Ordonnancement de tâches efficace et à complexité maîtrisée pour des systèmes temps-réel

F. Muhammad

UNIVERSITE DE NICE SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention: Informatique

présentée et soutenue par

Farooq MUHAMMAD

Ordonnancement de Tâches Efficace et à Complexité Maîtrisée pour des Systèmes Temps Réel

Thèse dirigée par *Michel AUGUIN*

soutenue le *9 Avril 2009*

Jury:

M.	Pascal Richard	Professeur Université de Poitiers	Rapporteur
M.	Yvon Trinquet	Professeur Université de Nantes	Rapporteur
M.	Joël Goossens	Professeur Université Libre de Bruxelles	Examinateur
M.	Robert de Simone	Directeur de Recherche INRIA	Président du Jury
M.	Michel Auguin	Directeur de Recherche CNRS	Directeur de Thèse
M.	Fabrice Muller	Maître de Conférence Université de Nice Sophia Antipolis	Co-Directeur de Thèse

“Hard work never killed anybody, but why take a chance?.”

Edgar Bergen

Abstract

by Muhammad Farooq

The performance of scheduling algorithm influences the performance of the whole system. Real time scheduling algorithms have theoretical optimal schedulable bounds, but these optimal bounds are achieved at the cost of increased scheduling events (preemptions and migrations of tasks) and high run time complexity of algorithms. We believe that by exploiting parameters of tasks, these algorithms can be made more efficient and cