7.

Case 2. $p_{\text{other}} \leqslant p_{\text{next}}$. *After executing τ_i for Q_i time units*, the following facts hold

1. the delay of the pre-emption that occurs when τ_i's progression reaches p_{other} has been totally accounted for (since $p_{\text{other}} < p_{\text{next}} \leqslant p_{\cap}$).

2. the progression of τ_i in this scenario becomes

$$\begin{aligned} \text{prog}_{\text{other}} &= \text{prog}^{(k)} + Q_i - \min(f_i(p_{\text{other}}), G_i(p_{\text{other}}) - SH) \\ &\geqslant \text{prog}^{(k)} + Q_i - \min(f_i(p_{\max}), G_i(p_{\max}) - SH) \\ &\geqslant \text{prog}^{(k+1)} \end{aligned} \tag{4.25}$$

3. the cumulative pre-emption delay becomes

$$\begin{aligned} \text{total\_delay}_{\text{other}} &= \text{total\_delay}^{(k)} + \min(f_i(p_{\text{other}}), G_i(p_{\text{other}}) - SH) \\ &\leqslant \text{total\_delay}^{(k)} + \min(f_i(p_{\max}), G_i(p_{\max}) - SH) \\ &\leqslant \text{total\_delay}^{(k+1)} \end{aligned} \tag{4.26}$$

Thus, after executing τ_i for Q_i time units Algorithm 7 progressed less in the execution of τ_i (Inequality 4.25) while suffering from a higher pre-emption delay (Inequality 4.26). As a consequence of Cases 1 and 2, it holds at each iteration of Algorithm 7 that choosing to

[1] when τ_i's execution will be completed

pre-empt the task when it reaches a progression of $p_{\max}$ ultimately leads to an upper-bound on its cumulative pre-emption delay. □

4.3.3 Reducing the pessimism of $f_i(t)$

For computation of the pre-emption delay estimation the obtained $f_i(t)$ function is pessimistic. The main source of pessimism is the fact that the earliest possible entry time for each basic block is considered in the function computation. Even though the $f_i(t)$ function is an upper bound on the pre-emption delay that might be paid at any progression point t in the task a less pessimistic function may be extracted.

In fact if a basic block is accessed earlier than its worst case access time then the execution is in a situation which will never lead to the WCET. If the task is pre-empted in this basic block at an earlier time than the worst case access time, the pre-emption delay that has to be paid is already doubly accounted on the WCET computation.

To exemplify this intuition consider the example provided in Figure 4.1 (page 75). For the case of BB$_4$, $s_4^{min} = 35$, $s_4^{max} = 65$ and $e_4^{max} = 20$. Lets arbitrate CRPD$_4 = 5$. If this BB$_4$ is accessed at $t = 40$, and a pre-emption occurs at $t' = 45$, when the task resumes its execution at most 7 units of time will be spent regaining cache state. The BB$_4$ exit time would be $t'' = 65$ whereas the worst case exit time computed statically is $s_4^{max} + e_4^{max} = 85$. One can then observe that $t'' < s_4^{max} + e_4^{max}$. The conclusion follows that: pre-emptions occurring in BB$_4$ at an earlier time in comparison to the worst-case execution time behaviour do not necessitate to consider the full worst-case pre-emption delay penalty. Clearly t'' is smaller than the statically computed worst-case exit time and hence it does not make sense to consider a pre-emption delay of 5 units when the basic block is accessed at time $t = 40$. A more thorough definition and explanation of the concept follows.

The optimised $f_i(t)$ function is then defined as:

$$f_i(t) = \max_b \{f_b^{\text{local}}(t)\} \tag{4.27}$$

The function $f_b^{\text{local}}(t)$ is defined for each BB_b. It represents the pre-emption delay value the tasks is subject to when pre-empted in each basic block but refrains from considering the full pre-emption delay cost when the basic block is accessed earlier than in the worst-case time access profile.

The function $f_b^{\text{local}}(t)$ is formally defined in the following manner:

$$f_b^{\mathrm{local}}(t) \overset{\mathrm{def}}{=} \begin{cases} 0 & , 0 \leqslant t < \max(s_b^{max} - \mathrm{CRPD}_b, s_b^{min}) \\ \mathrm{CRPD}_b \times (t - s_b^{max} - \mathrm{CRPD}_b) & , \max(s_b^{max} - \mathrm{CRPD}_b, s_b^{min}) \leqslant t \leqslant s_b^{max} \\ \mathrm{CRPD}_b & , s_b^{max} < t \leqslant e_b^{max} \\ 0 & , t > e_b^{max} \end{cases} \tag{4.28}$$

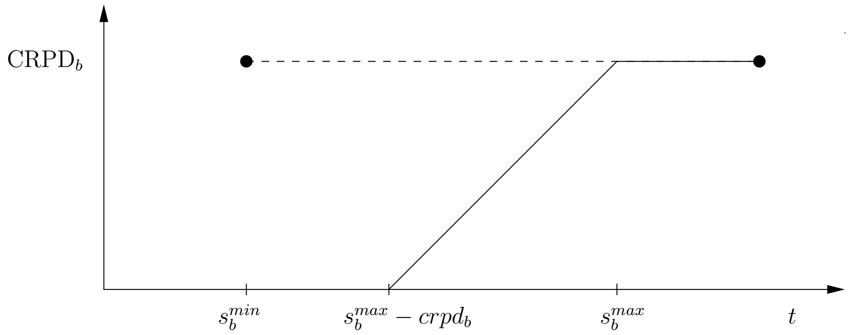


Figure 4.7: $f_b^{\mathrm{local}}(t)$ Graphical Example

Figure 4.7 presents a graphical representation of the $f_b^{\mathrm{local}}(t)$ function. From s_b^{min} to $s_b^{max} - CRPD_b$ the value of the function $f_b^{\mathrm{local}}(t)$ is zero. In the interval $s_b^{max} - CRPD_b$ to s_b^{max} the function $f_b^{\mathrm{local}}(t)$ has a first derivative of one. Lastly, from s_b^{max} to $s_b^{max} + e_b^{max}$ the function is constant at the $CRPD_b$ value.

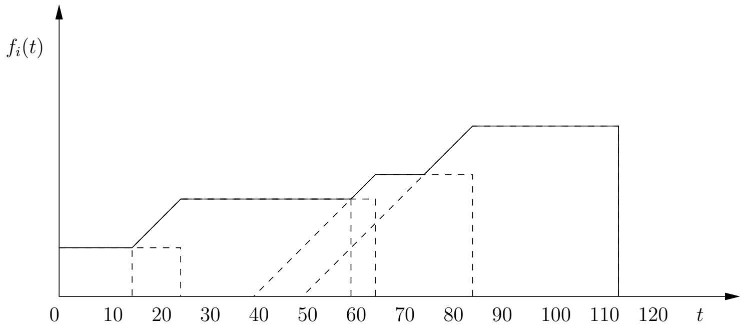


Figure 4.8: Task Wide $f_i(t)$ Obtention

In Figure 4.8 the procedure to compute the task-wide $f_i(t)$ function is graphically portrayed. Several $f_b^{\mathrm{local}}(t)$ function are displayed with dashed lines. For all the instants in time, the maximum value of all the local $f_b^{\mathrm{local}}(t)$ is taken as the value of $f_i(t)$.

Lemma 2. *Assuming an entry time t_1 such that $s_b^{max} - CRPD_b \leqslant t_1 < s_b^{max}$ and a pre-emption occurring at t_1, the progression point achieved in a Q_i length time window assuming a pre-emption delay payment of $f_b^{\mathrm{local}}(t_1)$ is equal to the progression point assuming an entry time $t_2 = s_b^{\mathrm{max}}$.*

Proof. $f_b^{\mathrm{local}}(t_1) = t_1 - (s_b^{\mathrm{max}} - \mathrm{CRPD}_b)$

$\mathrm{prog}_1 = Q_i - t_1 + (s_b^{\mathrm{max}} - \mathrm{CRPD}_b) + t_1$

$\mathrm{prog}_2 = Q_i - \mathrm{CRPD}_b + t_2$

$\mathrm{prog}_1 = \mathrm{prog}_2 \Leftrightarrow Q_i - (t_1 - (s_b^{\mathrm{max}} - \mathrm{CRPD}_b)) + t_1 = Q_i - \mathrm{CRPD}_b + t_2 \Leftrightarrow Q_i + s_b^{\mathrm{max}} = Q_i + t_2$.

Since $t_2 = s_b^{\mathrm{max}}$ the Lemma is proven □

There exits a set of possible paths from BB_1 to BB_b such that for all these possible program paths BB_b will be accessed at a time instant p' in the program where $p' \in [s_b^{min}, s_b^{max}]$. For all possible paths where the entry time of BB_b for which $p' < s_b^{max}$ then the execution is earlier in relation to the worst-case execution time by $s_b^{max} - p'$ time units.

When computing the $f_i(t)$ function for each progression point considered the point with the maximum pre-emption delay is chosen. It is the case that for the paths which diverge from the worst-case execution time behaviour then paying pre-emption delay in this paths would just bring them closer to the worst case execution time behaviour.

As long as the progression is sufficiently distanced from the worst-case pattern for BB_b execution then no pre-emption delay has to be considered, since this would lead to a double accounting of the pre-emption delay.

Consider a situation where $s_b^{max} - p' > \mathrm{CRPD}_b$, if the task would be executing in BB_b, by paying the pre-emption delay of CRPD_b would not bring this execution to the worst-case execution time scenario. In the same way if $0 < s_b^{max} - p' < \mathrm{CRPD}_b$ then paying a portion of the pre-emption delay payment given by $s_b^{max} - p'$ would bring the execution exactly into the worst case scenario. From this point onwards the maximum pre-emption delay for the basic block should be paid since the progression on BB_b is already in a worst-case scenario.

Lemma 3. *For all the* BB_b *that a given progression point p might belong to, the pre-emption delay payment is given* $\max_b\left\{\max(\min(\mathrm{CRPD}_b - (s_b^{max} - p), \mathrm{CRPD}_b), 0)\right\}$.

Proof. For a given BB_b, and a current actual progression point p, if $\mathrm{CRPD}_b - (s_b^{max} - p) < 0$ then for BB_b is $s_b^{max} - p$ time units earlier in relation to the worst-case entry time behaviour for the given block. Paying CRPD_b in this instance is not putting it in a worse situation than the worst-case entry $s_b^{max} - p$ and hence is not leading a optimistic analysis result. □

Theorem 9. *The Function $f_i(t)$, considering f_b^{local}, provides a safe upper bound on the pre-emption delay paid during a pre-emption of* BB_b *in a WCET context.*

Proof. In Lemma 2 it is shown that any scenario where a basic block is considered to be accessed between $s_b^{max} - \text{CRPD}_b$ and s_b^{max} will be treated such that is equivalent of accessing BB_b at time s_b^{max}. Additionally, in Lemma 3 it is proven that BB_b arriving before $s_b^{max} - \text{CRPD}_b$ the actual progression in the real execution is at least as good as arriving at s_b^{max}. Hence, using $f_i(t)$ provides a safe upper bound for the pre-emption delay to be taken into account during analysis.

□

4.3.4 Reducing the pessimism of $G_i(t)$

Following the same reasoning as in section 4.3.3, the $G_i(t)$ function holds pessimism for the pre-emption delay computation in a WCET analysis context.

The merge at node operation is then defined as:

$$g_b^{\text{local}} \otimes g_b^{\text{in}} \overset{\text{def}}{=} \begin{cases} g_b^{\text{in}}(t) & ,\text{if } t \leqslant \max(s_b^{\max} - g_i^{\text{local}}(e_b^{\max}), s_b^{\min}) \\ \max(g_b^{\text{in}}(t) + g_b^{\text{local}}(t - \max(s_b^{\max} - & g_i^{\text{local}}(e_b^{\max}), s_b^{\min}))) \\ & ,\text{if } \max(s_b^{\max} - g_b^{\text{local}}(e_b^{\max}), s_b^{\min}) < t < C_i \end{cases} \tag{4.29}$$

In this way the pre-emption delay is only accounted for when the task execution is in a path that would lead into the WCET.

Theorem 10. *The Function $G_i(t)$ is an upper-bound on the extrinsic cache misses occurring during the execution of task τ_i for any time t in a WCET context.*

Proof. The redefinition of the merge-at-node operation integrates the g_b^{local} function only at the time instant $t = \max(s_b^{\max} - g_b^{\text{local}}(e_b^{\max}), s_b^{\min})$. For all the time instants $t' < t$ if the BB_b is executing and generates an extrinsic cache miss then this value was already accounted for in the WCET analysis since:
$t' + g_b^{\text{local}}(e_b^{\max}) + e_b^{\max} < s_b^{\max} + e_b^{\max} \Leftrightarrow t' + g_b^{\text{local}}(e_b^{\max} < s_b^{\max} \Leftrightarrow t' < s_b^{\max} - g_b^{\text{local}}(e_b^{\max})$

The only change to the computation procedure of $G_i(t)$ lies on the merge-at-node operation. Hence proving the correctness of the merge-at-node operation redefinition along with Theorem 7 proves the present theorem.

□

4.4 Experimental Evaluation

In order to experimentally validate both the new pre-emption delay estimation algorithm 7 the $f_i(t)$ and $g^{\text{local}}(t)$ function extraction procedures were implemented in Heptane. Results are also provided for the previous function $f_i(t)$ (Algorithm 6).

The framework is trialled on a set of benchmarks available online [NCS+06]. Not all the benchmark results are presented. From the set of benchmark programs the results for acquisition task, autopilot task 6, autopilot task 9 and interrupt handler routine are shown.

All the results are given for a direct-mapped instruction cache of 4KB, with a line size of 32 bytes (8 instructions). The data cache is perfect (i.e. data cache misses are assumed never to occur). Only results regarding instruction caches are presented at this time since data caches were not yet addressed theoretically. The analysed code is MIPS code (fixed-size 4B instructions). The code also works with set-associative caches. There is only one level of cache, with 1 cycle when the fetched instruction hits the cache, 10 cycles in case of a miss.

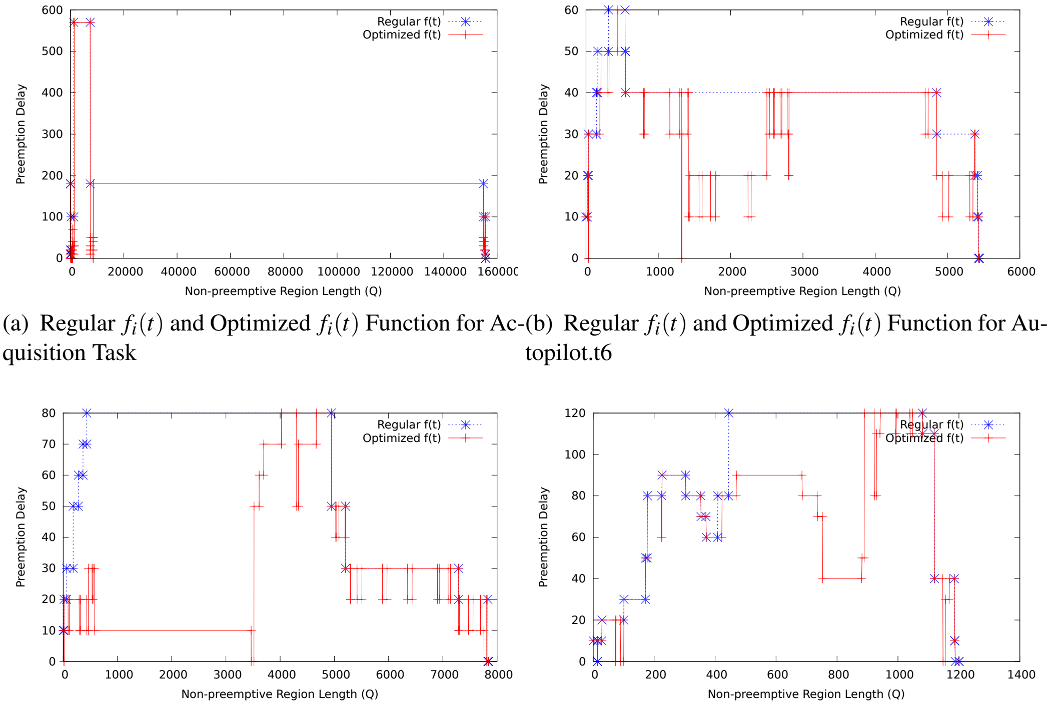
4.4.1 $f_i(t)$ functions

A comparative evaluation between the prior $f_i(t)$ function extraction procedure and the more recent one is presented.

As is apparent in all the benchmarks the current method enables a considerable increase in the amount of information present in the new $f_i(t)$ functions. The regular $f_i(t)$ (Equation (4.4)) is always greater than the optimized version (Equation (4.27)). With this new concepts it is then possible to decrease the level of pessimism present in the analysis of the pre-emption delay. The major differences are present in the results shown in Figures 4.9b, 4.9c and 4.9d. For these benchmarks, the results of the optimized $f_i(t)$ function are lesser pessimistic than the prior method presented initially in section 4.3.

4.4.2 Pre-emption Delay Estimations

The pre-emption delay estimations are presented in this subsection. Only the improved version of the $f_i(t)$ is used, since it is obvious that the results could never be worse than using the old version of the function. In all the results present in Figures 4.10a to 4.10d it is apparent that the results are less pessimistic both for the previous method (which only uses $f_i(t)$ information) and for the new method with the G_i knowledge in comparison to the naïve state of the art method (Equation (4.5)). The naive state of the art method in Equation 4.5 is an intuitive upper-bound on the pre-emption delay for the floating non-pre-emptive regions scheduling. Its results are the curves labeled as “SOTA” in figures 4.10a to 4.10d. The method using the G_i function information is never worse than the one which relies purely



(a) Regular $f_i(t)$ and Optimized $f_i(t)$ Function for Acquisition Task

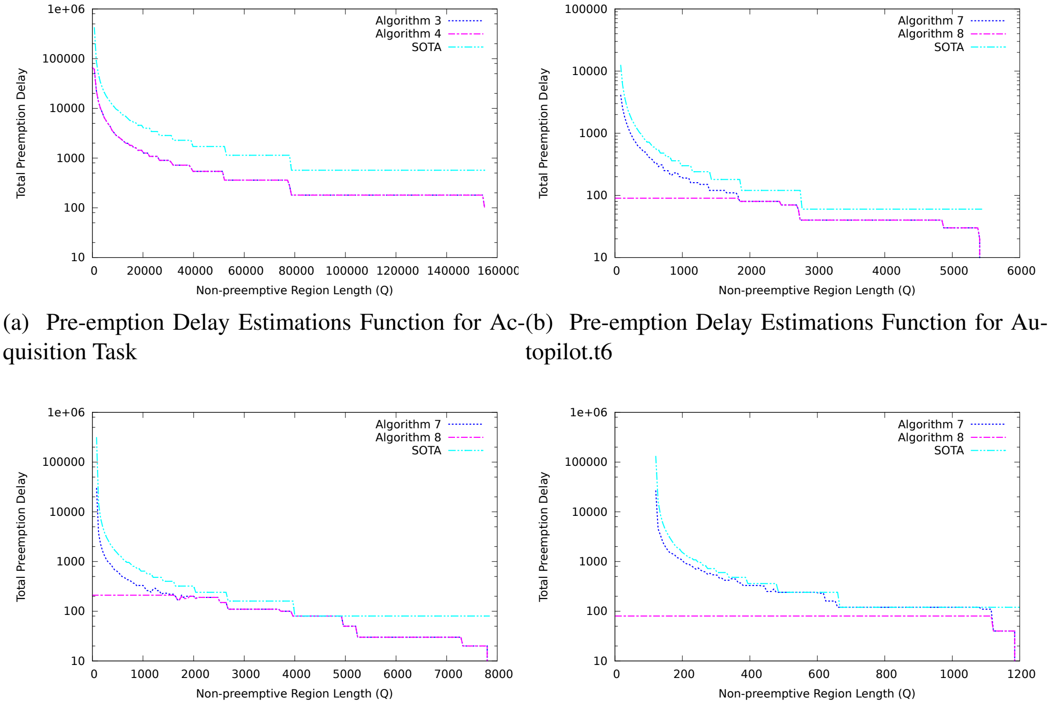
(b) Regular $f_i(t)$ and Optimized $f_i(t)$ Function for Autopilot.t6

(c) Regular $f_i(t)$ and Optimized $f_i(t)$ Function for Autopilot.t9

(d) Regular $f_i(t)$ and Optimized $f_i(t)$ Function for Interrupt Handler

Figure 4.9: $f_i(t)$ Experimental Results

on the $f_i(t)$ function, and enables solutions to be obtained for situations where Q_i is lower than some value of $f_i(t)$ as is shown in figures 4.10a to 4.10d.



(a) Pre-emption Delay Estimations Function for Acquisition Task

(b) Pre-emption Delay Estimations Function for Autopilot.t6

(c) Pre-emption Delay Estimations Function for Autopilot.t9

(d) Pre-emption Delay Estimations Function for Interrupt Handler

Figure 4.10: $g^{\text{local}}t$ Experimental Results

Chapter 5

Temporal-isolation Enforcement

Before a safety-critical system can be deployed and marketed, a certification authority must validate that all the safety-related norms are met. All the components comprising that system (the software, the hardware, and the interfaces) are scrutinized to ensure conformance to safety standards.

To provide the required degree of "sufficient independence" between components of different SILs, Industry and Academy have always been working in close collaboration, seeking solutions to (1) render the components of a same SIL as independent and isolated as possible from the components with different SILs and (2) upper-bound the residual impact that components of different SILs may have on each other after the segregation step, with the primary objective of certifying each subset of components at its own SIL.

This chapter presents a novel solution for ensuring the temporal-isolation property in systems where pre-emptions are subject to non-negligible overheads. In this case a *symmetric* isolation is enforced, i.e. where tasks do not impact each other irrespective of their individual SIL. A prior version of this framework has been presented in a previous work [MNP14]. A standard implementation of the symmetric temporal-isolation is by the usage of servers [AB04], [LKPB06]. These servers provide a certain share of the processing resource called budget, which is supplied to a task in a recurrent fashion.

While the general concepts of servers have been well explored, the use of implicitly shared resources, like caches is still an open issue for server-based systems. When a task executing in a server is pre-empted by a higher priority task, it loses at least partially its set of useful memory blocks in the caches (working set) and other shared resources. This loss of working set leads to an additional execution time requirement on resumption of execution, which either needs to be accounted for in the sizing of the budgets of individual tasks, or treated through online mechanisms.

Besides easing timing analyses, schedulability analyses, and the certification process, the main objective of temporal-isolation via servers is also to isolate applications from other temporally misbehaving applications and their corresponding effects. This misbehaviour can come in two distinct flavours. Firstly, in terms of a violation of the WCET assumptions, secondly in the minimum inter-arrival time assumption made in the analysis step. While the former has been treated to reasonable extent [LKPB06], the latter is of equal importance.

In this chapter a mechanism which enforces temporal isolation is presented. The devised solution is able to accommodate both forms of misbehaviour. At the end of the chapter an experimental evaluation attestss to the efficiency and benefit extracted from the usage of said mechanism.

5.1 Chapter-wise Update on System Model

When tackling a given problem the simplest possible system model which allows for an adequate property manipulations is employed. Generally a system model as presented in section 1.2 offers a robust footing on which to develop and prove theories for real-time systems. Unfortunately the model presented in section 1.2 relies on a fundamental assumption which is generally not present in the real world. That of which the task's assumed properties at design time are stringently met at runtime. Tasks are written according to a specification. It is the job of the developer entity to ensure that in fact the WCET is met in the given platform, that the minimum inter-arrival time is in fact observed (i.e. that under any circumstance will a given task request to execute at a greater frequency than anticipated).

As in section 1.2, the workload is modelled by a task-set $\mathscr{T} = \{\tau_1, \ldots, \tau_n\}$ composed of n tasks. Each task τ_i is characterized by the three-tuple $\langle C_i, D_i, T_i \rangle$ with the following interpretation: τ_i generates a potentially infinite sequence of jobs, with the first job released at any time during the system execution and subsequent jobs released *at least* T_i time units apart. Each job released by τ_i must execute for *at most* C_i time units within D_i time units from its release. Hence, the parameter C_i is referred to as the "worst-case execution time", D_i is the relative deadline of the task and T_i is its minimum inter-release time (often called, its period).

Contrary to the bulk of this document where the three parameters C_i, D_i and T_i are assumed to fully characterize and describe the tasks and its interaction with the system (at least the interactions which are of interest to this work – duration of resource usage), in this chapter these parameters solely represent the terms of an agreement between the task and the system.

This concept reflects an inherent distrust from the system point of view towards the implementation that embodies each task and the nature of the events that might trigger its execution – which may in turn be external to the embedded system itself. In order for each task's behaviour not to suffer from external misbehaviour the system has to ensure that each party respects its contract.

A task τ_i is said to "behave" if it does not require more system resources than by its contracted parameters. Otherwise, if any job of τ_i comes to request more than C_i time units to complete, or if τ_i releases two consecutive jobs in a time interval $< T_i$ time units, then τ_i is said to be "misbehaving". The other party – the system – is assumed to never violate its contracts with any task. The system associates to each task τ_i a *sporadic* server S_i defined by the two-tuple $\langle B_i, T_i^s \rangle$. The parameter B_i encodes the execution budget that S_i provides to τ_i in any time window of length T_i^s. This budget is consumed as task τ_i executes and a task can only execute if its budget is not depleted. The function $B_i(t)$ denotes the remaining budget in the server S_i at every time instant t.

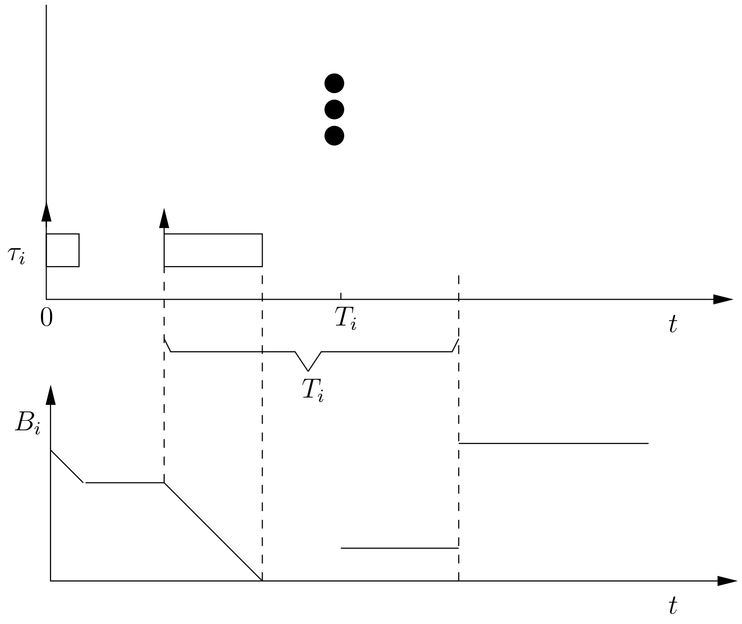


Figure 5.1: Sporadic Server Budget Replenishment

A sporadic server is, at any time instant, in either one of the two following states:

active when there is pending workload from task τ_i **and** $B_i(t) > 0$;

idle when there is no pending workload from task τ_i **or** $B_i(t) = 0$.

The sporadic server budget replenishment mechanics can be described succinctly by the protocol formulated with the two following rules:

- When the server transits to the **Active** state at a time t_1, a recharging event is set to occur at time instant $t_1 + T_i^s$;
- when S_i transits to the **idle** state at a time t_2, the replenishment amount corresponding to the last recharging time is set to the amount of capacity consumed by S_i in the interval $[t1, t2)$.

At the start of the system ($t = 0$) S_i is **idle** and $B_i(t) = C_i$. An example of the sporadic server budget usage and replenishment is provided in Figure 5.1. For sake of simplicity and clarity of exposure the server replenishment period is arbitrarily assumed to be equal to the tasks reported period (i.e. $T_i^s = T_i$) hence, T_i is used throughout this chapter as a synonym for T_i^s.

From this point onward, it is assumed that all the task deadlines are met at run-time as long as every job of each task τ_i executes within the execution budget granted by S_i and respects its timing parameters C_i and T_i. The framework proposed here ensures that, though any task τ_i can misbehave by violating its stated parameters, the other tasks in the system will never miss their deadlines as long as *they* behave. Note that it is assumed throughout this chapter that each server has only a single task associated to it. The server and task terms are used interchangeably in the remainder of the chapter.

5.2 Pre-emption Delay Accounting Approaches Comparison

In a reservation-based system, as previously stated, each task τ_i can only execute as long as $B_i(t)$ is greater than 0. If every job is guaranteed to meet its deadlines, then at each time t where task τ_i releases a job, it must hold that $B_i(t)$ is greater than or equal to C_i plus the maximum pre-emption delay that the job may be subject to during its execution. The variable $\delta_{j,i}$ represents the maximum interference that a task τ_j may induce in the execution time of task τ_i by pre-empting it. This maximum interference can be computed by using methods such as the ones presented in [AB09, RM06a, LLH$^+$01].

Given all these $\delta_{j,i}$ values, a naive solution to compute the budget B_i of each task $\tau_i \in \mathscr{T}$ is amenable to be devised. If it is assumed that task τ_j releases its jobs *exactly* T_j time units apart, then the maximum number of jobs that τ_j can release in an interval of time of length t is given by

$$n_j(t) \overset{\text{def}}{=} \left\lceil \frac{t}{T_j} \right\rceil \tag{5.1}$$

Therefore, during the worst-case response time of a task τ_i denoted by R_i, there are at most $n_j(R_i)$ jobs of task τ_j, $j < i$, that can potentially pre-empt τ_i (recall that if $j < i$ then $\tau_j \in hp(i)$). Since each of these pre-emptions imply an interference of at most $\delta_{j,i}$ time units on the execution of τ_i, a straightforward way to compute the budget B_i assigned to each task τ_i to meet all its deadlines is

$$B_i \overset{\text{def}}{=} C_i + \sum_{j=1}^{i-1} n_j(R_i) \times \delta_{j,i} \tag{5.2}$$

For the budget assignment policy defined in Equation (5.2), Equation (5.3) gives an upper-bound $\text{PD}_{\text{bgt}}^{\max}(t)$ on the total CPU time that is reserved in any time interval $[0,t]$ to account for all the pre-emption delays.

$$\begin{aligned}\mathrm{PD}_{\mathrm{bgt}}^{\max}(t) &\overset{\mathrm{def}}{=} \sum_{i=2}^{n}\left(n_i(t)\times\sum_{j=1}^{i-1} n_j(R_i)\times\delta_{j,i}\right)\\ &= \sum_{i=2}^{n}\left(n_i(t)\times\sum_{j=1}^{n} n_j(R_i)\times\delta_{j,i}\right)\end{aligned}\tag{5.3}$$

as $\forall\ j \geqslant i$ it holds that $\delta_{j,i}=0$.

It is worth noticing that Equation (5.2) assigns the budget of task τ_i by looking at how many times τ_i might get pre-empted during the execution of each of its jobs and how much each such pre-emption may cost. That is, this budget assignment policy implicitly considers the problem from the point of view of the *pre-empted* task.

An alternative approach to analyse the maximum pre-emption delay that a task can incur consists in considering the problem from the point of view of the *pre-empting* task. An example of such an approach has been presented by Stachulat et al [SSE05]. The authors defined the multi-set $M_{j,i}(t)$ as the set of all costs $\delta_{j,k}$ that tasks τ_j may induce in the execution requirements of all the tasks τ_k with a priority between that of τ_j and τ_i, in a time window of length t. A multi-set is a generalisation of the concept of "set" where the elements may be replicated (i.e. a multi-set may be for example $\{x,x,x,y,y\}$ whereas a regular set consists of a collection of elements such that no element is equal to any other in the same set). The multi-set $M_{j,i}(t)$ is formally defined at any time t as follow:

$$M_{j,i}(t) \overset{\mathrm{def}}{=} \left(\biguplus_{k=j+1}^{i-1}\biguplus_{m=1}^{n_k(t)}\biguplus_{\ell=1}^{n_j(R_k)}\delta_{j,k}\right)\biguplus_{g=1}^{n_j(t)}\delta_{j,i}\tag{5.4}$$

The operator $\uplus$ denotes the union over multi-sets. Let us look at a brief example to differentiate between the multi-set union and the set union. For the multi-set union it holds that $\{x,y\}\uplus\{x,y\}=\{x,x,y,y\}$ whereas for the set union the outcome is $\{x,y\}\cup\{x,y\}=\{x,y\}$.

Each set $M_{j,i}(t)$ enables the construction of the function $\Delta_{j,i}(t)$, denoting the maximum pre-emption delay caused by jobs from task τ_j on task τ_i in any time window of length t.

$$\Delta_{j,i}(t) \overset{\mathrm{def}}{=} \sum_{\ell=1}^{q_{j,i}(t)} \overset{\ell}{\max}(M_{j,i}(t))\tag{5.5}$$

where

$$q_{j,i}(t) \overset{\mathrm{def}}{=} \sum_{k=j}^{i-1}\min(n_k(t),n_j(t))\tag{5.6}$$

and the function $\overset{\ell}{\max}(M_{j,i}(t))$ returns the ℓth highest value in the set $M_{j,i}(t)$ – the equation $\Delta_{j,i}(t) \overset{\mathrm{def}}{=} \sum_{\ell=1}^{q_{j,i}(t)} \overset{\ell}{\max}(M_{j,i}(t))$ thus represents the sum of the $q_{i,j}(t)$ highest values in $M_{j,i}(t)$.

It is shown below that, considering the pre-emption delay from the perspective of the *pre-empting* task is always less pessimistic than considering the pre-emption delay from the point of view of the *pre-empted* task.

Theorem 11. *For each task $\tau_i \in \mathscr{T}$, it holds at any time t that*

$$\sum_{j=1}^{n} \Delta_{j,i}(t) \leqslant \mathrm{PD}_{\mathrm{bgt}}^{\max}(t) \tag{5.7}$$

Proof. From Equation (5.4) and since $\forall\ j \geqslant i$ it holds that $\delta_{j,i} = 0$, for all $\tau_j \in \mathscr{T}$ the sum of all elements in $M_{j,i}(t)$ is given by:

$$\begin{aligned} \sum_{e \in M_{j,i}(t)} e &= \sum_{k=j+1}^{i} \sum_{m=1}^{n_k(t)} \sum_{\ell=1}^{n_j(R_k)} \delta_{j,k} \\ &= \sum_{k=j+1}^{i} n_k(t) \times n_j(R_k) \times \delta_{j,k} \end{aligned} \tag{5.8}$$

The remainder of the proof is split into two lemmas that will straightforwardly yield Equation (5.7).

Lemma 4. $\sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e \leqslant \mathrm{PD}_{\mathrm{bgt}}^{\max}(t)$

Proof. From Equation (5.8), it is known that

$$\begin{aligned} \sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e &\leqslant \sum_{j=1}^{n} \sum_{k=2}^{n} n_k(t) \times n_j(R_k) \times \delta_{j,k} \\ &\leqslant \sum_{k=2}^{n} n_k(t) \times \sum_{j=1}^{n} n_j(R_k) \times \delta_{j,k} \\ \{from(5.3)\} &\quad \mathrm{PD}_{\mathrm{bgt}}^{\max}(t) \end{aligned}$$

□

Lemma 5. $\sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e \geqslant \sum_{j=1}^{n} \Delta_{j,i}(t)$

Proof. On the one hand, one can observe from Equation (5.4) that the number of elements in the multiset $M_{j,i}(t)$ is given by

$$\#M_{j,i}(t) = \left(\sum_{k=j+1}^{i-1} n_k(t) \times n_j(R_k) \right) + n_j(t) \tag{5.9}$$

and since $n_j(R_k) \geqslant 1, \forall j,k \in [1,n]$, it holds that

$$\#M_{j,i}(t) \geqslant n_j(t) + \sum_{k=j+1}^{i-1} n_k(t) \tag{5.10}$$

On the other hand, if $j = k$ then

$$\min(n_k(t), n_j(t)) = \min(n_j(t), n_j(t)) = n_j(t)$$

and thus this Equation (5.6) can be rewritten as:

$$\begin{aligned} q_{j,i}(t) &= n_j(t) + \sum_{k=j+1}^{i-1} \min(n_k(t), n_j(t)) \\ &\leqslant n_j(t) + \sum_{k=j+1}^{i-1} n_k(t) \end{aligned} \tag{5.11}$$

By combining Equations (5.10) and (5.11), it thus holds $\forall j, i \in [1, n]$ that

$$q_{j,i}(t) \leqslant \#M_{j,i}(t) \tag{5.12}$$

Remember that $\sum_{\ell=1}^{q_{j,i}(t)} \overset{\ell}{\max}(M_{j,i}(t))$ represents the sum of the $q_{i,j}(t)$ highest values in $M_{j,i}(t)$. From Inequality (5.12) and by definition of the function $\overset{\ell}{\max}(M_{j,i}(t))$, it can be concluded that $\forall t > 0$ and for all $\tau_i, \tau_j \in \mathscr{T}$:

$$\sum_{e \in M_{j,i}(t)} e \geqslant \sum_{\ell=1}^{q_{i,j}(t)} \overset{\ell}{\max}(M_{j,i}(t)) \tag{5.13}$$

By summing Inequality (5.13) over all $j \in [1, n]$, then

$$\begin{aligned} \sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e &\geqslant \sum_{j=1}^{n} \sum_{\ell=1}^{q_{i,j}(t)} \overset{\ell}{\max}(M_{j,i}(t)) \\ &\geqslant \sum_{j=1}^{n} \Delta_{j,i}(t) \end{aligned}$$

Hence the lemma follows. □

Finally, by combining Lemmas 4 and 5 it is easy to see that

$$\sum_{j=1}^{n} \Delta_{j,i}(t) \leqslant \mathrm{PD}_{\mathrm{bgt}}^{\max}(t)$$

□

5.3 Proposed Budget Augmentation Framework

5.3.1 Temporal-isolation Framework Description

Since a task may incur some delay due to a pre-emption, it is straightforward that an execution budget of $B_i = C_i$ may be insufficient for the task τ_i to complete if it gets pre-empted during its execution. On the other hand, the budget assignment policy defined by Equation (5.2) has been shown to be (potentially) pessimistic. Hence, a run-time mechanism where *every pre-empting task has to pay for the damage that it causes to the schedule* is proposed. According to Theorem 11, accounting for the pre-emption delay from the point of view of the pre-empting task enables a reduction on the over-provisioning of system resources. Formally, the execution budget B_i of each task τ_i is initially set to C_i and refilled according to the sporadic server definition. Then, each time a task τ_i resumes its execution after being pre-empted by other task(s), the remaining budget $B_i(t)$ of its associated server S_i is increased by $\sum_{\tau_j \in H(i)} \delta_{j,i}(t)$ (where $H(i)$ denotes the set of tasks that pre-empted τ_i) to compensate for the potential extra execution requirement that τ_i may incur.

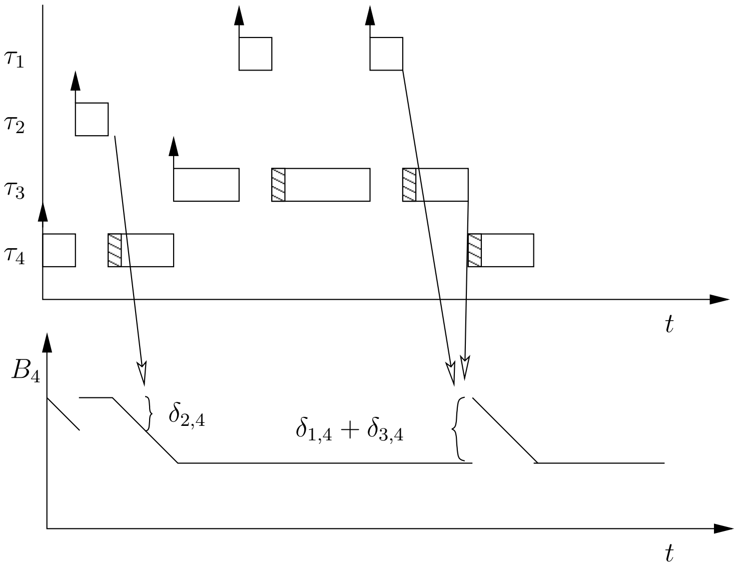


Figure 5.2: Budget Augmentation Example

An example of the described framework is presented in Figure 5.2. In that example the task set contains 4 tasks. Task τ_4 is first pre-empted by a job from τ_2. When τ_4 resumes execution at time t_1, immediately after τ_2 terminates, its remaining budget $B_4(t_1)$ is incremented by $\delta_{2,4}$ units. Then, two jobs of τ_1 pre-empt both τ_3 and (indirectly) τ_4. Each time τ_3 resumes its execution at the return from the pre-emption (at time t_2 and t_3), the execution budget $B_3(t_2)$ and $B_3(t_3)$ is incremented by $\delta_{1,3}$. Finally, when τ_3 terminates its workload and τ_4 resumes at time t_4, $B_4(t_4)$ is incremented by $\delta_{1,4}+\delta_{3,4}$ as both τ_1 and τ_3 may have evicted some of its cached data; hence forcing τ_4 to reload it from the memory.

5.3.2 Temporal-isolation Schedulability Analysis

When the pre-emption delay is assumed to be zero (i.e., when the cache subsystem is partitioned for example), the authors of [LSD89a] proposed the following schedulability test to check at design-time, whether all the task deadlines are met at run-time.

Schedulability Test 1 (From [LSD89a])**.** *A task set $\mathscr{T}$ is schedulable if, $\forall \tau_i \in \mathscr{T}$, $\exists t \in (0, D_i]$ such that*

$$C_i + \sum_{j=1}^{i-1} \mathrm{rbf}(S_j, t) \leqslant t \tag{5.14}$$

where

$$\mathrm{rbf}(S_j, t) \overset{\text{def}}{=} \left(\left\lfloor \frac{t}{T_j^s} \right\rfloor + 1 \right) \times B_j \tag{5.15}$$

Theorem 12 (from [SSL89])**.** *A periodic task-set that is schedulable with a task τ_i, is also schedulable if τ_i is replaced by a sporadic server with the same period and execution time*

Proof. According to the sporadic server protocol, every budget amount consumed by task τ_i starting at time instant t, is only replenished at $t+T_i$. The value $B_i = C_i$ assuming direct relationship between the task τ_i WCET and the corresponding server budget. The budget B_i can be consumed in the first instance in any conceivable fragmentation patters with initial active transitions at time instants $\{t_1, t_2, \cdots, t_k\}$ and corresponding execution requirements $\{\ell_1, \ell_2, \cdots, \ell_k\}$. By definition $\sum_{g=1}^{k} \ell_h \leqslant C_i$. Let us define an arrival curve modeling the worst-case workload request by a sporadic server assuming a several active transition events in the first interval $[0, T_i]$ and consecutive requests at the maximum possible rate in the future. A function upper-bounding the execution requirements for any t can be constructed as:

$$rbf_{chunk}^{\ell_i} = \left(\left\lceil \frac{t - t_\ell}{T_i} \right\rceil \right)_0 \times H_\ell$$

Straightforwardly the relation $rbf(\tau_i, t) \geqslant \sum_{\ell=1}^{k} rbf_{chunk}^{\ell_i}$ is obtained.

By generalizing the fragmented situation from the first period to multiple ones it the execution requirement considered for the sporadic task is always greater or equal to the worst-case budget consumption by the server associated to τ_i for any possible interval length. Hence the Lemma follows.

□

The correctness of the schedulability test 1 comes as a direct consequence of the Theorem 12 as the presented test is the one for a task-set composed of periodic tasks [LSD89a].

As introduced earlier, if every task τ_i augments its budget for $\delta_{j,i}$ time units after being pre-empted by a task τ_j, then an upper bound on the total budget augmentation in any time window of length t is given by $\sum_{j=1}^{i-1} \Delta_{j,i}(t)$.

It can be shown that in any given time window of length t, an upper-bound on the number of execution resumptions in a schedule is given by $q_{1,i}(t)$ since according to its definition the maximum possible number of job releases of tasks with priority greater than i, in an interval $[0,t]$, is accounted for. Therefore, assuming that performing each execution of the budget augmentation consumes F_{cost} units of time, the time-penalty attached to the implementation of the proposed framework has an upper bound of

$$\text{delay}(t) = q_{1,i}(t) \times F_{\text{cost}} \tag{5.16}$$

Integrating these quantities into Schedulability Test 1 yields the following test:

Schedulability Test 2. *A task set $\mathscr{T}$ is schedulable if, $\forall \tau_i \in \mathscr{T}$, $\exists t \in (0, D_i]$ such that*

$$C_i + \text{delay}(t) + \sum_{j=1}^{i-1} [\text{rbf}(S_j, t) + \Delta_{j,i}(t)] \leqslant t \tag{5.17}$$

Correctness of Schedulability Test 2. Function (5.6) upper-bounds the number of times that jobs from task τ_j may pre-empt jobs of priority lower than τ_j and higher or equal than τ_i in a window of length t. Function (5.5) ($\Delta_{j,i}(t)$) is the summation over the $q_{j,i}(t)$ largest values in the multi-set $M_{j,i}(t)$. The function $\Delta_{j,i}(t)$ is then an upper-bound on the amount of pre-emption delay compensation that can be extracted from task τ_j from any task of priority lower than τ_j and higher or equal than τ_i in a window of length t. Thus $\sum_{j=1}^{i-1} \Delta_{j,i}(t)$ is an upper-bound on the pre-emption delay compensation budget used by tasks of priority higher or equal than τ_i for any time t. As a consequence and by the correctness of schedulability test 1, the correctness of this schedulability test is proven. □

According to Schedulability Test 2 and as a consequence of Theorem 11, assuming $\text{delay}(t) = 0$ then the proposed framework enables a higher schedulability than considering the budget B_i of each server S_i to be equal to C_i plus the maximum pre-emption delay that any job of τ_i may be potentially subject to (see Equation (5.2)). However, in a scenario where $\text{delay}(t)$ is non-negligible the dominance relation does no longer hold.

5.4 Proposed Budget Donation Framework

The framework presented above is a combination of a reservation-based mechanism (budget initially assigned to each task) and a budget augmentation policy (budget inflated at return from pre-emption). This combination ensures that the temporal-isolation property is always

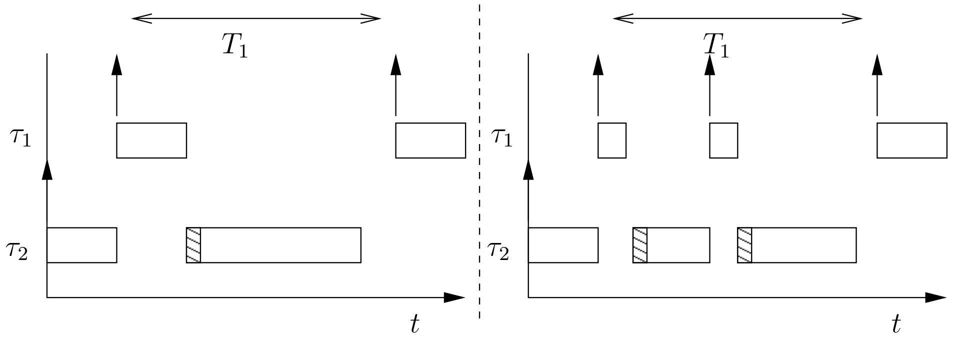


Figure 5.3: Excessive Pre-emption Delay Due to Minimum Interarrival Time Violation

met as long as none of the tasks violates its minimum inter-arrival constraint, i.e., as long as none of them release two consecutive jobs in a time interval shorter than its pre-defined period T_i. This condition of not violating the minimum inter-arrival constraint is implicitly assumed by Equations (5.5) and (5.6), in which the upper-bound on the pre-emption delay interference inherently relies on the number of jobs released by every task in a given time window.

If any task violates its minimum inter-arrival time constraint, the temporal-isolation property no longer holds. An example of this is depicted in Figure 5.3. On the left-hand side of the picture, task τ_1 releases a single job in a time interval of length T_1. This job executes for C_1 time units and task τ_2 suffers only from one pre-emption, leading to an increase of $\delta_{1,2}$ on its execution requirement. In the right-hand side of the picture, τ_1 releases more than one job in the same time interval of length T_1 and τ_2 now suffers from 2 pre-emptions, leading to an increase of $2 \times \delta_{1,2}$ on its execution requirement. In the latter scenario, task τ_2 augments its budget accordingly but may face a deadline miss, or a lower priority tasks may be subject to more interference than what was accounted for in the schedulability test.

In order to avoid this issue, a second server Y_i is associated to each task τ_i. This server Y_i has parameters $\langle Z_i, T_i^Y \rangle - Z_i$ is the budget and T_i^Y is the replenishment period. Unlike the server S_i, the budget Z_i is not consumed while τ_i is running. The purpose of this second server Y_i is to "pay" for the damage caused by τ_i in the system when τ_i pre-empts another task. That is, *each* task τ_j, when it is pre-empted by τ_i, obtains a budget donations by transferring some execution budget from the server Y_i to its execution budget B_j. These budgets Y_i impose a new condition to their associated task τ_i in order to accommodate for the minimum inter-arrival misbehaviour: task τ_i is allowed to pre-empt only if there is sufficient budget in Y_i. In this way the pre-emption delay that τ_i may cause in the schedule is tightly monitored.

The replenishment condition for server Y_i is defined as follows:

Server Y_i replenishment rule: At a time instant t', when task τ_i pre-empts some lower priority workload, a replenishment event is set for Y_i to occur at time instant $t' + T_i^Y$. The amount of budget replenished to server Y_i at the $t' + T_i^Y$ event is equal to the quantity reserved to pay the maximum pre-emption delay penalty associated to a single pre-emption.

This replenishment mechanism is in accordance with the sporadic server replenishment rules [SSL89] and hence the server Y_i is a sporadic server.

These two parameters Z_i and T_i^Y are set by the system designer and the question of how to define them will be discussed later. For now, bear in mind that these two parameters are given for each task $\tau_i \in \mathscr{T}$.

The purpose of each server Y_i is to ensure that new jobs of a task τ_i can only be released as long as the maximum pre-emption delay that τ_i can induce in the schedule (according to the schedulability test) is available in Y_i. To effectively implement this solution, the *budget augmentation* mechanism presented in the previous section is reformulated as a *budget transfer* mechanism. The main concept remains simple:

1. To release a new job (say at time t), a task τ_j is required to have *at least* $P_j^{\max}$ time units in its budget $Z_j(t)$. This quantity $P_j^{\max}$ is the maximum delay that τ_j can cause on the lower priority tasks by pre-empting them. It is straightforwardly defined as

   $$P_j^{\max} \stackrel{\text{def}}{=} \sum_{k=j+1}^{n} \delta_{j,k} \tag{5.18}$$

   If $Z_i(t) < P_j^{\max}$ then τ_j is not authorized to release a new job at time t and must wait until the earliest time instant $t' > t$ when $Z_j(t') \geqslant P_j^{\max}$.

2. Unlike the budget augmentation protocol proposed in the previous section, each time a task τ_i resumes its execution (say at time t) after being pre-empted (let $H(i)$ denote the set of tasks that pre-empted τ_i), τ_i does not experience its execution budget $B_i(t)$ being simply augmented by $\sum_{\tau_j \in H(i)} \delta_{j,i}$ time units, with $\sum_{\tau_j \in H(i)} \delta_{j,i}$ coming from thin air. Instead, $\delta_{j,i}(t)$ time units are *transferred* from the budget $Z_j(t)$ of each task $\tau_j \in H(i)$ to its execution budget $B_i(t)$.

It is important to stress that this framework is explained by considering two distinct servers to ease explanation and to allow for more flexibility in order to have different replenishment periods for the execution and pre-emption delay donation server. Nevertheless, if there is not sufficient budget in the pre-emption compensation server and there is sufficient budget on the execution server then the required budget on the execution server has to be transferred to the pre-emption delay server. This prevents the generation of unwanted blocking scenarios.

Informally speaking, the underlying concept behind this budget transfer protocol can be summarized as follows: "a task τ_i is allowed to pre-empt only if it can pay for the maximum damage that it may cause to *all* the tasks that it may pre-empt". If the task τ_i can pay the required amount of time units, i.e., τ_i has a provably sufficient amount of time units saved in its budget $Z_i(t)$, then it can release its new job and the pre-empted tasks will claim their due pre-emption delay compensation when they will eventually resume their execution.

This simple concept makes the framework safe. Rather than a formal proof, a set of arguments to substantiate the claim is provided. Suppose that a task τ_i starts misbehaving by frenetically releasing jobs that execute for an arbitrarily short time; hence clearly violating its minimum inter-arrival constraint.

1. From the point of view of a higher priority task (say, τ_j): each job of τ_j can pre-empt *at most* one job from τ_i and before releasing each of its jobs, τ_j makes sure that there is enough provision in its budget Y_j to compensate for the damage caused to the lower priority tasks, including τ_i.

2. From the point of view of the misbehaving task τ_i: this task will keep on generating jobs until its budget $Z_i(t)$ is depleted. For each job released, the framework ensures that the job can actually pay for the damage caused to the lower priority tasks. Regarding the higher priority tasks, each job of τ_i may be pre-empted and request some extra time units upon resumption of its execution. However, this extra budget requested has been accounted for when the higher priority jobs were allowed to be released – as mentioned in 1).

3. From the point of view of a lower priority task (say, τ_k): each job of τ_k may be pre-empted multiple times by the abnormal job release pattern of τ_i. However, upon each resumption of execution, τ_k will be compensated for the delay incurred by receiving some extra time units from the budget $Z_i(t)$ of the misbehaving task – as guaranteed in 2).

As seen, the sole purpose of each server Y_i, $\forall i \in [1,n]$ is to control the pre-emption delay that the task τ_i induces on the schedule. Since the upper-bound on the pre-emption delay related interference is now dictated by these servers, Schedulability Test 2 presented in the previous section can be rewritten as:

Schedulability Test 3. *A task set $\mathscr{T}$ is schedulable if, $\forall \tau_i \in \mathscr{T}$, $\exists t \in (0, D_i]$ such that*

$$C_i + \text{delay}(t) + \sum_{j=1}^{i-1} \left[\text{rbf}(S_j, t) + \text{rbf}(Y_j, t)\right] \leqslant t \tag{5.19}$$

Correctness of Schedulability Test 3. The replenishment mechanism of server Y_i is in accordance with the sporadic server replenishment rules. As a consequence of this fact and according to the Theorem 12 the maximum amount of budget consumed in any interval of length T_i^Y is Z_i. This means that $\text{rbf}(Y_j,t)$ is an upper-bound on the budget used for execution by any task that was pre-empted by task τ_j and got the due compensation in any interval of length t. By this reasoning and the correctness of schedulability tests 1 and 2 the correctness of this schedulability test is thus proven. □

The choice of the parameters of each Y_j server is left at the criteria of the system designer. However, in a given period T_i any task τ_i will require at least the execution of one job. As a consequence the budget Z_i of Y_i must necessarily be greater than or equal to $P_i^{\max}$. The simpler approach would be to define each server Y_i as $\langle Z_i = P_i^{\max}, T_i^Y = T_i\rangle$ as Y_i would have enough budget to compensate for all the interference that τ_i may cause in the schedule, assuming that the minimum inter-arrival constraint is not violated. However, the system designer may prefer $T_i^Y > T_i^S$ to provide more flexibility in case the task τ_i is expected to violate its minimum inter-arrival constraint, or even to accommodate intended bursty arrival of requests.

As a last note it is important to state the advantage of this framework with respect to a simple mechanism imposing a limitation on the number of jobs a task may release in a given time window. With this framework the number of jobs that a task may release without breaking the temporal-isolation guarantees is variable (and always greater than the worst-case that had to be considered if a static number of jobs had to be enforced) since this depends on the number of lower priority jobs that it has actually pre-empted so far. This allows for a more dynamic system with overall better responsiveness.

5.5 Limiting the Pre-emption Induced Budget Augmentation for Misbehaving Tasks

Suppose that a task τ_i misbehaves by violating its minimum inter-arrival constraint, i.e., it releases more than one job in a time interval $< T_i$, and one of its jobs gets pre-empted by a higher priority task, it is meaningful to consider whether task τ_i should get a pre-emption delay compensation when it will eventually resume its execution. In fact, it might be preferable not to transfer extra time units to its budget B_i until it returns to a state where it is respecting its contract parameters.

For this choice of design, Schedulability Test 3 becomes:

Schedulability Test 4. *A task set $\mathscr{T}$ is schedulable if, $\forall \tau_i \in \mathscr{T}$, $\exists t \in [0, D_i]$ such that*

$$C_i + \text{delay}(t) + \sum_{j=1}^{i-1} \text{rbf}(S_j,t) + \sum_{j=1}^{i-1} \min\left(\text{rbf}(Y_j,t), \sum_{\ell=1}^{q'_{j,i}(t)} \overset{\ell}{\max}(M_{j,i}(t))\right) \leqslant t$$

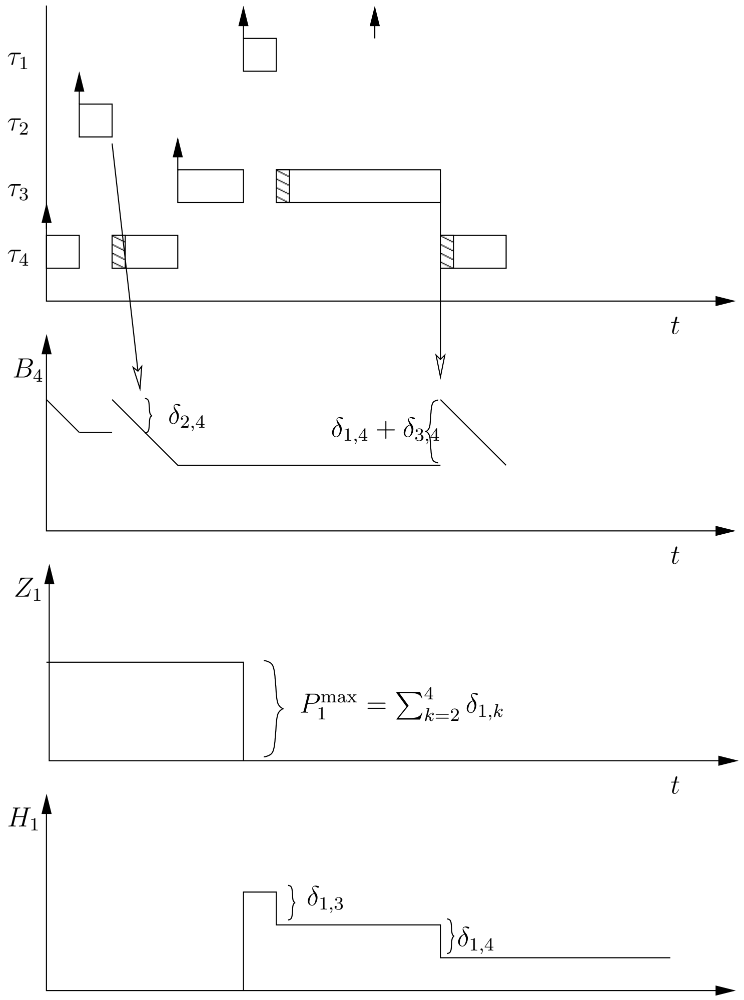


Figure 5.4: Budget Augmentation Example

where

$$q'_{j,i}(t) \overset{\text{def}}{=} \sum_{k=j}^{i-1} n_k(t)$$

since the tasks are not eligible for pre-emption delay compensation when they don not meet the minimum interarrival requirements, the maximum pre-emption delay related interference in the schedule is still upper-bounded by the analysis presented in Equation (5.5).

5.6 Implementation Issues

An efficient implementation is key for a useful approach to server-based scheduling. The general principle pursued is that the pre-emption delay repayment budget is only granted when an actual pre-emption occurs and is only transferred on resumption of execution.

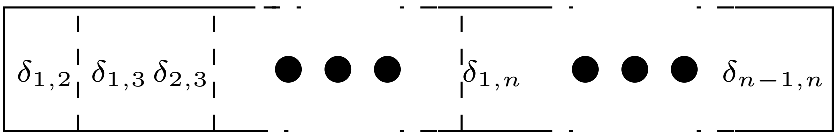


Figure 5.5: Pre-emption Delay Compensation Array

As a first step the pre-emption delay repayment budget values $\delta_{i,j}$ are arranged in a linear *pre-emption delay array* (Figure 5.5).

A second data structure employed to assist on the management of the temporal isolation framework is termed the pre-emption queue. This queue trivially maintains a depiction of the pre-emption order. When a task is first dispatched to execute upon the processor and hence pre-empts a previously running task then this task is inserted into the pre-emption queue. When a task terminates its execution it is removed from the queue.

As a third element, each task τ_i's control block has two bitfields ($out_{bf}(\tau_i)$ and $in_{bf}(\tau_i)$) of length n. The "in" bitfield contains the information about which tasks have to compensate τ_i and the "out" bitfield conserve the information about the tasks which have to compensate tasks which were pre-empted by τ_i. The mechanism utilizing this information is described in Algorithm 8. At the moment of job release from τ_i, the "out" bitfield is initialized with only the bit relative to τ_i set (i.e. $out_{bf}(\tau_i) = 0x1 << (i-1)$).

When a job τ_j terminates its execution at time t, it holds true that τ_j is the head of the pre-emption queue. At this time instant the task immediately preceding τ_j in the pre-emption queue (τ_i) has its "in" bitfield assigned in the following way: $in_{bf}(\tau_i) = in_{bf}(\tau_i) \vee out_{bf}(\tau_j)$. Scheduling decisions are only taken after this procedure terminates. After this operation it is not necessarily true that τ_i will become the head of the pre-emption queue and execute on the processor.

When a job from τ_i resumes execution, after a pre-emption, the scheduler examines the "in" bitfield $in_{bf}(\tau_i)$. The "in" bitfield holds the information on which tasks have pre-empted τ_i since the last time instant at which it was executing. The budget from server S_i is then accordingly augmented. If the considered framework is the one ensuring temporal-isolation with respect to minimum inter-arrival time misbehaving the corresponding pre-emption delay donations are deduced from the pre-emption delay donation servers of the tasks which have pre-empted τ_i.

After the budget of the task is duly augmented, the "out" bitfield is logically ORed with the "in" bitfield (i.e. $out_{bf}(\tau_i) = in_{bf}(\tau_i) \vee out_{bf}(\tau_j)$. Immediately after this step the "in" bitfield is reset ($in_{bf}(\tau_i) = \mathtt{0x0}$). These three procedures need to be carried out atomically.

Let us use a different schedule example to better visualize the bitfield related operations. In Figure 5.6 an example of the bit-field evolution as a response to a given set of events is presented. There are 9 events present. When a task has no active job in the system its bit

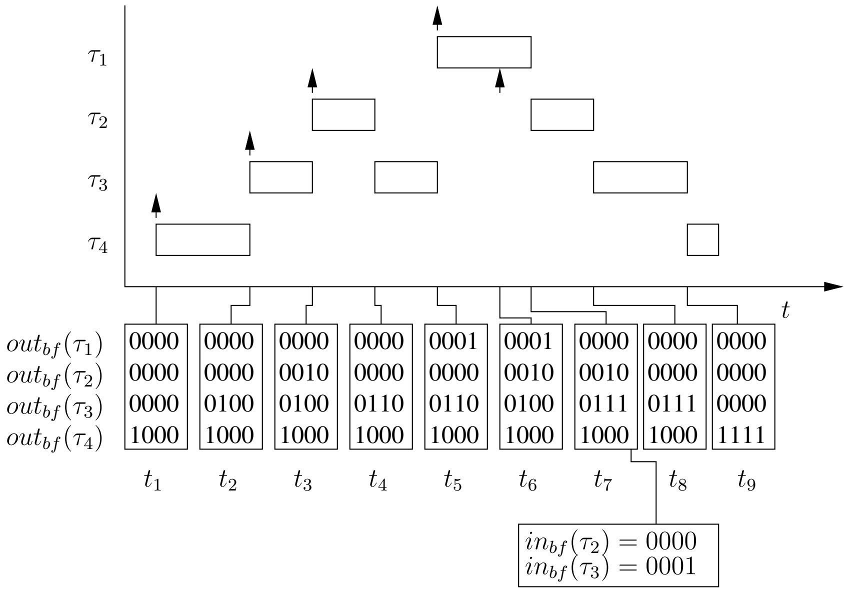


Figure 5.6: Bit Field Snapshots of Relevance

field is by default 0. At events 1–3, tasks τ_1 to τ_3 release jobs, consequently their bit fields are set to their respective index. At time t_4, the first job of task τ_2 finishes its execution. At this time instant, and since τ_2 is the prior pre-emption queue head and τ_3 is the task after the queue head, the bit field from task the $in_{bf}(\tau_3) = in_{bf}(\tau_3) \vee out_{bf}(\tau_2)$. After this operation is performed $in_{bf}(\tau_2) = 0$. At t_5 and t_6 jobs from τ_1 and τ_2 are released correspondingly. At each event the corresponding bitfield is set accordingly. The job from task τ_1 terminates at t_7. At this time instant, the “out” bitfield of task τ_1 is passed onto the subsequent task in the pre-emption queue (τ_3). Hence $in_{bf}(\tau_3) = in_{bf}(\tau_3) \vee out_{bf}(\tau_1)$ Since τ_2 was not in the ready queue the “out” bitfield from τ_1 does not get passed to its “in” bitfield.

The algorithm describing the bitfield inspection and budget augmentation is presented in Algorithm 8. Notice that in the scenario where minimum inter-arrival times cannot be relied upon, after the budget is augmented (line 6), the same value has to be decreased from the Y_{index} server of the pre-empting task. In the same framework it is then necessary to set up a replenishment event for server Y_{index}, T_{index} units after the execution resumption (line 7). At this replenishment event Z_{index} is set to be replenished by $\delta[\text{array\_index}]$ units. The variable offset$_i$ is an offset with respect to the base of the δ array, where the first element pre-emption delay compensation value ($\delta_{1,i}$) from task τ_i is stored (Figure 5.5).

5.7 Example of Framework Usage with CRPD

The previously described theory may be exploited in order to ensure temporal-isolation with respect to any subsystem holding state concurrently accessed by tasks in the system. In this

Algorithm 8: Pre-emption Delay Augmentation Algorithm

```
$\tau_i$ execution resumption after pre-emption: $index = 0$;
while index $< i$ do
    if $in_{bf}(\tau_i)\&$ 0x1 then
        array_index = offset$_i$ + index;
        $B_i += \delta$[array_index];
        $Y_{index} -= \delta$[array_index];
        Set_replenishment_event($\tau_{\text{index}}, \delta$[array_index])
    $in_{bf}(\tau_i) = in_{bf}(\tau_i) >> 1$;
    $index++$;
$in_{bf}(\tau_i) = 0x0$
```

section a description on how temporal-isolation is ensured, when caches are present in the execution platform by usage of the presented framework, is provided.

At each program point a task has a set of useful memory blocks (UCB) in cache. These memory blocks were loaded into the cache at some prior program point and will be reused in the future. Each program point p of task τ_k has an associated UCB set denoted as UCB_k^p [AB09]. Additionally, the set of cache lines accessed during the execution of task τ_j (ECB_j) [AB09] may be constructed.

The maximum CRPD which a pre-emption by task τ_j may induce in task τ_k is then defined as [AB09]:

$$\delta_{j,k} \overset{\text{def}}{=} BRT \times \max_p \left\{ \left| \text{UCB}_k^p \bigcap \text{ECB}_j \right| \right\} \tag{5.20}$$

BRT is a constant, denoting the worst case latency for a cache miss to be served.

5.7.1 Temporal-isolation Assumptions

In order to ensure full temporal-isolation with the used pre-emption delay estimation mechanisms, when using information from the pre-empting tasks to compute the $\delta_{j,i}$, it is mandatory to ensure that the ECB sets of each pre-empting task are met at run-time, as it is assumed that a task can execute for longer than expected, which implies that it might in turn also use a larger cache footprint.

Both instruction and data cache footprints have to be relied upon. In order to ensure the worst-case cache footprint, it suffices to ensure that each task only accesses the basic blocks assumed in the static analysis. This can be easily achieved with the work by Martín Abadi et al. [ABEL09], where each jump instruction is guarded against jumps into illegal memory locations. Since at run-time, all the possible basic blocks accessed are the exact same as the ones assumed in the static analysis the instruction cache ECB set is never violated. When using [ABEL09], all the jump targets in the code are statically defined, i.e. jump to function pointers in the code are not allowed.

In the case of data caches, since the basic blocks accessed at run-time are enforced, it is sufficient to ensure that no dynamic memory is used, which is generally the case in real-time workload. Another assumption is that any access to an array is guarded against accesses outside of the array bounds. As a last assumption, recursive calls are forbidden in order to ensure that the stack footprint is known offline.

If the ECB set of any task cannot be relied upon, it is always safe to consider that the maximum pre-emption delay induced in task τ_k when pre-empted by any task τ_j is the time to reload the entire UCB_k set:

$$\forall j : \delta_{j,k} = BRT \times |\mathrm{UCB}_k| \,. \tag{5.21}$$

5.7.2 Experimental Results

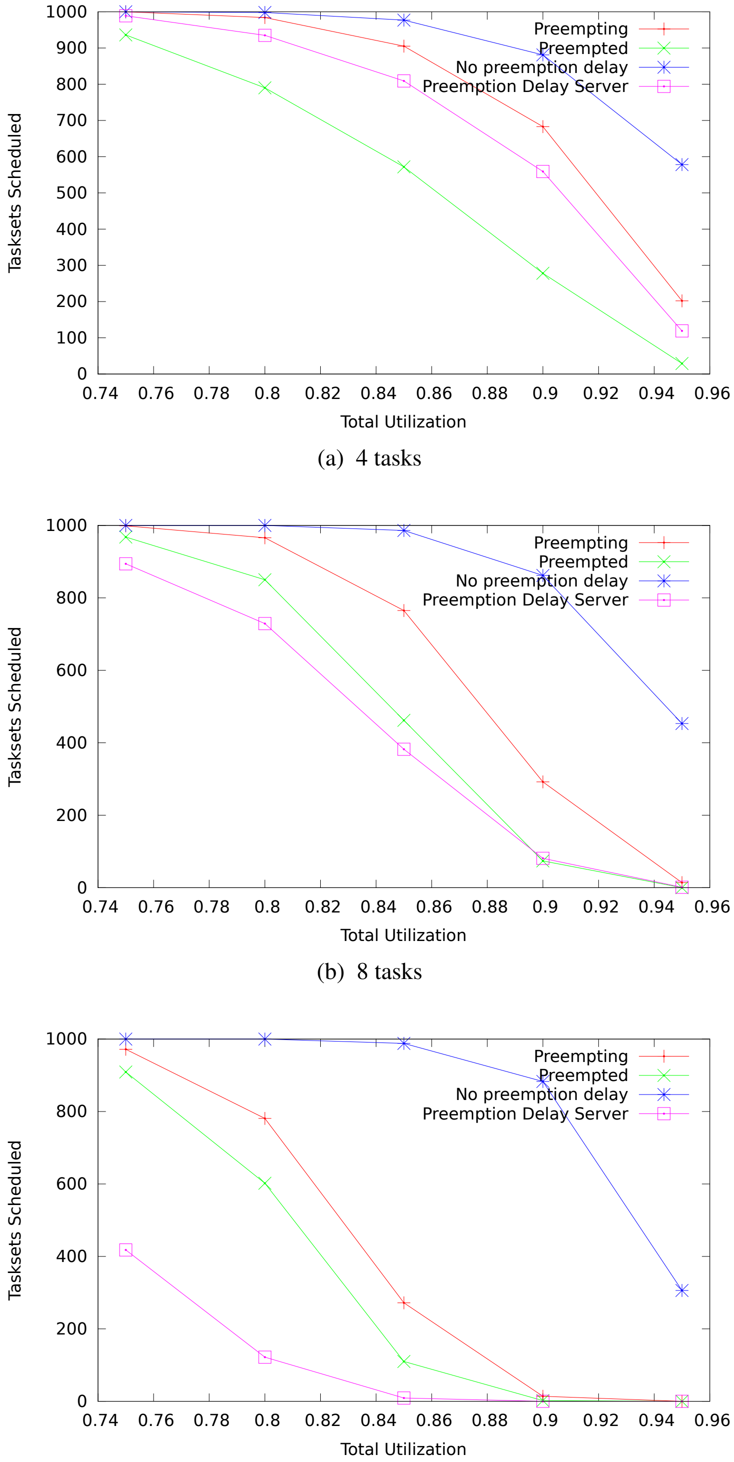
In order to assess the validity of the contribution a set of experiments was conducted. The schedulability guarantees from the proposed framework is trialled against the scenario where the maximum pre-emption delay is integrated into the budget of the execution server. Results for the same task sets are also displayed for a schedulability test oblivious of pre-emption delay. As in previous chapters, all tasks are generated using the unbiased task set generator method presented by Bini (UUniFast) [BB04a].

Task systems are generated for every utilization step in the set $\{0.75, 0.8, 0.85, 0.9, 0.95\}$ in a random fashion, their maximum execution requirements (C_i) were uniformly distributed in the interval $[20, 400]$. Knowing C_i and the task utilization U_i, T_i is obtained. At each utilization step 1000 task sets are trialled and checked whether the respective algorithm considers it schedulable. Task set sizes of 4, 8, and 16 tasks have been explored. The relative deadline of tasks is equal to the minimum inter-arrival time ($D_i = T_i$).

When each task is randomly generated pre-emption delay cost is obtained for each task pair in the task-set. The cache considered is composed of 10 cache lines. For each task a set of useful cache lines is computed, the usage of each cache line follows a uniform distribution. Similarly for each task a set of cache lines which are accessed during its execution is computed. The cardinality of interception of the useful set of τ_i with the accessed set of τ_j upper-bounds the maximum number of cache lines that τ_i has to re-fetch has a result of a pre-emption by τ_j.

The servers S_i have been attributed parameters $B_i = C_i$ and $T_i^S = T_i$. For the situation where the pre-emption delay server Y_j is put to use, its parameters are $Z_j = \sum_{j<k} \delta_{j,k}$ and $T_j^Y = T_j$.

The results are depicted in Figures 5.7a to 5.7c. In the plots the scenario where the pre-emption delay is incorporated into the task execution budget is displayed is presented



(a) 4 tasks

(b) 8 tasks

(c) 16 tasks

Figure 5.7: Schedulability Comparison With CRPD

by the green line with “x” points. The presented framework for well-behaving minimum inter-arrival times is represented by the red line with “+” points. The purple line with square points represents the framework performance for situations where the minimum inter-arrival times cannot be trusted. Finally the blue line with “star” points displays the results for fixed task priority schedulability test disregarding pre-emption delay.

From the displayed results it is apparent that the schedulability achieved with the proposed framework is generally significantly higher than the one enabled by the simpler version considering the pre-emption delay as part of the execution budget. When the minimum inter-arrival times cannot be relied upon the schedulability degrades. It is important to note that the proposed framework ensures temporal-isolation and there exists no other solution apart from this one which ensures the temporal-isolation property for the given system model. Furthermore, when the number of tasks is small, the framework which provides the stronger guarantees appears to have on average a higher scheduling performance. The schedulability reduction attached to the framework for misbehaving tasks with respect to the minimum inter-arrival time is the price to pay for the added guarantees.

5.8 Temporal-isolation Framework Considerations

Reservation-based systems are one fundamental way to enforce temporal-isolation in safety critical real-time systems. It is shown in this chapter that, when pre-emption delay is present in the system, the prior state of the art temporal-isolation mechanisms induce pessimism in the analysis and in the budget allocation procedures. This inherent limitation is one of the fundamental motivators for the presented run-time budget augmentation mechanism. This framework enables a provably reduction on the budget over-provisioning in platforms with non-negligible pre-emption delay overheads. For a more realistic model in which the minimum inter-arrival time of tasks cannot be relied upon, there existed no prior result in the literature ensuring temporal-isolation (for systems where pre-emption delay is non-negligible). A second framework which relies on budget transfers between pre-empting and pre-empted tasks effectively enforces the temporal-isolation property in such a considerably challenging system model (where execution requirements and minimum inter-arrival times cannot be trusted upon).

Chapter 6

Multi-processor Limited Pre-emptive Theory

As multicores are currently a mainstream computing apparatus, and will remain so in the foreseeable future, some COTS platforms are being increasingly adopted as target platforms in the embedded domain. In time it is expected that these will be used in the high criticality embedded world. It is thus important to study the scheduling theory that would govern the operation of such systems.

Current real-time operating systems provide supports symmetric multiprocessor scheduling. A number of global scheduling policy types exist. These can be categorized into distinct classes with respect to where tasks and their constituent jobs are allowed to execute [DB11]. One extreme is the fully partitioned scheduling. Tasks are statically allocated to one processor, its workload can then only execute on the same single processor. On the other end of the spectrum lies the global scheduling where there is no *a-priori* restriction on which processor the workload of any task will execute. Both fully partitioned and global fixed task priority scheduling policies are provided out of the box in popular real-time operating systems [Win].

A shortcoming of the literature on the schedulability assessment for global multicore scheduling is that the workload is assumed not to incur additional overheads when pre-emptions and migrations happen. The cost of pre-emptions and especially migrations between cores that do not share cache have been shown to be quite significant [BBA10]. As in fully pre-emptive disciplines for single core it is difficult to quantify the number of pre-emptions a given task is subject to. Moreover in multicore when a pre-emption occurs a subsequent migration may take place. It is then beneficial to quantify these events and, if possible, avoid these effects when these prove unnecessary for correct system temporal behaviour. With this intent in mind the theory of limited pre-emptive scheduling is devised for symmetric multicores. The model presented allows for the accurate definition of a limited set of points where a given task may be pre-empted, hence allowing for reduced pessimism when analysing the effect of pre-emptions and migrations. The work presented in this Chapter largely follows what has already been presented in [MNP+13] and [DBM+13] but extends it as well in what respects to pre-emption delay integration into the analysis and on schedulability assessment of the ADS policy.

6.1 Global Fixed Task Priority Response Time Analysis

In this chapter the limited pre-emptive global scheduling theory is defined. The schedulability test is presented with three approaches to estimate the blocking from lower or equal priority non-pre-emptive regions. The new scheduling policy is shown to ensure that a job can be blocked by lower priority workload only before its first dispatch. This compares favourably with the scheduling policy termed RDS which is subject to multiple instances of blocking throughout the execution of a given job. The presented scheduling policy (ADS) dominates global fully pre-emptive fixed task priority and fully non-pre-emptive scheduling with respect to schedulability. As final contributions the blocking estimation mechanisms are compared against each other. Finally, the ADS policy is shown to drastically reduce the number of pre-emptions occurring in the schedule when compared to global fully pre-emptive scheduling.

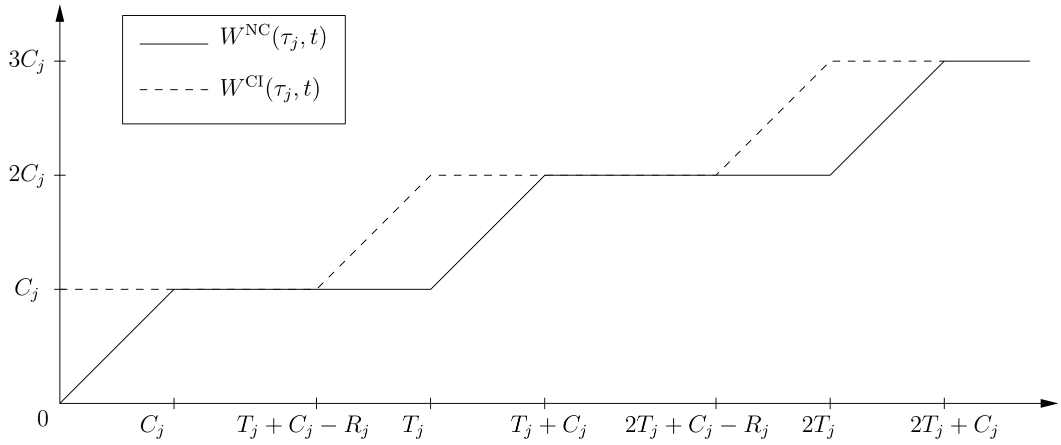


Figure 6.1: Functions $W^{\mathrm{NC}}(\tau_j,t)$ and $W^{\mathrm{CI}}(\tau_j,t)$ depiction for a given task τ_j

In global fully pre-emptive fixed priority scheduling, if a task τ_j has no pending workload at the beginning of an interval, an upper bound on the workload which it may execute in an interval of length t is given by [GSYY09]:

$$W_j^{\mathrm{NC}}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \times C_j + \min(t \mod T_j, C_j) \tag{6.1}$$

In (6.1) a release of a job from task τ_j is assumed to occur at time instant 0 and subsequent releases occur at the minimum inter-arrival time, in fact leading to a worst-case amount of workload released in an interval of length t when no carry-in is present.

Similarly, if the same task τ_j might have some pending workload released before the beginning of the interval, a conservative upper-bound on the workload it may execute in an interval of length t is given by [GSYY09]:

$$W_j^{\mathrm{CI}}(t) = \left\lfloor \frac{\max(t-C_j,0)}{T_j} \right\rfloor \times C_j + C_j + \min\left([[t-C_j]_0 \mod T_j - (T_j - R_j)]_0, C_j\right) \quad (6.2)$$

In Equation (6.2) a job of task τ_j is assumed to be released before the start of the interval and that it will execute the bulk of the workload after the interval starts. The upper-bound present in Equation (6.2) assumes that the carry-in job from task τ_j was subject, since its release at time instant $-R_j^{UB} - C_j$ until the beginning of the interval, to the worst-case scenario such that this job has not executed any workload yet. The execution of the workload will terminate R_j time units after the job was released. Subsequent jobs are released with the minimum inter-arrival separation. The second job is then released at time instant $T_j - R_j + C_j$. This constitutes a conservative estimation of the workload a task τ_j executes in a given time interval of length t if there exists some pending workload released at some point before the beginning of the interval.

For the same task τ_j Equation (6.2) is an upper-bound on Equation (6.1), this can be observed from the graphical representation of both functions presented in Figure 6.1. The difference between the upper-bound considering a carry-in scenario and one where no carry-in is considered yields the following non-negative quantity:

$$W_j^{\mathrm{diff}}(t) = W_j^{\mathrm{CI}}(t) - W_j^{\mathrm{NC}}(t) \quad (6.3)$$

In [GSYY09] it is shown that a conservative upper-bound on the amount of higher priority workload which will execute during the response time of a job from task τ_i is constructed by assuming that some $m-1$ higher priority tasks have pending workload at the time of τ_i's job release and that a synchronous release of jobs from the remaining higher priority tasks occurs at the same instant as τ_i's job release. Writing this upper-bound in more formal terms yields:

$$\Omega_i(t) \overset{\mathrm{def}}{=} \left(\sum_{l=1}^{m-1} \max^{\ell}_{\tau_j \in \mathrm{hp(i)}} W_j^{\mathrm{diff}}(t) \right) + \sum_{\tau_j \in \mathrm{hp(i)}} W_j^{\mathrm{NC}}(t) \quad (6.4)$$

Where $\max^{\ell}_{\tau_h \in \tau_1, \cdots, \tau_{i-1}}$ returns the ℓ^{th} greatest function value along the higher priority task's workload dimension. In a situation where no higher priority task has carry-in workload at the beginning of the time interval, a sum over all the $W^{\mathrm{NC}}(\tau_j, t)$ functions of tasks of higher priority than τ_i would yield an upper-bound on the higher priority workload that would execute in the time interval of length t. Since it is known that in the worst-case some $m-1$ tasks present carry-in workload at the beginning of the interval, then this additional workload will never exceed the difference between the maximum workload in a no carry-in situation and the carry-in situation. By choosing the $m-1$ higher priority tasks which for a given interval length display the biggest difference between the no carry-in and carry-in situation and by summing these differences over all functions of no carry-in an upper-bound on the workload which higher priority tasks may execute in the interval of length t is obtained.

In [Bak03] Baker showed that in a multi-core platform composed of m identical cores, a unit of higher priority workload can interfere with the execution of task τ_i by at most $\frac{1}{m}$ time units. Consider that some task τ_j is executing on one processor during one time unit. By executing on this processor it will not prevent τ_i from getting hold of some other processing entity as there exist $m-1$ others in the platform. In order for τ_i to be prevented from executing, the m processors need to be busy. In order for m cores to be busy for one time unit then m time units of higher priority workload need to execute on the overall platform. Combining (6.4) and this fact enables the statement of the upper-bound on the overall interference a task τ_i is subject to in a time interval of length t. This is written as:

$$I_i(t) = \frac{\Omega_i(t)}{m} \tag{6.5}$$

When considering the fully pre-emptive global fixed task priority scheduling policy, by exploiting the upper-bound on the interference by higher priority workload, the following sufficient schedulability test is devised:

$$\forall \tau_i \in \mathscr{T}, \exists t \in [0, D_i] : I_i(t) + C_i \leqslant t \tag{6.6}$$

A task-set is said to be schedulable if for all task $\tau_i \in \mathscr{T}$ the condition in (6.6) holds. The schedulability test presented in (6.6) is a sufficient test. This means that some task-sets may indeed manage to meet all the deadlines even if for some tasks the condition in (6.6) is not met. Nevertheless if the condition is met for all the tasks in the taskset then the timing requirements of all the tasks are guaranteed to be met at run-time.

Similarly to the single processor case this condition need not be tested for all of the values in the continuous interval $[0, D_i]$ but rather for a finite number of time instants. Both functions $W_j^{\mathrm{NC}}(t)$ and $W_j^{\mathrm{CI}}(t)$ are piecewise linear functions, i.e. they may be defined as a set of linear functions in distinct time intervals. As a consequence of this, the $\Omega_i(t)$ function itself may be defined as a set of linear functions in distinct time intervals as well. Hence the relation $t - I_i(t)$ is maximized in the $[0, D_i]$ interval for some value $t \in \Gamma_i$. The set Γ_i encloses the Γ_i' set of points where the first derivative of the function $\Omega_i(t)$ changes in value and the time instant of the deadline D_i of τ_i. The $\Gamma_i' \cup \{D_i\}$ set of points contains the points of interest when trying to maximize the value of $t - I_i(t)$.

Let us now focus on the derivation of the set Γ_i containing the points where the condition has to be tested.

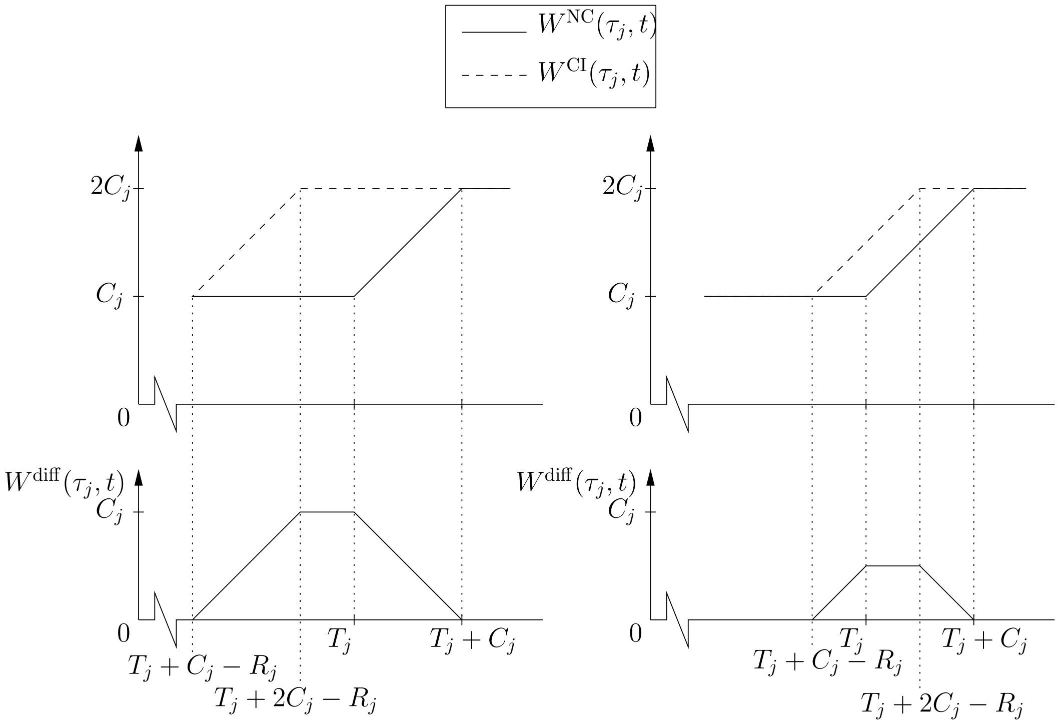


Figure 6.2: Functions $W^{\text{diff}}(\tau_j,t)$ depiction for two distinct relations between C_j and R_j

The difference between $\Omega_i(t)$ and the supply line is maximized at some points where the $\Omega_i(t)$ first derivative changes value. The $\Omega_i(t)$ function has increases in its first derivative at specific points which are related to scenarios here described.

Since the $\Omega_i(t)$ function is a sum of two functions for each higher priority task, the changes in the $\Omega_i(t)$ function first derivative can only occur at any t where either the first derivative of $W^{\text{diff}}(\tau_h,t)$ or $W^{\text{NC}}(\tau_h,t)$ changes or if both change.

1. first derivative increase of $W^{\text{NC}}(\tau_j,t)$:
   occurs at every $k \times T_j$ (this can be observed in Figure 6.1)

$$\Gamma_i^1 \overset{\text{def}}{=} \bigcup_{j\in\{1,\cdots,i-1\}} \left\{ k \times T_j | k \in \mathbb{N}, 0 \leqslant k \leqslant \frac{t}{T_j} \right\}$$

2. first derivative increase of $W^{\text{diff}}(\tau_j,t)$:
   occurs at every $k \times T_j + C_j - R_j$ (this can be observed in figures 6.1 and 6.2)

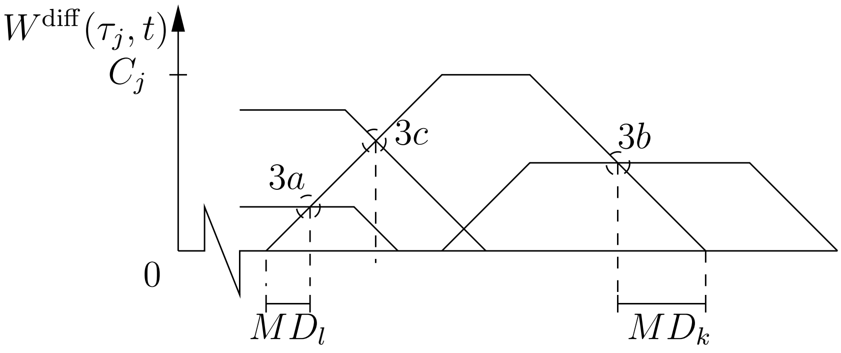


Figure 6.3: Intersection Points Between the $W^{\text{diff}}(\tau_j,t)$ Functions of Distinct Higher Priority Tasks

$$\Gamma_i^2 \stackrel{\text{def}}{=} \bigcup_{j\in\{1,\cdots,i-1\}} \left\{ k\times T_j + C_j - R_j | k \in \mathbb{N}, 0 \leqslant k \leqslant \frac{t+R_j-C_j}{T_j} \right\} \tag{6.7}$$

3. When there is a change in the set of $m-1$ $W^{\text{diff}}(\tau_j,t)$ functions with greatest value. Three distinct situations have to be considered. These distinct typology of occurrences can be observed in Figure 6.3 where the three situations are marked accordingly:

   (a) When "upward segment" of one $W^{\text{diff}}(\tau_j,t)$ "intersects a plateau" of another $W^{\text{diff}}(\tau_k,t)$ function

   (b) When "downward segment" of one $W^{\text{diff}}(\tau_j,t)$ "intersects a plateau" of another $W^{\text{diff}}(\tau_k,t)$ function

   (c) When "upward segment" of one $W^{\text{diff}}(\tau_j,t)$ "intersects" with a "downward segment" of another $W^{\text{diff}}(\tau_k,t)$ function

Let us define the following variable $MD_j = \min\{C_j, R_j - C_j\}$ which represents the maximum value of the function $W^{\text{diff}}(\tau_j,t)$ for any time instant t. The upward segments of $W^{\text{diff}}(\tau_j,t)$ start at every $k\times T_j + C_j - R_j$ and terminate at $k\times T_j + C_j - R_j + MD_j$. Whereas a downward segment starts at every $k\times T_j + C_j - MD_j$ and terminate at $k\times T_j + C_j$.

In a time window of length t, a task τ_j has at most $K_j = \left\lfloor \frac{t+R_j-C_j}{T_j} \right\rfloor$ upward segments of function $W^{\text{diff}}(\tau_j,t)$ since for each $k \leqslant \frac{t-C_j+R_j}{T_j}$ one upward segment starts inside the time interval of length t. For the case 3a it suffices to check for each $k \leqslant K_j$, $k\times T_j + C_j - R_j + \max(MD_h, MD_j)$ and for each remainder higher priority task τ_h. Only MD_h units after its beginning can an upward segment of task τ_j intersect a plateau of another task τ_h. In case the maximum value of $W^{\text{diff}}(\tau_h,t)$ is greater than the maximum value of $W^{\text{diff}}(\tau_j,t)$ then an upward segment from τ_k will never intersect the plateau of τ_h.

$$\Gamma_i^{3a} \overset{\text{def}}{=} \bigcup_{j\in\{1,\cdots,i-1\}} \bigcup_{h\in\{1,\cdots,i-1\}\setminus j} \left\{ k\times T_j + C_j - R_j + \max(MD_h, MD_j) |\, k\in\mathbb{N}, 0\leqslant k\leqslant K_j \right\} \tag{6.8}$$

In a time window of length t, a task τ_j has at most $L_j = \left\lfloor \frac{t-C_j+MD_j}{T_j} \right\rfloor$ downward segments of function $W^{\text{diff}}(\tau_l, t)$ since for any $l \leqslant \frac{t-C_j+MD_j}{T_j}$ a downward segment of τ_j starts in a time interval of length t. For the case 3b it is enough to check the condition for all $l \leqslant L_j$, $l\times T_j - C_j - \max(MD_h, MD_j)$ and for each remainder higher priority task τ_h. If a downward segment from task τ_j intersects a plateau from task τ_h then the intersection occurs at $l\times T_j + C_j - MD_h$, i.e. MD_h time units before the downward segment from τ_j ends. If the plateau of τ_h is greater than the plateau from τ_j this intersection never occurs.

$$\Gamma_i^{3b} \overset{\text{def}}{=} \bigcup_{j\in\{1,\cdots,i-1\}} \bigcup_{h\in\{1,\cdots,i-1\}\setminus j} \left\{ l\times T_j + C_j - \max(MD_h, MD_j) | l\in\mathbb{N}, 0\leqslant l\leqslant L_j \right\} \tag{6.9}$$

The third case 3c proves slightly more complex. For each of the K_j downward segments from τ_j, the possible upward segments of task τ_h with which it intersects need to be found. An intersection between an upward segment from τ_h and a downward segment from τ_j occurs at midpoint between the start of the upward segment and the end of the downward segment. Again, a downward segment from τ_j terminates at $k\times T_j + C_j$. This downward segment can only intersect with the upward segment from τ_h which started before time instant $k\times T_j + C_j$. The later upward segment from τ_h then starts at time instant

$$\left\lfloor \frac{k\times T_j + C_j - C_h + R_h}{T_h} \right\rfloor \times T_h + C_h - R_h \tag{6.10}$$

The set Γ_i^{3c} is then defined as:

$$\Gamma_i^{3c} \overset{\text{def}}{=} \bigcup_{j\in\{1,\cdots,i-1\}} \bigcup_{j\in\{1,\cdots,i-1\}\setminus j} \left\{ \frac{\left\lfloor \frac{k\times T_j + C_j - C_h + R_h}{T_h} \right\rfloor \times T_h + C_h - R_h + k\times T_j + C_j}{2} | k\in\mathbb{N}, 0\leqslant k\leqslant K_j \right\} \tag{6.11}$$

The set Γ_i is finally constructed by applying the union set operation over all the previously described sets:

$$\Gamma_i \overset{\text{def}}{=} \Gamma_i^1 \cup \Gamma_i^2 \cup \Gamma_i^{3a} \cup \Gamma_i^{3b} \cup \Gamma_i^{3c} \cup \{D_i\} \tag{6.12}$$

When the schedulability condition is tested over a discrete set of points the response time computation can still be efficiently carried out:

$$t_1 = \min\{t \in \Gamma_i | I_i(t) + C_i \leqslant t\} \tag{6.13}$$

$$t_2 = \max\{t \in \Gamma_i | t < t_1\} \tag{6.14}$$

$$R_i^{UB} = t_1 + \frac{I_i(t_1) + C_i - t_1}{1 - \frac{I_i(t_2) - I_i(t_1)}{t_2 - t_1}} \tag{6.15}$$

The quantity $R_i^{UB} \in (t_2, t_1]$ is the intersection between the line segment

$$I(t) = \frac{I(t_1) - I(t2)}{t_1 - t_2} \times t + I(t_1) - \frac{I(t_1) - I(t2)}{t_1 - t_2} \times t_1 \tag{6.16}$$

defined $\forall t \in (t_2, t_1]$ and the supply line $f(t) = t$.

The formulation of the sufficient schedulability condition presented in this work is equivalent to the one in [GSYY09].

6.2 GFTP Limited Pre-emptive Scheduling Policies

In a multi-core platform global fixed task priority fully pre-emptive scheduling discipline may be informally described as a policy where at any time t the m highest priority tasks with available workload execute on the m processors comprising the platform.

When tasks are composed of both pre-emptible and non-pre-emptible workload then more complex protocols may be devised reflecting the more complex nature of the workload. In this work two scheduling policies are considered:

1. *Regular Deferred Scheduling* (RDS): At any point in time, the pre-emptible jobs executing on any processor are eligible for being pre-empted by a higher priority job;

2. *Adapted Deferred Scheduling* (ADS): At any point in time a pre-emption can only occur if the lowest priority running job is pre-emptible, in which case the lowest priority running job is pre-empted from the processor on which it runs and the highest priority waiting job is dispatched onto the same processor.

Bear in mind that these are only two generalisations of the regular fully pre-emptive scheduling discipline. The RDS policy is the straightforward derivation of the fully pre-emptive scheduler whereas the ADS is an adaptation which enables some interesting properties with respect to lower priority interference to be achieved.

In the RDS policy, a higher priority job from τ_i might suffer interference from lower priority non-pre-emptive regions more than once after τ_i has commenced execution. A situation where said priority inversion occurs after the start of τ_i execution may be observed

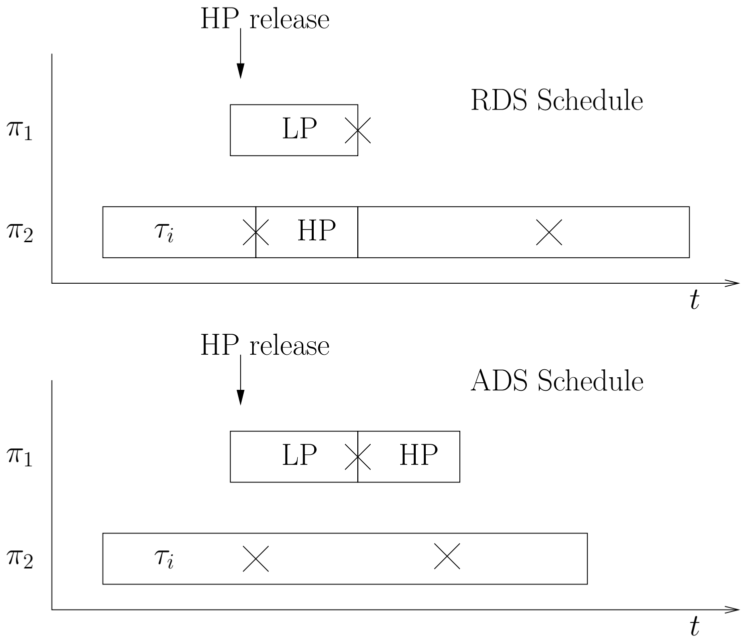


Figure 6.4: Possible Priority Inversion After a Job From τ_i Commences Execution in RDS

in the top schedule displayed in Figure 6.4. In Figure 6.4 the crosses represent fixed pre-emption points. The bottom schedule in the same picture displays the ADS schedule for the specific workload pattern. In this case, before τ_i is pre-empted, all other lower priority workload has to be pre-empted first from the platform. A task can only be pre-empted if it is the task currently running upon some processor such that it has the lowest priority among all tasks currently executing in the system.

Once a task starts to execute its last non-pre-emptive region it ceases to suffer interference, as a consequence it is only subject to interference during the execution of the first $C_i - Q_i$ units of workload. The schedulability test then becomes:

$$\exists t \in [0, D_i - Q_i] : t - \frac{1}{m} \times \left(WA_i^{diff}(t) + \sum_{\tau_j \in \text{hp(i)}} W_j^{NC}(t) + A_i^{NC}(t) \right) - (C_i - Q_i) > 0 \tag{6.17}$$

The corresponding upper-bound on the response time of a given task τ_i can thus be computed as:

$$R_i^{UB} = \min \left\{ t | t - \frac{1}{m} \times \left(WA_i^{diff}(t) + \sum_{\tau_j \in \text{hp(i)}} W_j^{NC}(t) + A_i^{NC}(t) \right) - (C_i - Q_i) > 0 \right\} + Q_i \tag{6.18}$$

Similarly to the fully pre-emptive scenario, the term $\sum_{\tau_j \in \mathrm{hp(i)}} W_j^{NC}(t)$ upper-bounds the maximum interference that higher priority workload may induce on τ_i when no higher priority carry-in exists. The $A_i^{NC}(t)$ function characterizes the maximum amount of interference due to lower priority workload released inside the interval of interest, which may exhibit non-pre-emptive regions and hence prevent higher priority workload from executing on the processors. The function $WA_i^{diff}(t)$ encapsulates the maximum interference contribution from carry-in workload. This is workload which was released before or immediately before the beginning of the interval of interest and which will be executed inside the interval of interest.

Similarly to the fully pre-emptive GFTP sufficient test [GSYY09], where only $m-1$ higher priority tasks with carry-in workload are considered, in the limited pre-emptive scheduling policies at most $m-1$ higher priority tasks are considered to have carry-in (although the worst-case response time of higher priority tasks may be higher in the limited pre-emptive scenario).

In the case of non-pre-emptive regions there might exist at most m lower priority tasks which were executing immediately before the start of the interval of interest. If some core is executing lower priority workload then this implies that this core cannot be executing higher priority carry-in in the beginning of the interval of interest. As a consequence then, if there are k lower priority tasks executing non-pre-emptively in the beginning of the interval of interest, then there can be at most $m-k$ higher priority tasks with carry-in. The maximum additional workload due to the k lower or equal priority tasks executing in the beginning of the interval of interest is denoted by A_i^k. The computation of an upper-bound of A_i^k given a set of lower or equal priority non-pre-emptive regions is the subject of the subsequent sections.

The $WA_i^{diff}(t)$ function is defined as follows:

$$WA_i^{diff}(t) \overset{\text{def}}{=} \max_{k \in \{1,\cdots,m\}} \left\{ A_i^k + \sum_{l=1}^{m-k} \overset{\ell}{\max_{\tau_h \in \tau_1,\cdots,\tau_{i-1}}} W^{\text{diff}}(\tau_h, t) \right\} \tag{6.19}$$

6.3 RDS Lower Priority Interference

The RDS policy's sufficient schedulability test was previously published in [DBM+13]. In the RDS policy the function $A_i^{NC}(t)$ can be constructed by considering that each non-pre-emptive region of any task with priority lower than τ_i may indeed interfere with τ_i execution. A lower priority non-pre-emptive region, from task τ_ℓ, may commence immediately before the release of a higher priority task τ_j. If task τ_i happens to be executing pre-emptively or if it happens that the release of τ_j coincides with a pre-emption point, τ_i is pre-empted by τ_j. In this case τ_i is indeed being interfered by τ_ℓ which is of lower priority. This situation can occur for any non-pre-emptive region of a lower priority task. Hence an upper-bound on the interference by lower priority workload can be written by resorting to Equation (6.20) and Equation (6.21).

$$A_i^{NC}(t) = \\ \sum_{j=i+1}^{n} \left\lfloor \frac{\max(t - C_j, 0)}{T_j} \right\rfloor \times Cv_j + \min\left([[t - Cv_j]_0 \mod T_j - (T_j - R_j)]_0, Cv_j\right) \tag{6.20}$$

$$A_i^k(t) = \sum_{j=i}^{n} Cv_j \tag{6.21}$$

In equations (6.20) and (6.21) the the quantity Cv_j is equal to the maximum amount of time that jobs from task τ_j execute non-pre-emptively.

Equation (6.21) models the non-pre-emptive workload that might be released prior to the interval of interest and execute inside it. In turn the Equation (6.20) models the non-pre-emptive lower priority workload which can be released inside the interval of interest and execute inside the same interval.

In order to evaluate the schedulability increase obtained with RDS only a single last non-pre-emptive region is considered per task. Considering multiple last non-pre-emptive regions would virtually not aid on increasing the schedulability since only the last region manages to postpone effectively interfering workload and having more than one non-pre-emptive region would lead to a heavier share of lower priority workload being considered as interfering with the higher priority tasks.

6.4 ADS Lower Priority Interference

In the ADS policy $A_i^{NC}(t) = 0$. Lower priority tasks may execute on other processors while τ_i is executing, but higher priority workload will only be able to pre-empt τ_i when τ_i is the lowest priority task executing on any processor. As a consequence of this, lower priority workload can only interfere with τ_i execution if they are currently executing once τ_i is released. This is the key difference between ADS and regular fixed priority scheduling with deferred pre-emption (RDS).

Contrary to the single processor limited pre-emptive theory, where multiple jobs from τ_i in a $level - i$ busy period need to be checked for the temporal correctness, this is not the case in the presented schedulability condition ((6.17)). As a consequence, when $m = 1$ the analysis provided is still safe albeit pessimistic as the provided test is sufficient but not necessary, whereas the test available in the literature for single processor is both necessary and sufficient [BLV07b, BLV09].

Theorem 13 (Correctness of the Schedulability Condition (6.17))**.** *Let us assume that the term $A_i^k(t)$, in* (6.19) *, gives an upper-bound on the workload generated by all the jobs from tasks with priority lower than or equal to i, released before time t and executed non-pre-emptively on k processors (the upper-bound computation is the subject of later sect). If* (6.17) *is satisfied for all task τ_i then the task-set is schedulable.*

Proof. For a given $k \in [1,m]$, the equation in the brackets of (6.19) gives an upper-bound on the carry-in workload at time t, where (i) k processors execute non-pre-emptive workload coming from k lower or equal priority jobs released before time t, and (ii) $m-k$ processors execute pre-emptive workload coming from $m-k$ higher priority jobs released before time t.

Therefore, $WA_i^{diff}(t)$ as defined in Equation (6.19), which takes the max for all k, is an upper-bound on the carry-in workload at time t. From the definitions of $W_j^{NC}(t)$ and since $A_i^{NC}(t) = 0$, it holds for a given time-instant t that the sum $WA_i^{DIFF}(t) + \sum_{\tau_j \in \text{hp(i)}} W_j^{NC}(t)$ gives an upper-bound on the total workload at time t (including both carry-in and non carry-in workload from lower- and higher-priority tasks). The correctness of the condition given by Inequality (6.17) immediately follows from the meaning of this sum, i.e., if the condition is satisfied for a given t, then it means that any job from task τ_i will always be able to execute for at least $C_i - Q_i + \varepsilon$ time units within $D_i - Q_i$ time units from its release. Given that every job of τ_i will get the highest priority after executing for $C_i - Q_i + \varepsilon$ time units, it implies that all jobs of τ_i will have to execute its (at most) Q_i remaining time units within the last Q_i time units to their respective deadline, which they will always do. □

In order to assess the schedulability of τ_i it is important to quantify A_i^k where $k \in \{1,\cdots,m\}$. The derivation of upper-bounds for A_i^k is the subject of the subsequent section.

6.5 Fixed Task Priority Limited Pre-emptive Schedulability Test

Having a characterization of the lower priority interference that each higher priority task is subject to, enables the derivation of a sufficient schedulability condition.

The task-set is deemed schedulable for a given set of last non-pre-emptive region for all the task $\tau_i \in \mathscr{T}$ if:

$$\forall \tau_i \in \mathscr{T}, \forall k \in \{1,\cdots,m\}, \beta_i^k(Q_i) \geqslant A_i^k \tag{6.22}$$

where

$$\beta_i^k(Q_i) \overset{\text{def}}{=} \max_{t \in [0, D_i - Q_i]} \left\{ m \times t - \sum_{\tau_j \in \text{hp(i)}} W_j^{NC}(t) - m \times (C_i - Q_i) - \sum_{l=1}^{m-k} \max_{\tau_h \in \tau_1,\cdots,\tau_{i-1}}^{\ell} W^{\text{diff}}(\tau_h, t) \right\} \tag{6.23}$$

The computation of the Function (6.23) can be performed by analyzing the limited set of time instants Γ_i described previously.

$$F_i^k(t) = \max_{\{t' \in \Gamma_i \cup \{t\} | t' \leqslant t\}} \left\{ m \times t' - \sum_{\tau_j \in \text{hp(i)}} W_j^{NC}(t') - m \times C_i - \sum_{l=1}^{m-k} \max^{\ell}_{\tau_h \in \tau_1, \cdots, \tau_{i-1}} W^{\text{diff}}(\tau_h, t') \right\} \tag{6.24}$$

Equation (6.23) may then be rewritten as:

$$\beta_i^k(Q_i) = F_i^k(D_i - Q_i) + Q_i \times m \tag{6.25}$$

6.6 Maximum Interference from Lower or Equal Priority Non-Pre-emptive Regions in ADS

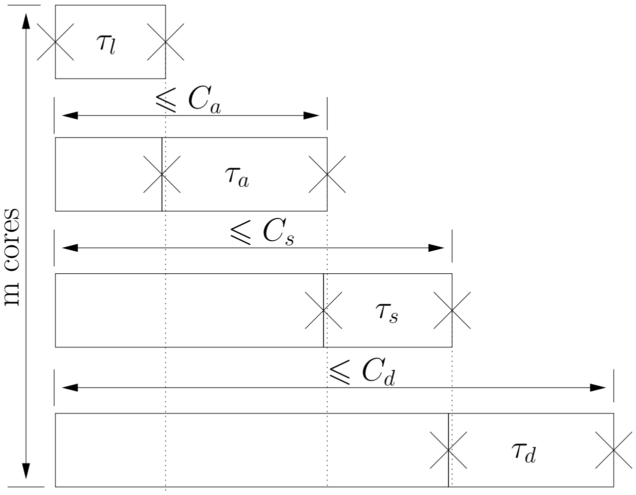


Figure 6.5: Maximum Interference Function Due to m Non-pre-emptive Regions of Lower or Equal Priority Tasks

The ADS scheduling policy considered in this section dictates that a task can only be dispatched to run on one of the m cores, at a time instant t, when either a core is idle or a pre-emption point of the lowest priority task running on any m processors has been reached. The task which is pre-empted is then the lowest priority task running on any of the cores at time t.

The worst-case interference pattern generated by the lower or equal priority non-pre-emptive region execution is represented in Figure 6.5 for a platform where $m = 4$. In Figure 6.5 the crosses represent fixed pre-emption points. In a scenario where m processors are executing lower or equal priority workload and several higher priority releases occur, the first task to be pre-empted is τ_l which is of lower priority than the remainder ($\tau_l \prec \tau_a \prec \tau_s \prec \tau_d$) at time t_l when a pre-emption point from τ_l is reached. In the worst case scenario task τ_a enters a non-pre-emptive region at time $t_l - \varepsilon$ and its next pre-emption point is reached at $t_l + \varepsilon + Q_a$. Subsequently in the worst-case situation τ_s has entered a non-pre-emptive region just before the pre-emption point of τ_a was reached.

Let us assume the total ordered set $\mathscr{LQ}_i = \{Q_n, \cdots, Q_i\}$ where, if $d > l$ then $Q_d \succ Q_l$. Each element of the $\mathscr{LQ}_i$ set represents the length of the non-pre-emptive region of task with priority equal or lower than τ_i, the priority ordering among tasks is represented by the total ordering of the elements of the set.

Given a subset $\mathscr{SQ}_i$ of $\mathscr{LQ}_i$ with k elements one can compute the A_i^k area accurately, this is achieved in Algorithm 9. Algorithm 9 places non-pre-emptive regions in $\mathscr{SQ}_i$ in priority order. The first element Q_1 is the lowest priority element in $\mathscr{SQ}_i$ (observe that in case the subscript ℓ in Q_ℓ references the ℓth element in the totally ordered set $\mathscr{SQ}_i$), in the worst case its pre-emption point will be reached at $t_1 = Q_1$. At the end of the first iteration the variable *span* is equal to Q_1. The span variable keeps track of the rightmost pre-emption point from all the tasks when these are placed in priority ordering in order to construct the maximum A_i^k from the set of $\mathscr{SQ}_i$ values. Some task τ_s may not have enough C_s such that its pre-emption point would be placed at $span + Q_s$, in which case the *span* variable either remains constant if $C_s < span$, or $span = C_s$ otherwise.

Algorithm 9: Low Priority Interference Computation from a $\mathscr{SQ}_i$ subset of k Lower or Equal Priority Non-pre-emptive Regions

```
Input  : $\mathscr{SQ}_i, i, k$
Output: $A_i^k$
$A = 0$
span $= 0$
for $y \in \{k, \cdots, 1\}$ do
    if $C_y \leqslant span + Q_y$ then
        if $C_y > span$ then
            $span = C_y$
        $A = A + C_y$
    else
        $A = A + span + Q_y$
        $span = span + Q_y$
return $A_i^k$
```

Algorithm 9 takes as input a set of k tasks with lower or equal priority than τ_i and returns the *exact* worst-case interference from these k tasks on task τ_i – as a result, Algorithm 9 can be used to derive the exact worst-case interference from the lower priority tasks on any task τ_i. However, doing so would require to enumerate all possible subsets of k tasks out of the set of all the tasks with a lower or equal priority than τ_i and the computation of the exact interference would be of exponential complexity.

As a compromise between accuracy and computation time, three methods are proposed to derive an *upper-bound* on the worst-case lower priority interference. The first one is the simplest (the least accurate and fastest) as it factors neither the task priorities, nor the WCET constraints in the computation and considers the maximum non-pre-emptive region length from all lower priority non-pre-emptive regions.

ADS Blocking Estimation 1.

The most straightforward method relies on considering the largest lower priority non-pre-emptive region and constructing the A_i^k area with it in conjunction with at most a single instance of the non-pre-emptive region of task τ_i so as to encompass the self-pushing effect. This bound is stated in (6.26).

$$A_i^k \leqslant \frac{k}{2} \times (k+1) \times \max\{\mathscr{L}\mathscr{Q}_i\} \tag{6.26}$$

ADS Blocking Estimation 2.

The second method is slightly more complex, it considers a variety of last non-pre-emptive regions present in $\mathscr{S}\mathscr{Q}_i$. The largest interference is obtained when the largest element of $\mathscr{L}\mathscr{Q}_i$ is accounted for k times (assuming the lowest priority for this task), the second largest is added up $k-1$ times (assuming the second lowest priority for this task), etc., until the k^{th} largest element which is only considered once. This upper-bound is stated in (6.27).

$$A_i^k \leqslant \sum_{j=1}^{k} Q_j^{\max} \times (k-j+1) \tag{6.27}$$

where $Q_j^{\max}$ denotes the j^{th} largest element in the set $\mathscr{L}\mathscr{Q}_i$.

ADS Blocking Estimation 3.

By taking the priority ordering among the lower or equal priority tasks into account, it is possible to construct a less pessimistic upper-bound on the A_i^k quantity. The problem can be formulated as follows: Find k $\mathscr{L}\mathscr{Q}_i$ element indexes $\{x_1, x_2, \ldots, x_k\}$ such that for all $j \in [1,k]$: $x_j \in [1, n-i]$, $\tau_{x_j} \prec \tau_{x_{j+1}}$ and the Function (6.28) is maximized

$$F = \sum_{j=1}^{k} Q_{x_j} \times (k-j+1) \tag{6.28}$$

Henceforth a method providing a solution for this problem is presented in Algorithm 10. Firstly, consider the following problem reformulation which will introduce notation required for the Algorithm description and allow for its explanation.

Problem 1: Given a set of non-negative values $\{Q_1, Q_2, \dots, Q_{n-i+1}\}$ ordered by task priority (note that these subscripts relate to the position of the element in the totally ordered set, higher priority is associated to higher set index value) and a non-negative integer $k \leqslant n-i+1$, a table T was constructed with k rows and $n-i+1$ columns such that the value $v_{y,z}$ of the cell in row y and column z is set to $v_{y,z} = (k-y+1) \times Q_z$ (rows are indexed from 1 to k and columns from 1 to $n-i+1$). The problem consists of finding S for which there exists $\{x_1, x_2, \dots, x_k\}$ such that each x_j, $1 \leq x_j \leq n-i+1$, denotes the index of a column and it holds that

1. $x_j < x_k, \forall\ 1 \leq j < k$, and
2. $S = \sum_{y=1}^{k} v_{y,x_j}$ is maximum

The solution for this problem can be visualized as finding a path from the level k to level 1 in the network present in Figure 6.6. at each chosen node only nodes to the left can be taken as admissible entries into the path.

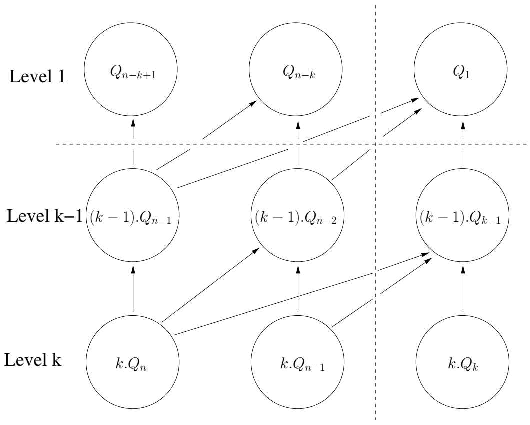


Figure 6.6: Depiction of the solution provided for Problem 1

It is easy to see that a solution S to problem 1 is also a maximum value for the sum of (6.28), and thus an upper-bound on A_i^k.

Lemma 6. *$S = v'_{k,n-i+1}$ is a solution to problem 1.*

Proof. The proof is obtained by (double) induction, first on y (row index) and then on z (column index). It is shown for all y and z, with $1 \leq y \leq k$ and $1 \leq z \leq n-i+1$, that $S = v'_{y,z}$ is the maximum value of the sum $\sum_{\ell=1}^{y} v_{\ell,x_\ell}$, assuming that the k variables $x_1, x_2, \dots x_k$ are such that $x_\ell \in [1,z]$, $\forall \ell$.
base case: $y = 1$ and $z = 1$.

The case is straightforward: $v'_{1,1} = v_{1,1}$ and thus $S = v'_{1,1}$ is the maximum value of the sum $\sum_{\ell=1}^{y} v_{\ell,x_\ell}$ where there is only a single variable x_1 and $x_1 = 1$ is the only choice.

Algorithm 10: Algorithm to Compute S

The idea is to construct another table T' based on T as follows:

1. As T, the table T' has k rows and $n-i+1$ columns.
2. $v'_{1,1} = v_{1,1}$
3. For the other cells $v'_{1,z}$ of the first row, with $z = 2,\ldots,n-i+1$,
   $v'_{1,z} = \max(v_{1,z}, v'_{1,z-1})$
4. For each row $y = 2,\ldots,k$:
   (a) Set $v'_{y,y} = v_{y,y} + v'_{y-1,y-1}$
   (b) The other cells $v'_{y,z}$ of the y^{th} row, with $z = y+1,\ldots,n-i-k+y+1$,
   $v'_{y,z} = \max(v_{y,z} + v'_{y-1,z-1}, v'_{y,z-1})$.

Finally, $S = v'_{k,n-i+1}$

Inductive step on z: $y = 1$ and $1 < z \leq n-i+1$.

By the induction, it is assumed that for $y = 1$ and for all $p \in [1, z-1]$, $S = v'_{1,p}$ is the maximum value of the sum $\sum_{\ell=1}^{y} v_{\ell,x_\ell}$ where there is a single variable x_1 and x_1 is chosen within $[1,p]$.

By construction, $\forall z = 2,\ldots,n-i+1$, the value $v'_{1,z}$ is defined as $v'_{1,z} = \max(v_{1,z}, v'_{1,z-1})$ and thus either $S = v'_{1,z}$ is equal to $v'_{1,z-1}$ (the maximum previously recorded and x_1 is chosen within $[1,z-1]$) or it is equal to $v_{1,z}$, in which case $S = v_{1,z}$ is the maximum and $x_1 = z$ leads to the maximum sum $\sum_{\ell=1}^{1} v_{\ell,x_\ell}$ ($= S$).

Inductive step on y, base case on z: $1 < y \leq k$ and $z = y$.

The value $v'_{\ell,\ell}$ is defined as $v'_{z,z} = v_{z,z} + v'_{z-1,z-1} = \sum_{\ell=1}^{z} v_{z,z}$. Since $x_\ell < x_{\ell+1}$ for all $1 \leq \ell < y$, the only choice for the y variables $x_1, x_2, \ldots x_y$ is to have $x_\ell = \ell$ for all $\ell \in [1,y]$. Therefore, $S = v'_{z,z}$ is the maximum value of the sum $\sum_{\ell=1}^{z} v_{\ell,\ell}$.

Inductive step on y and z: $1 < y \leq k$ and $1 < z \leq n-i+1$.

By the induction, it is assumed that for all $r \in [1, y-1]$ and for all $p \in [1, z-1]$, $S = v'_{r,p}$ is the maximum value of the $\sum_{\ell=1}^{r} v_{\ell,x_\ell}$ where the variables $x_1, x_2, \ldots, x_r$ are chosen within $[1,p]$.

By definition, the maximum value of the sum $\sum_{\ell=1}^{y} v_{\ell,x_\ell}$ assuming that the y variables $x_1, x_2, \ldots, x_y$ are chosen within $[1,z]$, is equal to the maximum between

1. the maximum value of the sum $\sum_{\ell=1}^{y} v_{\ell,x_\ell}$ assuming that the y variables $x_1, x_2, \ldots, x_y$ are chosen within $[1, z-1]$, and
2. the maximum value of $\sum_{\ell=1}^{y-1} v_{\ell,x_\ell}$ assuming that the $y-1$ variables $x_1, x_2, \ldots, x_{y-1}$ are chosen within $[1, z-1]$ and $x_y = z$.

This is reflected at step (4b) where $v'_{y,z}$ is set to the maximum of both.

It is thus true that $v'_{y,z}$ holds the maximum value of the sum $\sum_{\ell=1}^{y} v_{\ell,x_\ell}$ assuming that the variables $x_1, x_2, \ldots, x_y$ are chosen within $[1,z]$. As the result holds for $y = k$ and $z = n-i+1$ then $S = v'_{k,n-i+1}$ is a solution to problem 1. □

Algorithm 10 tests at most $(n-i-k+1)\cdot k$ different scenarios since it traverses at most $n-i-k+1$ elements k times, which compares favourably with the brute force approach which would test the $\binom{n}{k}$ different legal scenarios.

So far the A_i^k area has been upper-bounded by only taking into consideration the priority ordering between the non-pre-emptive regions of lower or equal priority tasks. It might be the case in fact that the worst-case execution time of the lower or equal priority tasks does not enable the result obtained with Algorithm 10 ever to occur in practice.

In the worst-case the blocking area cannot exceed the sum of the k largest lower or equal priority task WCET:

$$A_i^k \leqslant \sum_{j=1}^{k} \max_{\ell \in lep(i)}^{j} C_\ell \tag{6.29}$$

6.7 System Predictability with Fixed Non-pre-emptive Regions

Let us consider a scenario where the priority ordering among tasks is provided *a-priori*. The goal is to compute a set of non-pre-emptive regions for each task such that the number of pre-emptions observed in a schedule is reduced. A first approach to solving this problem is presented in Algorithm 11. The task-set is parsed starting from the lowest priority task τ_n. At each priority level i, the set of minimum Q_i^k values which render the m schedulability constraints (6.22) are found. From these m values the largest one is chosen. Since the $\beta_i^k(Q_i)$ functions are monotonically non-decreasing, if $Q_i^{k'} > Q_i^k$ and $\beta_i^k(Q_i^k) = A_i^k$ then $\beta_i^k(Q_i^{k'}) \geqslant A_i^k$. Hence choosing the maximum Q_i^k out of all the m minimum values which make the m inequalities true will still ensure the attainment of the schedulability condition. If, for any of the m schedulability conditions there exists no Q_i quantity for which $\beta_i^k(Q_i) = A_i^k$ then the task-set is deemed unschedulable.

Algorithm 11: Minimum Last Non-pre-emptive Region Length (Q_i) Assignment

```
for $i \in \{n, \cdots, 1\}$ do
    for $k \in \{1, \cdots, m\}$ do
        if $\exists \{Q_i | \beta_i^k(Q_i) = A_i^k\}$ then
            $Q_i^k = \{Q_i | \beta_i^k(Q_i) = A_i^k\}$
        else
            return UNSCHED
    $Q_i = \max_{1 \leqslant k \leqslant m}\{Q_i^k\}$
return SCHED
```

Lemma 7 (Minimum Non-pre-emptive Region Assignment)**.** *Algorithm 11 provides the minimum set of Q_i values $\forall \tau_i \in \mathscr{T}$ such that the task-set is schedulable under ADS with a given priority assignment*

Proof. Proof by induction. For task τ_n the quantity Q_n computed by Algorithm 11 is the smallest last non-pre-emptive region length such that task τ_n is schedulable. As a consequence, the blocking that τ_n induces on the higher priority task is the minimum possible such that τ_n is schedulable.

Inductive step: Algorithm 11 yields the minimum last non-pre-emptive region length for a task $\tau_i, 1 \leqslant i \leqslant n$ such that τ_i is schedulable. As a consequence of this the set of task $\{\tau_i, \cdots, \tau_n\}$ induces the lowest possible worst-case blocking to the higher priority workload such that those tasks are schedulable.

If for the same priority assignment any value $Q'_i \in \{Q'_i, \cdots, Q'_n\}$ it would happen that $Q'_i < Q_i$, then task τ_i would be unschedulable as a consequence. □

Theorem 14. *The ADS policy dominates the fully pre-emptive and fully non-pre-emptive global fixed task priority with respect to schedulability*

Proof. This result is easily proven by observing that according to Lemma 7 the Q vector outcome of Algorithm 11 is the smallest such that the taskset is schedulable. As a consequence, if $\mathscr{T}$ is schedulable under fully pre-emptive global fixed task priority the set Q resulting from Algorithm 11 is such that $\forall Q_i \in \{Q_1, \cdots, Q_n\} : Q_i = 0$. Otherwise if $\mathscr{T}$ is not schedulable with fully pre-emptive global fixed task priority but it is with ADS then $\exists Q_i \in Q : Q_i > 0$. Similarly if a task is only schedulable with fully non-pre-emptive the Q vector produced by algorithm 11 would be such that each $Q_i = C_i$. Since $A_i^k \leqslant \sum_{j=1}^{k} \max^{j}_{\ell \in lp(i)} C_l$ the maximum blocking lower priority tasks induce in ADS can in the worst-case be equal to that of fully non-pre-emptive and never greater. In a situation where $\forall i, Q_i = C_i$ the ADS policy is equivalent to the fully non-pre-emptive policy (i.e. the schedules produced are identical). □

Having the mechanism to produce the minimum set of Q values which ensures the schedulability of the task-set in ADS the next step is to compute a set of non-pre-emptive regions where at least some of its constituents are larger than the corresponding components of the minimum vector but never smaller. Having larger Q_i potentially leads to a smaller number of pre-emptions in the actual schedule as will be showcased in the Experimental Section.

Algorithm 12 takes as input the minimum Q vector ensuring schedulability. Contrary to the minimum Q vector computation procedure, now a top down approach in used (i.e. starting from the highest priority to the lowest). The resulting Q' vector has all its elements larger or equal to the minimum Q_i vector since by definition this is the smallest possible

Algorithm 12: Last Non-pre-emptive Region Length (Q_i) Assignment

```
for $i \in \{1,\cdots,n\}$ do
    for $k \in \{1,\cdots,m\}$ do
        $Q_i^k = \max\{Q | \forall j \in hep(i), \beta_j^k(Q_i) \geqslant A_j^k\}$
    $Q_i = \min_{1 \leqslant k \leqslant m}\{Q_i^k\}$
```

ensuring schedulability. At each priority level the maximum Q_i quantity is assigned which still preserves the schedulability of higher priority tasks. It is considered that any remainder lower or equal priority task τ_ℓ has a Q_ℓ equal to the maximum between any Q_j where $j \in hep(i)$ and the minimum Q_ℓ which renders τ_ℓ schedulable. This is due to the plausible scenario where a tasks with lower priority than τ_i processed in further iterations requesting a greater value than its minimum Q_ℓ. Since at the given iteration the algorithm does not have information about future developments and in order to reduce complexity the future requests of lower priority tasks are limited to the values known to the algorithm in the current iteration. These are the set of minimum last non-pre-emptive region lengths ensuring the schedulability of each lower priority task and the set of assigned higher priority task last non-pre-emptive regions.

6.8 Experimental Section for an Overhead-free Platform Model

In this chapter the theory for limited pre-emptive global fixed task priority scheduling is presented. In order to assess the performance of this scheduling discipline firstly the relative performance of the three methods of estimating the blocking induced by lower or equal priority workload are examined.

6.8.1 Blocking Estimation

A total of 100 sets are generated, with n Q elements. These sets are intended to represent the last non-pre-emptive regions from all tasks in a given taskset. Each last non-pre-emptive region length is a randomly generated value in the range $[0,300]$. The quantity A_1^k to A_{n-k}^k is upper-bounded for each of these Q sets using the three methods described previously. The estimations are performed for each generated set of Q values starting with priority 1 (i.e. computing A_1^k) until priority level $n-k$ (A_{n-k}^k). The average last non-pre-emptive region length (Q_i) is computed over the 100 task-sets for each priority level i using each of the three methods. The results are presented in Figure 6.7.

From the results in Figure 6.7 it is clear that the third estimation mechanism outperforms the first two as expected. The first one is the crudest approximation, its estimations tend to be much more pessimistic than the other two. Whereas the second one, albeit simple

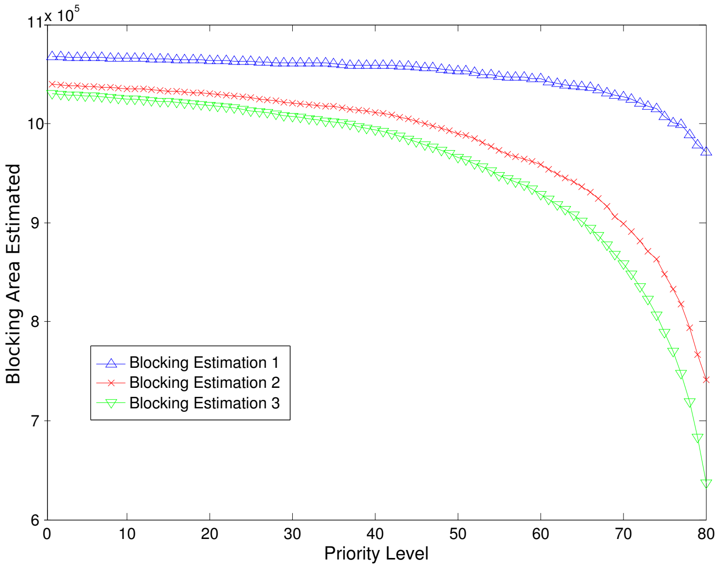


Figure 6.7: Blocking Estimations (k=8,n=88).

enough, provides results that are similar to the third and most complex of the three. As the priority level decreases (i.e. task index increases) the estimations tend to decrease since any subset of k values will necessarily be smaller than or equal to any in a larger set. The two latter methods tend to decrease their estimation faster as the priority level increases since the number of values to chose from decreases whereas the first method, by basing its estimation on the maximum value present in the set will not reduce its estimation as steeply.

6.8.2 Pre-emptions in Simulated Schedules

In order to assess the performance of the ADS scheduling policy with respect to the observed pre-emptions in a given schedule a simulator was created. Task-sets are randomly generated and the schedule produced by fully pre-emptive global fixed task priority and ADS is generated. In the simulated schedules the number of direct pre-emptions is extracted. Each task-set is randomly generated where $Utot$ is the target total utilization. The individual task utilizations $0 < u_i \leqslant 1$ are obtained through the random fixed sum algorithm [ESD10] which provides uniformly distributed task-sets with total utilization in excess of 1. The execution requirement of each task C_i is a uniformly distributed random variable in the interval $[100, 500]$. The relative deadlines of the tasks are computed then as $D_i = \frac{C_i}{u_i}$. The period of each task T_i is equal to the relative deadline ($T_i = D_i$).

The priority assigned to the tasks is the same for both fully pre-emptive and ADS simulations. The heuristic employed to assign priorities is DkC [DB09]. The schedules are simulated for platforms comprising m cores. In Figures 6.8 and 6.9 $m = 2$. Whereas Figures 6.10 6.11 relate to simulations on four processors. The simulations are run for 5000000

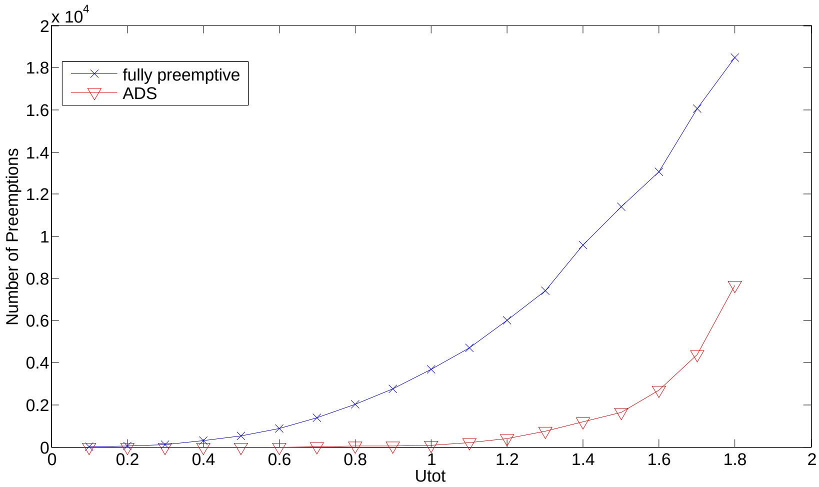


Figure 6.8: Observed Pre-emptions in Simulated Schedules, m=2,n=20

time units. The total utilizations of the taskset are varied from 0.1 to m with steps of 0.1 units. At each utilization level 100 random task-sets are generated and their schedules simulated.

Since ADS is compared against the global fully pre-emptive fixed task priority only task-sets which are schedulable by the latter are considered. As a consequence, while computing the last non-pre-emptive regions for each task with Algorithm 12 the minimum Q vector considered is one where all elements are zero. Since the performance of the blocking estimation 1 is the most modest, this was put to use in order to get a sense of the worst-case performance of ADS and to show that even in those circumstances it compares quite favourably against the fully pre-emptive scheduler with respect to run-time pre-emptions.

From the presented results it is apparent that a large number of the pre-emptions are removed from the actual schedule. From figures 6.8 to 6.11 it is obvious that the number of pre-emptions in ADS tends to decrease with increases in n. This is due to the spread of the available utilization among constituents of the task-set. Consequently tasks will tend to have moderately similar deadlines and execution requirements. This is beneficial for obtaining larger admissible non-pre-emptive regions when compared to the execution time. The number of pre-emptions in both scheduling policies increase with the total utilization of the task-set, still the pre-emption increase in fully pre-emptive tends to be steeper than in the ADS schedule. Since the processors tend to be occupied for larger time intervals it is more likely that newly released jobs will induce a pre-emption.

As the number of processors increase the relative benefits of the ADS policy suffer a mild degradation (comparison between m=4 and m=2). This is due to the poor performance of the blocking estimation mechanism put to use in this simulation effort (ADS estimation 1) as it will severely over-estimate the actual worst-case blocking time tasks will be subject

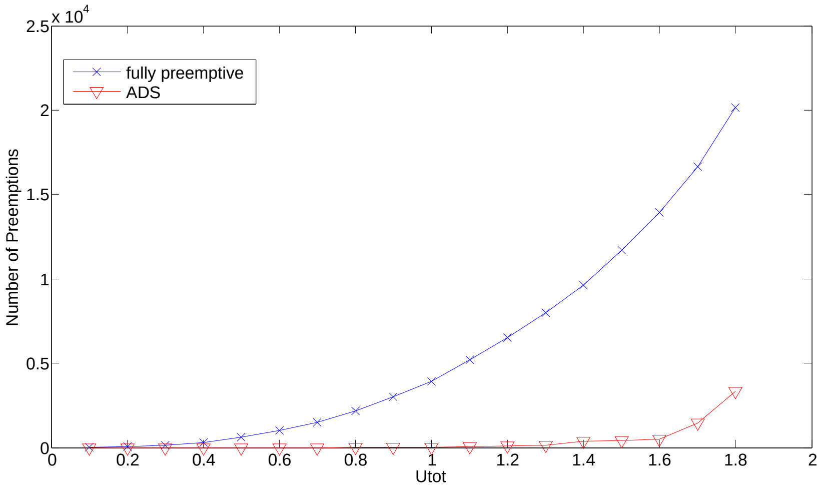


Figure 6.9: Observed Pre-emptions in Simulated Schedules, m=2,n=32

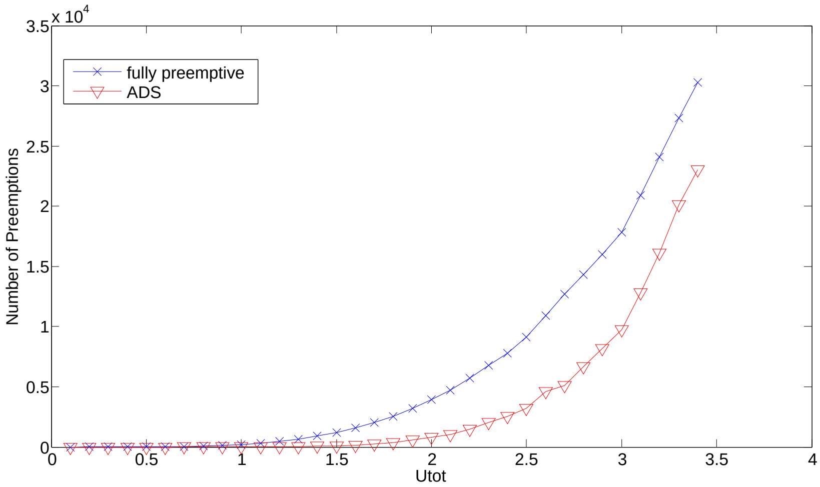


Figure 6.10: Observed Pre-emptions in Simulated Schedules, m=4,n=32

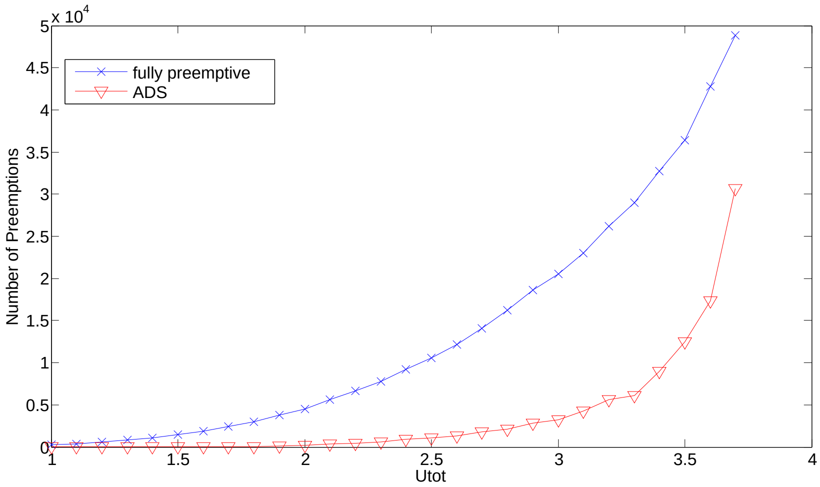


Figure 6.11: Observed Pre-emptions in Simulated Schedules, m=4,n=64

to and as a consequence will lead to smaller non-pre-emptive region lengths. This induces more pre-emptions points in the tasks and hence more possibilities for pre-emptions to occur. To be noted that the blocking estimation 1 and estimation 2 in this case would yield similar results since by taking a top down approach and by assuming that the lower priority non-pre-emptive regions would be equal to Q_i, the A_i^k estimate is the same for both methods as there would exist only a single distinct non-pre-emptive region length value which would be mandatorily the maximum. Another shortcoming general to all the approximate blocking estimation mechanisms presented in this work is that these do not take into account the maximum execution requirement of the lower or equal priority tasks. As m increases so does the pessimism involved in the estimation step since the stair-case pattern of blocking is subject to cruder overestimations as the number of steps increases.

6.8.3 Schedulability assessment of RDS vs. ADS

This section gathers experimental data comparing the schedulability performance of the RDS against the ADS policy. The experimental results were extracted using DkC [DB09] priority ordering. Task-sets are generated using the random fixed sum algorithm [ESD10] providing uniformly distributed task-sets with total utilization in excess of 1. The variable m denotes the number of cores and n the total number of tasks. Results are presented for the following platform configuration pairs ($\{m,n\}$): $\{2,10\}$, $\{2,20\}$, $\{4,20\}$ and $\{4,40\}$. The results are obtained considering total utilizations ranging from $U_{tot} = \frac{m}{16}$ until $U_{tot} = m - \frac{m}{16}$ with a stepping of $\frac{m}{16}$ between consecutive points. For each utilization point 1000 task-sets are generated and the schedulability of these is assessed using the ADS, RDS and fully pre-emptive schedulability tests. The response time of each task is computed, starting from the lowest priority. For each task τ_i which are not schedulable with $Q_i = 0$, the smallest possible last non-pre-emptive region ensuring schedulability is computed (if one exists). The label ADS refers to the method presented in [DBM+13] with a single last non-pre-emptive region (i.e. all tasks have at most one non-pre-emptive region, the remainder of the workload executes pre-emptively).

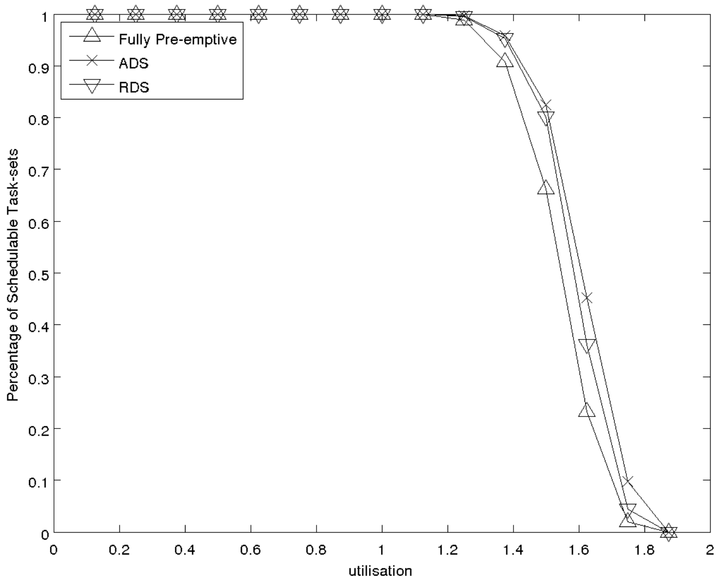


Figure 6.12: Results for m=2 and n=10 with $T_i \in [400, 40000]$

The figures 6.12 to 6.15 relate to scenarios where the execution time of the tasks varies between 40 and 40000. As previously observed in [MNP+13], ADS appears not to behave well with increases on the number of processors. This is due to the inherent pessimism taken when upper-bounding the worst-case blocking suffered in ADS. It grows larger with the number of available cores. Increasing the number of tasks also makes the schedulability of ADS deteriorate in comparison to RDS. The RDS scheduling policy seems to provide a better overall performance in comparison to ADS, in scenarios where the pre-emption delay is negligible. Nevertheless it is apparent that no domination relation exists between the two (ADS and RDS). As expected the fully pre-emtive scenario performs not as well as the limited pre-emptive strategies.

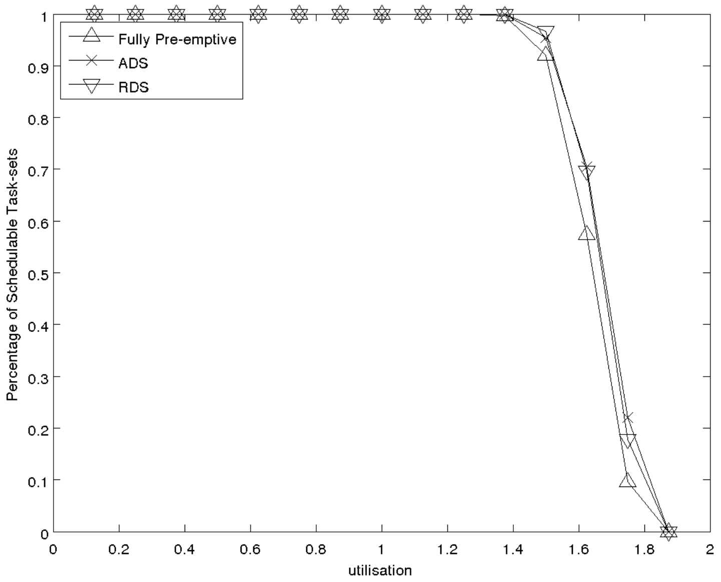


Figure 6.13: Results for m=2 and n=20 with $T_i \in [400, 40000]$

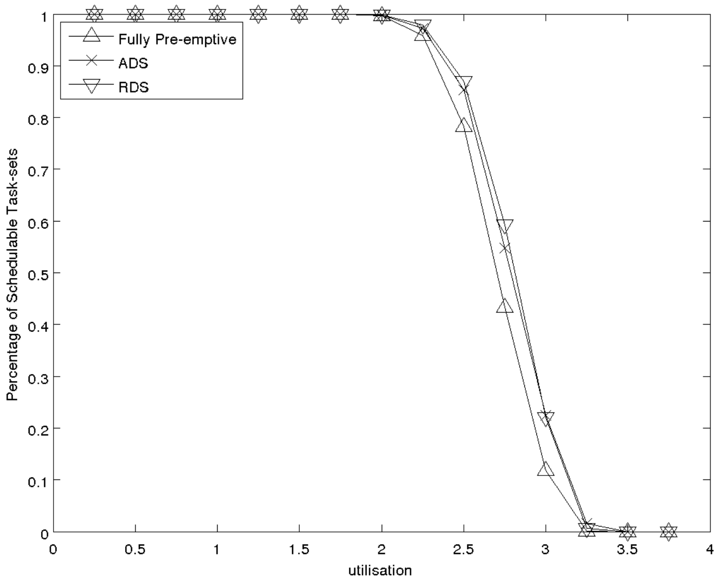


Figure 6.14: Results for m=4 and n=20 with $T_i \in [400, 40000]$

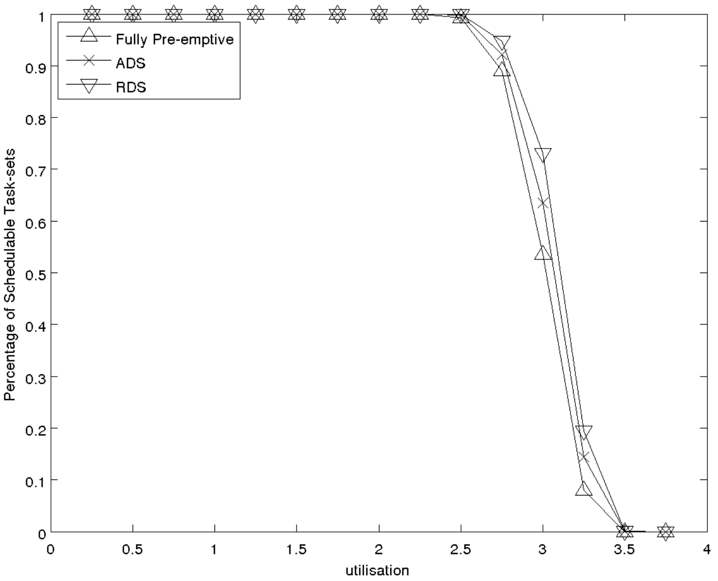


Figure 6.15: Results for m=4 and n=40 with $T_i \in [400, 40000]$

6.9 Accounting for Pre-emption Delay in the Global Schedule

The limited pre-emptive theory is of little interest if the actual pre-emption delay is not integrated into the schedulability analysis. In this section bound on the additional workload due to the pre-emptive behaviour are obtained for both fully pre-emptive and the limited pre-emptive models.

6.9.1 Pre-emption and Migration Delay Bound for Fully Pre-emptive GFTP

In multiprocessor, alongside pre-emptions, migrations can additionally. A pre-emption may originate a migration (in case these are allowed by the scheduler) if the pre-empted workload commences to execute on another processor. When a task migrates a similar phenomenon to cache pollution from pre-empting tasks occurs as the cache of the processor the task migrated to will hypothetically not hold any of the memory lines the migrated task will reference when executing. This delay is generally termed by CPMD (Cache related pre-emption and migration delay).

An upper-bound on the pre-emption delay generated in the level-i fully pre-emptive GFTP schedule can be derived by considering that:

- each higher priority task released can induce at most one pre-emption or migration
- when a job from task τ_j induces a pre-emption or migration on a jobs of tasks τ_k where $k > j$:
  1. if the pre-emption delay is suffered by some task τ_k $(j < k < i)$ then the overhead is trivially treated as higher priority workload
  2. if τ_i suffers the pre-emption then two scenarios might occur:
     (a) the pre-emption delay is paid while other higher priority workload is executing on the other cores. In this situation the additional workload constituted by the pre-emption delay is indistinguishable from another higher priority workload and hence can be considered to have an equal interference contribution (divide the workload by m).
     (b) the pre-emption delay is paid when at least one of the remainder cores is idle on work of priority level-i. In this scenario it might seems as the interference contribution by $CPMD$ units the pre-emption delay exceeds the $\frac{CPMD}{m}$, as there is no simultaneous contribution from higher priority workload which is not directly interfeering with τ_i execution. Nevertheless note that in order for a first pre-emption to occur m prior higher priority job releases had to occur. Subsequent pre-emption occur if some k higher priority jobs terminate their execution allowing τ_i to regain access to one processor. Subsequently another k higher priority tasks have to be released for

> the next pre-emption to occur. If all higher priority tasks are assumed to virtually generate the worst-case pre-emption penalty a safe pre-emption delay upper-bound is obtained since even if the pre-emption delay is actually being paid while other processors are idle on level-i workload at least $m-1$ virtual CPMD compensations were considered in the analysis to fully accommodate the interference suffered by task τ_i.

From the aforementioned set of assumptions a function quantifying the level-i schedule pre-emption and migration delay is stated in Equation (6.30) as was derived by Devi [Dev06]. We have to consider that $m-1$ higher priority tasks were already executing and were pre-empted hence we consider the $m-1$ largest migration delay penalties to interfere with the execution of τ_i.

$$\mathrm{PD}^{\mathrm{FP}}(t,i) = \left(\sum_{k=1}^{m-1} \max_{h \in \{1,\cdots,i-1\}}^{k} \mathrm{CPMD}_h \right) + \sum_{j=1}^{i-1} \left\lfloor \frac{t}{T_j} \right\rfloor \cdot \max_{k \in \{j+1,\cdots,i\}} \{\mathrm{CPMD}_k\} \qquad (6.30)$$

6.9.2 Pre-emption and Migration Delay Bound for ADS GFTP

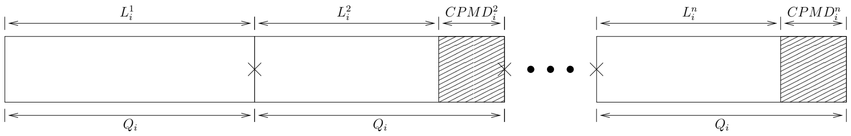


Figure 6.16: Execution model

In the limited pre-emptive schedule with multiple non-pre-emptive regions each job is composed of a set of non-pre-emptive regions delimited by pre-emption points. As is depicted in the Figure 6.16 each non pre-emptive region has a length at run-time which cannot exceed Q_i time units. In order to achieve this, the maximum distance for execution in isolation between the two pre-emption points delimiting the k^{th} non-pre-emptive region is $L_i^k = Q_i - CPMD_i^k$. Where $CPMD_i^k$ denotes the maximum pre-emption/migration delay that can be paid during the execution of the k^{th} non-pre-emptive region.

Without loss of generality we simplify the model by stating that $CPMD_i^1 = 0$ and $\forall k \in \{2,\cdots,n-1\}$ then $CPMD_i^k = CPMD_i^{k+1}$. As a consequence of this, $L_i^1 = Q_i$ and $\forall k \in \{2,\cdots,n-2\}$ we have that $L_i^k = L_i^{k+1}$. As for the last non-pre-emptive region we have that $L_i^n = C_i - N_i^{\mathrm{ADS}} \cdot L_i^2$.

Since, in the ADS framework, each task can only be pre-empted at a non-pre-emptive region boundary, a job from task τ_i, with an isolation WCET of C_i, can be pre-empted at most:

$$N_i^{\mathrm{ADS}} = \left\lfloor \frac{C_i - Q_i}{Q_i - \mathrm{CPMD_i}} \right\rfloor + 2 \tag{6.31}$$

$$\mathrm{PD}^{\mathrm{ADS}}(t,i) = \sum_{k=1}^{m-1} \max_{h \in \{1,\cdots,i-1\}}^{k} \mathrm{CPMD}_h + \sum_{j=1}^{i-1} \left\lfloor \frac{t}{T_j} \right\rfloor \cdot N_i^{\mathrm{ADS}} \cdot \mathrm{CPMD}_j \tag{6.32}$$

6.9.3 Pre-emption and Migration Delay Bound for RDS (Last Region only) GFTP

In this model, each job from a task τ_i executes the initial $C_i - Q_i$ units of work in a pre-emptive manner. It also holds true that any higher priority job release can induce at most one pre-emption or migration. Hence the RDS last region only model has the same pre-emptive bound defined in Eq. (6.30) where only the initial $C_i - Q_i$ units of workload are eligible to be pre-empted.

6.9.4 Pre-emption and Migration Delay Bound for RDS (Multiple Non-pre-emptive Regions) GFTP

In a similar fashion as for the ADS schedule each task can only be pre-empted at a non-pre-emptive region boundary, hence the upper-bound stated in equation (6.32) is itself a bound on the pre-emptions delay generated in the level-i RDS schedule.

Contrary to fully pre-emptive and ADS, in the RDS with multiple non-pre-emptive regions, each job release may potentially induce m pre-emptions. This effect is depicted in Figure 6.17. In this picture the following task index relationship exists $j < k < l < d$, i.e. τ_j has the highest priority and τ_d has the lowest.

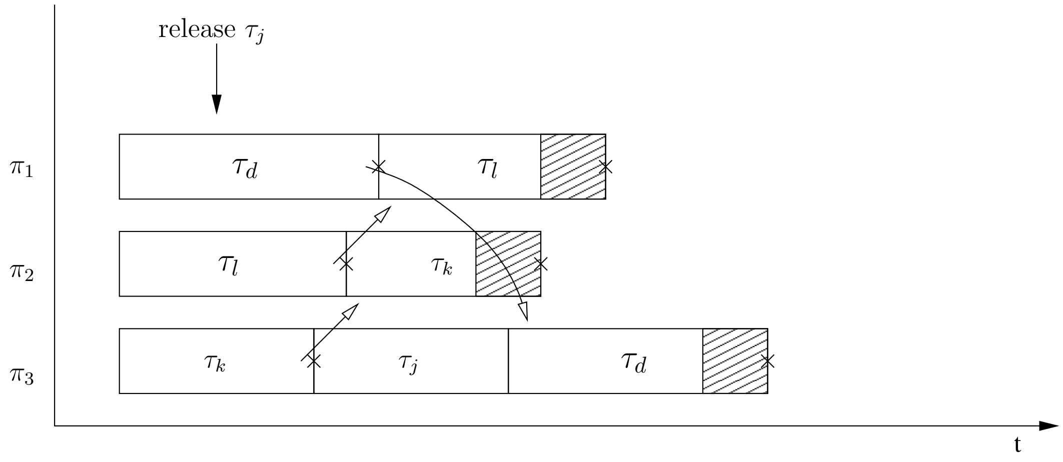


Figure 6.17: depiction of a m pre-emption chain triggered by a single job release

In accordance with this observation then the fully pre-emptive pre-emption and migration delay, unlike for ADS, is not an upper-bound on the pre-emption delay occurring in the RDS schedule with multiple non-pre-emptive regions.

All pre-emption delay functions are trivially integrated into the previously defined workload functions. The remainder of the schedulability tests remain unchanged.

6.10 Schedulability Assessment of ADS vs. Fully Pre-emptive with CPMD

In this section the performance of ADS is compared to the one of Fully Pre-emptive for platforms where the CPMD is non-negligible.

Task-sets are generated using the random fixed sum algorithm [ESD10] providing uniformly distributed task-sets with total utilization in excess of 1. The variable m denotes the number of cores and n the total number of tasks. Results are presented for the following platform configuration pairs ($\{m,n\}$): $\{2,10\}$, $\{2,20\}$ and $\{3,15\}$. The results are obtained considering total utilizations ranging from $U_{tot} = 1$ until $U_{tot} = m - 0.1$ with a stepping of 0.1 between consecutive points. For each utilization point 1000 task-sets are generated and the schedulability of these is assessed using the ADS, RDS and fully pre-emptive schedulability tests. The response time of each task is computed, starting from the lowest priority.

The experimental set-up is similar to the one employed in section 6.8.3. As an addition to the previous model a $CPMD_i$ cost is randomly generated for each task τ_i in the interval $[0,40]$. This constitutes a single pre-emption cost. Every time a job from task τ_i is pre-empted it will incur on an execution time increase of $CPMD_i$.

Since the higher priority interference is a function of the number of pre-emptions points each higher priority task has then the assignment of last non-pre-emptive regions is drawn from higher priorities downwards.

6.10.1 Discussion of Pre-emption Delay Results

The results comparing schedulability achieved in platforms with non-negligible pre-emption delay comparing the ADS policy with fully pre-emptive are displayed in figures 6.18 to 6.20. Here it is apparent that the limited pre-emptive scheduling policy ADS has a much better schedulability performance when compared to the fully pre-emptive mechanisms. In fact when compared to the results obtained in prior sections (figures 6.12 to 6.15) it is trivial to see that the relative gains presented by the ADS strategy dramatically increases when pre-emption delay is considered. As the task-set size increases so does the relative gain provided by the ADS strategy since in the fully pre-emptive scheduling there will be more higher priority releases and hence an increase in the estimation of pre-emptions and

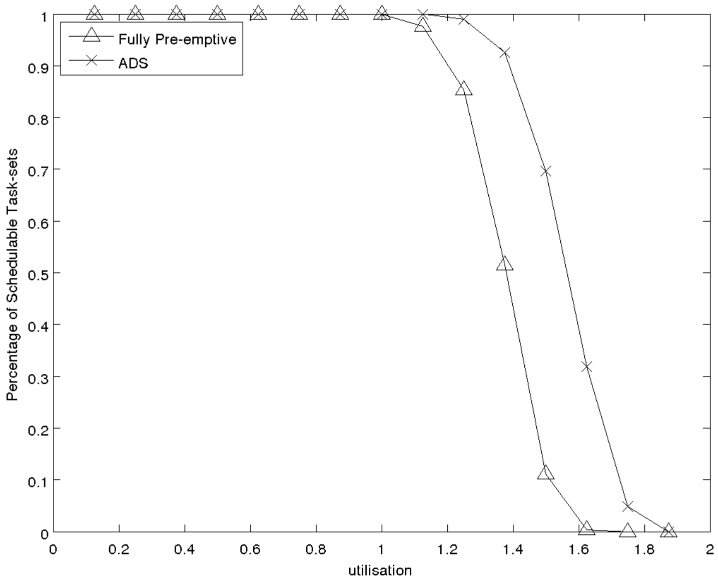


Figure 6.18: Results for m=2 and n=10 with $T_i \in [400, 40000]$ and CPMD value per Pre-emption as a random variable in the interval $[0, 40]$

migrations. The schedulability results for fully pre-emptive are taken as a proxy of those of the RDS policy with a single final non-pre-emptive region. This mode is subject to largely the same pre-emption delay penalty as fully-pre-emptive. Moreover the multiple non-pre-emptive region RDS is ignored as this will have a pre-emption delay function which is the same as the ADS policy but a schedulability which resembles fully non-pre-emptive since all lower priority non-pre-emptive regions are assumed to interfere with the execution of higher priority tasks.

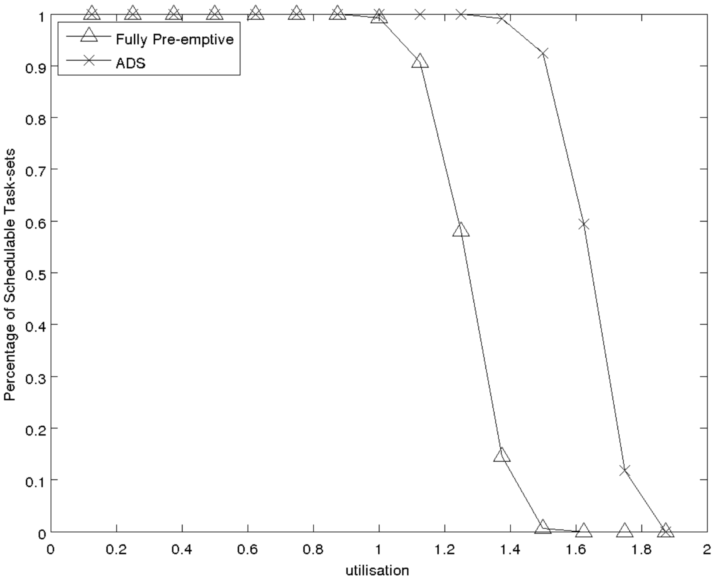


Figure 6.19: Results for m=2 and n=20 with $T_i \in [400, 40000]$ and CPMD value per Pre-emption as a random variable in the interval $[0, 40]$

6.11 Global Floating Non-pre-emptive Scheduling

Contrary to the fixed non-pre-emptive regions (as stated in previous chapters) the non-pre-emptive regions can be commanded by the scheduler and not rely on pre-specified pre-emption points. In the single processor case the floating non-pre-emptive region scenario is quite simple in nature due to the fact that $m = 1$. When there is only one task executing at the time of a higher priority release it is straightforward to assume that the pre-emption will be delayed as much as the higher priority schedule can accommodate before a deadline miss occurs in the future.

In the global multiprocessor scenario this decision is not so trivial. This stems from the fact that multiple tasks of lower priority than the current release may be executing on each processor. The question is then, how many different protocols can be derived and which desirable properties can be inferred from these?

For each task τ_i a maximum allowed lower priority blocking term can be straightforwardly derived from the sufficient schedulability condition. The sufficient schedulability equation for fully pre-emptive scheduling is presented in Equation (6.6). A sufficient blocking tolerance for any task τ_j is a quantity β_j, similarly to the single core scenario, such that the following relationship is held:

$$\forall \tau_j \in \mathscr{T}, \exists \beta_j, t \in [0, D_j] : I_j(t) + C_i + \beta_j = t \tag{6.33}$$

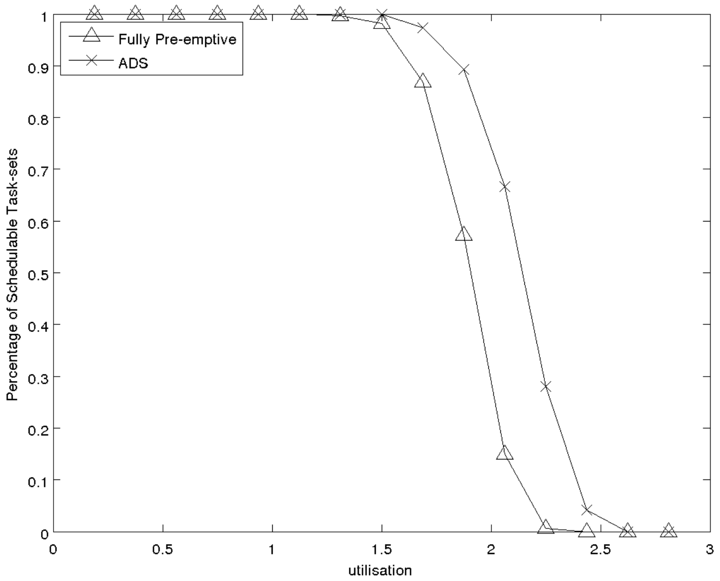


Figure 6.20: Results for m=3 and n=15 with $T_i \in [400, 40000]$ and CPMD value per Pre-emption as a random variable in the interval $[0, 40]$

In the scenario where the condition present in Equation (6.33) is met all tasks are fully pre-emptively schedulable and can endure an additional interference from β_j units of lower priority workload without jeopardizing their deadlines in any conceivable scenario.

It is important to realize that several protocols can be devised for the floating non-pre-emptive region in global multicore scheduling by exploiting variable degrees of information available at run-time and a variable degree of dynamism in the choices to be taken. Only a sub-set of these – composed of the ones which are deemed most relevant – is presented here. Also, only static methods are presented in order to ease access by the reader.

In the following discussion, in order to ease clarity of exposure and similarly to the single-core situation the variable NPL_i denotes the length of each non-pre-emptive region of task τ_i.

The simplest of these protocols is a direct adaptation of the single processor one:

Floating NP Protocol 1: After any task τ_j releases a job, in case there exists a job from any task τ_i where $i > j$ and τ_i is the lowest priority task in execution then a non-pre-emptive region commences in all m processors with a duration of NPL_i . Any number of additional tasks may release jobs during the interval where the non-pre-emptive region is occurring. When the non-pre-emptive region interval elapses the m highest priority jobs in the ready queue are dispatched onto the processors.

In order to guarantee the safety of this protocol the following condition has to be respected when deciding on the non-pre-emptive region length to assign tasks:

$$\mathrm{NPL}_i \leqslant \frac{\min_{1<k<i}\{\beta_k\}}{m} \tag{6.34}$$

The second protocol may also be portrayed as a derivation of the single processor floating non-pre-emptive region model for the global scheduler.

Floating NP Protocol 2: Each lower priority task τ_i executing upon a processor, whenever a higher priority release occurs – provided it is not currently executing non-pre-emptively– commences to execute non-pre-emptively for a predefined non-pre-emptive region length NPL_i.

This length is defined in a manner such that higher priority tasks can never suffer an interference which would jeopardize their deadlines' attainment. Consider the set $\mathrm{NPL}(a,k)$ to denote the set of non-pre-emptive region lengths $\{\mathrm{NPL}_a, \mathrm{NPL}_{a+1}, \cdots, \mathrm{NPL}_{k-1}, \mathrm{NPL}_k\}$.

The following set of conditions are sufficient to ensure that all higher priority tasks meet their deadlines in protocol 2 for any possible set of non-pre-emptive region length which respects the posed conditions.

$$\forall k \in \{1, \cdots, n\}, \beta_k \geqslant \sum_{k=1}^{m} \overset{k}{\max}\{\mathrm{NPL}(k,n)\} \tag{6.35}$$

These conditions are overall more flexible with respect to the choice of non-pre-emptive regions which are admissible to be chosen in comparison to the Condition (6.34). Still the simple solution for the assignment defined as $\mathrm{NPL}_i = \frac{\min_{1<k<i}\{\beta_k\}}{m}$ respects both conditions (6.34) and (6.35) and hence can be employed with both mechanisms.

In a situation where the non-pre-emptive region length is not monotonically decreasing with decrease in priority an additional term has to be considered when computing the interference each task can suffer. In fact each task is then subject to an upper-bound on its interference due to lower priority tasks (*LP*) which can be quantified as:

$$LP = \left\lfloor \frac{C_i}{\mathrm{NPL}_i} \right\rfloor \times \sum_{k=1}^{m-1} \overset{k}{\max_{i<\ell\leqslant n}} \{\mathrm{NPL}_i - \mathrm{NPL}_\ell\} \tag{6.36}$$

As far as the design-time guarantees are concerned, the Protocol 1 only ensures that each task τ_i can execute non-pre-emptively for a time interval which is of at least NPL_n time units. The off-line guarantees offered by Protocol 2 are better in the sense that each task τ_i is guaranteed to execute non-pre-emptively for NPL_i consecutive time units.

Other mechanisms can be devised to exploit additional run-time knowledge in the same light as what is exposed in Chapter 3.1 these are not studied for the multiprocessor.

6.12 Schedulability Increase for Fixed Non-pre-emptive GEDF

So far the limited pre-emptive scheduling has been solely discussed in the context of Global Fixed Task Priority disciplines. It can nevertheless be applied to Global EDF. The question remains whether there exists a schedulability gain when comparing fully pre-emptive to the limited pre-emptive model. In the single processor case – as a consequence of the optimality of EDF when scheduling constrained deadlines task-sets – the limited pre-emptive model does not yield an increased schedulability gain [YBB11b].

A simple example task-set present in table 6.1 suffices when showing that indeed the fixed non-pre-emptive model allows for a schedulability increase in the GEDF scenario.

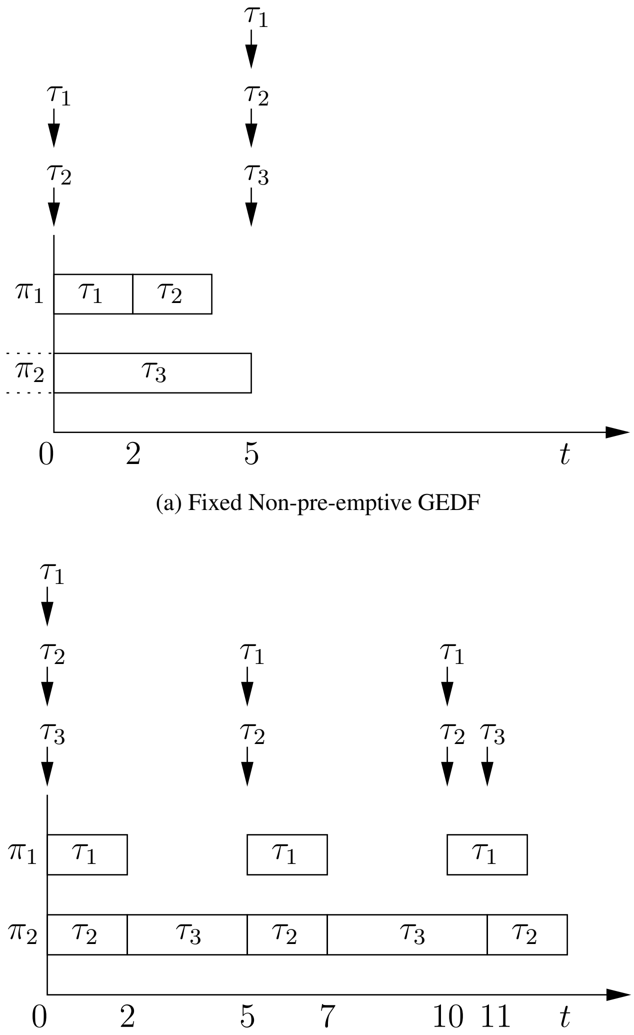
	C_i	D_i	T_i
τ_1	2	5	5
τ_2	2	5	5
τ_3	8	11	∞

Table 6.1: GEDF example task-set

Consider a platform composed of two processors (m=2)

First consider the case for the schedulability of tasks τ_1 and τ_2, a constructed worst-case scenario for these tasks is displayed in Figure 6.21a. In this initial case there exists a job of τ_3 which has a deadline at time instant 5 and as a consequence was released at $t = -6$. By absurdity, since one is solely focusing on the schedulability of tasks τ_1 and τ_2, consider that this job of task τ_3 suffered sufficient interference such that at $t = 0$ it still has a remaining workload of 5 time units which it completes by its absolute deadline. In this case (Figure 6.21a) task τ_2 is suffers an interference of 2 time units and terminates its workload at $t = 4$ long ahead of its deadline. Note that τ_1 and τ_2 are interchangeable since these have the same workload requests and relative periods and deadlines. Since both τ_1 or τ_2 cannot , in any other conceivable situation, suffer more interference when scheduled by GEDF than in this contrive example the logical conclusion that both (τ_1 and τ_2) will always meet their deadlines is derived.

Consider now the scenario depicted in Figure 6.21b. In this distinct scenario the schedulability of τ_3 is investigated. In this scenario we observe a synchronous release situation which leads to τ_3 suffering a total interference of 4 time units, this will lead to a deadline miss at $t = 11$. Nevertheless it is apparent that if task τ_3 would execute in a non-pre-emptive manner it would always be able to commence execution never later than 2 units since its release. Since once it would start to execute it would do so until completion it would terminate its computation at least 2 time units before its deadline. Bear in mind that the full non-pre-emptive nature of task τ_3 is in fact a particular case of the limited pre-emptive scenario. With this simple toy example the effective domination between Limited Pre-emptive


τ_1
τ_1 τ_2
τ_2 τ_3
π_1 τ_1 τ_2
π_2 τ_3
0 2 5 t
τ_1
τ_2 τ_1 τ_1
τ_3 τ_2 τ_2 τ_3
π_1 τ_1 τ_1 τ_1
π_2 τ_2 τ_3 τ_2 τ_3 τ_2
0 2 5 7 10 11 t

(a) Fixed Non-pre-emptive GEDF

(b) Fully Pre-emptive

Figure 6.21: GEDF Example Schedules.

GEDF and GEDF is established. This means that there are task-sets for which a valid limited pre-emptive GEDF schedule exists whereas GEDF itself fails to schedule the task-set.

Showing this relationship for very particular and concrete examples gives an indication that a LPGEDF analysis could be derived such that it would dominate the regular GEDF sufficient test. This work follows mainly on the lines of what has been previously described for Limited Pre-emptive GFTP.

Both the GFTP and GEDF schedulability test rely on the consideration of the set of tasks which may interfere with the one under analysis and the creation of upper-bounds on the interference these tasks can exert.

The schedulability test for global EDF is constructed by considering the general bound per higher priority task considered in Equation (6.2) in conjunction with another bound on the workload specific to the EDF priority ordering (Eq (6.37))

$$W_j^{NC-\mathrm{EDF}}(t) = \max\left(0, \left(\left\lfloor \frac{t-D_j}{T_j} \right\rfloor + 1\right) \times C_j\right) \tag{6.37}$$

The $WA_i^{diff-EDF}(t)$ function is defined as follows:

$$WA_i^{diff-EDF}(t) \overset{\text{def}}{=} \max_{k\in\{1,\cdots,m\}} \left\{W_j^{CI}(t) - W_j^{NC-\mathrm{EDF}}(t)\right\} \tag{6.38}$$

Following what is derived by Baruah in [Bar07] a task-set $\mathscr{T}$ is deemed schedulable on fully pre-emptive GEDF if the following condition holds:

$$\begin{gathered}\forall \tau_i \in \mathscr{T}, \forall L \in [0, L_{UB}^i] : \\ L + D_i - \frac{1}{m} \times \left(WA_i^{diff-EDF}(L+D_i) + \sum_{\tau_j \in \mathscr{T}} W_j^{NC-\mathrm{EDF}}(L+D_i) - C_i\right) + C_i \geqslant 0\end{gathered} \tag{6.39}$$

where L_{UB} is defined as [Bar07] :

$$L_{UB}^i = \frac{\sum_{k\in\{1,\cdots,m\}} max_{j\in\{1,\cdots,n\}}^k C_j - D_i \times (m - U(\mathscr{T})) + \sum_{j\in\{1,\cdots,n\}} (T_j - D_j) \times U_j + m \times C_i}{m - U(\mathscr{T})} \tag{6.40}$$

Following the observation that with limited pre-emptive scheduling each job cannot be pre-empted once it commences executing its final non-pre-emptive region then the schdulability test for the fixed non pre-emptive region model becomes:

$$\begin{gathered}\forall \tau_i \in \mathscr{T}, \forall L \in [0, L_{UB}^i] : L + D_i - Q_i - \frac{1}{m} \times \\ \left(WA_i^{diff-EDF}(L+D_i) + \sum_{\tau_j \in \mathscr{T}} W_j^{NC-\mathrm{EDF}}(L+D_i) - C_i\right) + C_i - Q_i \geqslant 0\end{gathered} \tag{6.41}$$

Consequently, for each task, a lower bound on the maximum allowed deferral time without jeopardizing the deadline obtention is written in a similar fashion to the GFTP:

$$\begin{gathered}\beta_i^{EDF} \overset{\text{def}}{=} \min_{L\in[0,L_{UB}]} \Bigg\{ B | L + D_i - Q_i - B - \frac{1}{m} \times \\ \left(WA_i^{diff-EDF}(L+D_i) + \sum_{\tau_j \in \mathscr{T}} W_j^{NC-\mathrm{EDF}}(L+D_i) - C_i\right) + C_i - Q_i = 0 \Bigg\}\end{gathered} \tag{6.42}$$

Chapter 7

Summary and Future Directions

Most of the computational mechanisms controlling physical processes must adapt to the timing constraints governing their dynamics. Characterizing the temporal properties of these computational processes is detrimental to ensure their fitness its purpose. By resorting to simplified models of the hardware platform and of the software these temporal details can be formally studied and propertied obtained and proven safe. This thesis borrows several widely employed models in the real-time community, namely the workload executing on each platform is described as a recurrent execution demand with a given worst-case quantity which is requested with a minimum temporal separation between consecutive instances. The hardware model is one where each task can only execute on one processor at a time and once a pre-emption occurs there can exist a potential pre-emption delay once the pre-empted task resumes execution.

By using these common models this thesis proposes novel contributions to the real-time are in four main areas.

Thesis Contributions:

- **Single Processor Limited Pre-emptive Theory Extension:**

  The static nature of the state of the art considering the limited pre-emptive model was extended. This allowed for a considerable reduction on the observed number of pre-emptions. Moreover a smaller upper-bound on the number of pre-emptions each task is subject to is provided (comparing the run-time extension against the static version of the limited pre-emptive scheduling). The run-time extension is only applicable to the floating non-pre-emptive model, naturally, as the fixed non-pre-emptive model relies on pre-specified pre-emption points which are not amenable to straightforward run-time changes. Similarly to the proven schedulability increase enabled by the fixed non-pre-emptive regions a similar behaviour was sought for the floating model. In this thesis it is shown that even though the floating non-pre-emptive model does dominate the fully pre-emptive fixed task priority from the schedulability point of view, it is hard to analyse and a critical instant is currently unknown. In this thesis a similar mechanism was devised where a given task would lock the ready-queue after a pre-specified time length if it still has remaining workload to execute. This prevents it from being interfered upon by any task which was released after this particular time instant. The provided methodology to assign the ready-queue locking interval is not optimal. For some task-sets currently deemed unschedulable there exists RQL_i assignments which would indeed schedule these. The Ready-queue locking mechanism was further integrated with the pre-emption threshold in order to further extend the schedulability and gain heighten reduction on the pre-emption count.

- **Pre-emption Delay Estimation for the Floating Non Pre-emptive Region Model:** The usage of the floating non-pre-emptive region model in the state of the art works so far only allowed for a reduction on the number of pre-emptions quantified at design time. Generally in the real-time literature the maximum pre-emption delay penalty existing in any execution point of a task is considered, conservatively, to be the penalty suffered for every pre-emption. Generally it is expected that the pre-emption cost will vary and in ideal cases the large pre-emption cost value will be significantly different from the more average pre-emption cost observable in through the tasks' code. In this thesis a method to model the variable pre-emption cost as a function of the task progression is presented. An algorithm taking this information as input is presented to derive a lesser pessimistic pre-emption delay for the floating non-pre-emptive region model. This algorithm is extended incorporating another function which models the cumulative number of intrinsic cache misses. Tasks most likely have many possible executions paths. Different paths might have different pre-emption delay values. In turn some of the paths which are not part of the critical path can have their slack to the critical path exploited when considering the pre-emption delay of the task. Paths with high associated pre-emption delay penalties and large intrinsic cache misses count can have their function reduced taking into account the slack of these path to the critical one. By considering these altered function a still safe pre-emption delay estimation is obtained with a most of the times significant reduction on the analysis pessimism.

- **Ensuring Temporal Isolation in Platforms with Non-negligible Pre-emption Overheads:** Temporal isolation is more often than not overlooked in the real-time theory. This is partly due to the fact that for a large part there is ample support for this guarantee in platforms where the pre-emption delay is inexistent or largely negligible. Another aspect of this is that generally works that take pre-emption delay into consideration are solely trying to address this issue and its quantification in particular. When other issues are the subject of study the pre-emption delay is generally absent from the system model. This thesis proves that temporal isolation to in some cases is no longer ensured in platforms where the pre-emption delay exists in a significant form. It is nevertheless proven as well that the proposed mechanisms allow for generally lower resource expenditure – in terms of allocated budgets – when only the declared execution times of tasks cannot be trusted upon.

- **Derivation of the Limited Pre-emptive Theory for Global Multiprocessor Scheduling:**

  The state of the art in for limited pre-emptive was mainly focused on single core scheduling. Some works existed regarding non-pre-emptive scheduling of tasks in multiprocessor global scheduling. This thesis nevertheless constitutes the first derivation of the limited pre-emptive theory for global multiprocessor scheduling algorithms. On the derivation of the fixed non-pre-emptive region model two possible protocols were investigated and compared against. The floating non-pre-emptive region model theory is itself addressed as well. Even though most of the work was carried out assuming fixed global task priority schedulers it is shown that the limited pre-emptive scheduling mechanism allows for a schedulability increase in the GEDF case. Experimental results were obtained in order to assess whether the same patterns observed for single core would translate into global scheduling with respect to schedulability and overall estimation of pre-emptions.

Thesis– **Limited pre-emptive schedulers allow for significantly reducing pessimism in the pre-emption delay accounting in single-core and for enhanced schedulability when employing global fixed task priority schedulers**

Throughout this document, apart from the accessory section on temporal isolation, this thesis has been proven in a specific theoretical model. Real platforms are much more complex than the models employed. It is plausible that this theoretical advantage is not directly translatable into system design as some mechanism overheads may degrade the performance of the limited pre-emptive framework construction. For sake of completeness it is worth noting that, in the theoretical model average case behaviour, the limited pre-emptive schedule may be subject to higher pre-emption delay though this can never be the case in the worst-case estimations.

7.1 Future Directions

Single-core Run-time Proofs-of-concept: Similarly to many other works in the real-time community, this thesis entails a more theoretical approach to the previously discussed subjects. Real-time as a study area exists with the purpose of providing practitioners a relevant guidance in their systems building problems, particularly in what concerns to the timing issues. The work carried out in the course of this thesis has itself the intent of being practically viable. As such it is worth to trial whether the proposed run-time extensions to the floating non-pre-emptive region scheduling has a real benefit when comparing the implementation overhead of the policy. The additional resources spent taking the decisions may lead to inefficiencies in the system and lead to actual shorter executions from the pre-empted tasks. The mechanism itself also relies on a interrupt routine to be run which might itself – if care is not taken while implementing it – pollute the cache in a significant way and hence increase the pre-emption delay it is trying to mitigate. In par with this run-time extension although not to the same level the ready-queue locking mechanism should also be implemented as an actual real-time scheduling policy in order to assess its run-time overhead in comparison to single-core EDF as its main goal is to increase schedulability so that the system designer does not have to resort to single core EDF which is known to present a high run-time overhead for particular task-sets.

Limited Pre-emptive Model Pre-emption Delay Computation: Every literature on the limited pre-emptive model whether quantifying the number of pre-emptions or quantifying the actual pre-emption delay does it so per task. The outcome of every work – this thesis is no exception – is a bound on the number of times each task can be pre-empted. In certain scenarios the overall bounds for the pre-emption delay in the entire schedule obtained might be more pessimistic than the outcome of the trivial fully pre-emptive mechanisms. This scenario can manifest itself when the length of the non-pre-emptive regions is small when compared to the total length of the task. The theory that exploits the additional knowledge provided by the limited pre-emptive theory and derives tight schedule-wise pre-emption estimations demands further study. Thus far it not clear which form this solution would take even from a high level perspective. In the case of the fully pre-emptive scenario it is now well accepted that quantifying the pre-emptions form the point of view of the pre-empting tasks rather than a per task bound leads to much better results as many pre-emptions are not multiply accounted. This is likely to be the same case in the limited pre-emptive methodology.

Temporal Isolation in Multicores: The work presented in this thesis pertaining to the temporal isolation has been derived considering purely single core scenarios. Most of the works derived for single-core platforms can generally be directly ported to multicore devices provided that fully partitioned scheduling is employed. This tenet is not applicable to the temporal isolation property since the interference that concurrent tasks – executing on distinct processors – reciprocally induce, occurs on shared architectural devices which are different in nature from the ones existing in the singlecore. In a multicore platform not only the tasks would share the usage of architectural features such as the cache, these would also access memory controllers and other mechanisms alike. Even if some trivial solutions could be devised whether for the fully partitioned and for the global cases with respect to the memory controller concurrent access – for instance setting up budgets for the memory access – these lead to heavy resource usage. These solutions do not mitigate the contention in the shared resources but rather just police it. It is common for the schedulability assuming the worst-case scenario to become inferior when compared to single-core execution. Further work is then required to ensure that workload can indeed execute efficiently in a multi-processor platform while still ensuring a sufficient degree of separation between workloads running concurrently on different cores.

Multiprocessor Limited Pre-emptive Practice: As for the single processor works presented in this thesis, the limited pre-emptive work suffers from lack of a real platform implementation which enables its fitness as a scheduling solution to be tested. Moreover, it is widely accepted that the nature of the global scheduling algorithms considered (GFTP and GEDF) is such that very little information about the worst-case run-time events is known at design time (i.e. number of pre-emptions and migrations). In this way the limited pre-emptive theory helps the system designer in that it allows for a considerable more amount of information to be used in the temporal analysis. Nevertheless more effort should be taken in order to assess whether the scheduling policies could be slightly changed in order to tight the maximum considered number of migrations. As of the current state of the work all pre-emptions are considered to induce a migration. The presented theory also lacks an integration of the bus contention which tasks are faced with. The limited pre-emptive model, namely the fixed non-pre-emptive kind, may prove a valuable framework in which bus requests are clamped together in order to provide more determinism on their temporal occurrence and hence lending more information for the intra-core interference arising from the system execution.

References

[AB04] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Journal on Real-Time Systems*, 27, 2004.

[AB09] Sebastian Altmeyer and Claire Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*, 2009.

[AB11] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 2011.

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13(1), 2009.

[AMR10] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the crpd bound for set-associative caches. In *LCTES*, 2010.

[Bak03] Theodore Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *RTSS*, 2003.

[Bar05] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *ECRTS*, 2005.

[Bar07] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, 2007.

[BB04a] Enrico Bini and Giorgio Buttazzo. Biasing effects in schedulability measures. In *ECRTS*, 2004.

[BB04b] Enrico Bini and Giorgio Buttazzo. Schedulability analysis of periodic fixed priority systems. *Transactions on Computers*, 2004.

[BB10] M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *Transactions on Industrial Informatics*, 2010.

[BBA10] Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *OSPERT 2010*, Jun 2010.

[BBM$^+$10] M. Bertogna, G. Buttazzo, M. Marinoni, Gang Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *ECRTS*, 2010.

[BCSM08] B.D. Bui, M. Caccamo, Lui Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA*, 2008.

[BDM09] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 2009.

[Bet10] Adam Betts. *Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs*. PhD thesis, University of York, 2010.

[BLV07a] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS*, 2007.

[BLV07b] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS*, 2007.

[BLV09] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Journal of Real-Time Systems*, 2009.

[BMSO$^+$96] J.V. Busquets-Mataix, J.J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *RTSS*, 1996.

[Bur95] Alan Burns. Preemptive priority-based scheduling: an appropriate engineering approach. In *Advances in real-time systems*. 1995.

[BXM$^+$11] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *ECRTS*, 2011.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *SIGACT-SIGPLAN*, 1977.

[CP01] A. Colin and I. Puaut. Worst-case execution time analysis of the rtems real-time operating system. In *ECRTS*, 2001.

[DB09] Robert I. Davis and Alan Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *RTSS*, 2009.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Survey*, 43(4):35:1–35:44, Oct 2011.

[DBM$^+$13] Rob Davis, Alan Burns, José Marinho, Vincent Nélis, Stefan M. Petters, and Marko Bertogna. Global fixed priority scheduling with deferred pre-emption. *RTCSA*, 2013.

[DD10] M. Destelle and J.-L. Dufour. Deterministic scheduling reconciles cache with preemption for wcet estimation. In *ECRTS*, 2010.

[Dev06] UmaMaheswari Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[DFP01] Radu Dobrin, Gerhard Fohler, and Peter Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. *RTSS*, 2001.

[DL97] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, 1997.

[DTB93] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *RTSS*, 1993.

[EEKS06] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. *Operating Systems Review*, 2006.

[ESD10] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATER*, 2010.

[GMR00] Laurent Georges, Paul Muhlethaler, and Nicolas Rivierre. A few results on non-preemptive real time scheduling. Research Report RR-3926, INRIA, 2000.

[GSYY09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *RTSS*, 2009.

[JCR07] Lei Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE*, 2007.

[KBL10] U. Keskin, R.J. Bril, and J.J. Lukkien. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *ETFA*, 2010.

[Lam] W. Lamie. Preemption threshold. Technical report. Available online: http://rtos.com/articles/18833.

[LB05] C. Lin and S.A. Brandt. Improving soft real-time performance through better slack reclaiming. In *RTSS*, 2005.

[LHS$^+$98] 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. *Transactions on Computers*, 1998.

[LKPB06] Caixue Lin, Tim Kaldewey, Anna Povzner, and Scott A. Brandt. Diverse soft real-time processing in an integrated system. In *RTSS*, 2006.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[LLH$^+$01] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding cache-related preemption delay for real-time systems. *Transactions on Software Engineering*, 2001.

[LRT92] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *RTSS*, 1992.

[LSD89a] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, 1989.

[LSD89b] John Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, 1989.

[McK04] Sally A. McKee. Reflections on the memory wall. In *CCF*, 2004.

[MNP$^+$13] José Marinho, Vincent Nélis, Stefan M. Petters, Marko Bertogna, and Rob Davis. Limited pre-emptive global fixed task priority. *RTSS*, 2013.

[MNP14] José Marinho, Vincent Nélis, and Stefan M. Petters. Temporal isolation with preemption delay accounting. *ETFA*, 2014.

[MNPP12a] José Marinho, Vincent Nélis, Stefan M. Petters, and Isabelle Puaut. Preemption delay analysis for floating non-preemptive region scheduling. In *DATE*, 2012.

[MNPP12b] José Marinho, Vincent Nélis, Stefan Markus Petters, and Isabelle Puaut. An improved preemption delay upper bound for floating non-preemptive region. In *SIES*, 2012.

[MP10] José Marinho and Stefan Markus Petters. Runtime crpd management for rate-based scheduling. In *WARM, CPSWeek*, 2010.

[MP11] J. Marinho and Stefan M. Petters. Job phasing aware preemption deferral. In *EUC*, 2011.

[MPB12] José Manuel Marinho, Stefan M. Petters, and Marko Bertogna. Extending fixed task-priority schedulability by interference limitation. In *RTNS*, 2012.

[MRNP11] José Marinho, Gurulingesh Raravi, Vincent Nélis, and Stefan Markus Petters. Partitioned scheduling of multimode systems on multiprocessor platforms: when to do the mode transition? In *RTSOPS*, 2011.

[NAMP11] Vincent Nélis, Bjorn Andersson, José Marinho, and Stefan Markus Petters. Global-edf scheduling of multimode real-time systems considering mode independent tasks. In *ECRTS*, 2011.

[NCS$^+$06] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *WCET*, 2006.

[NMR03] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.

[NP08] L. Nogueira and L.M. Pinho. Shared resources and precedence constraints with capacity sharing and stealing. In *IPDPS*, 2008.

[OY98] Sung-Heun Oh and Seung-Min Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *RTCSA*, 1998.

[PF01] Stefan M. Petters and Georg Färber. Scheduling analysis with respect to hardware related preemption delay. In *WCET*, 2001.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems Journal*, 2007.

[RM06a] Harini Ramaprasad and Frank Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *RTAS*, 2006.

[RM06b] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemption points. In *RTSS*, 2006.

[SE04] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT*, 2004.

[SKKC00] Youngsoo Shin, Daehong Kim, Daehong Kimü, and Kiyoung Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems, 2000.

[SSE05] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, 2005.

[SSL89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal on Real-Time Systems*, 1989.

[SW00] M. Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *RTSS*, 2000.

[TM07] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *Transactions on Embedded Computing Systems*, 2007.

[WA12] Jack Whitham and Neil Audsley. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *RTAS*, 2012.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *Transactions on Embedded Computing Systems*, 2008.

[Win] Wind River. VxWorks Platforms. http://www.windriver.com/products/product-notes/PN_VE_6_9_Platform_0311.pdf.

[WS99] Yun Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *RTCSA*, 1999.

[YBB09] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. *RTCSA*, 2009.

[YBB10] Gang Yao, G. Buttazzo, and M. Bertogna. Comparative evaluation of limited preemptive methods. In *ETFA*, 2010.

[YBB11a] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Journal Real-Time Systems*, 2011.

[YBB11b] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Journal of Real-Time Systems*, 2011.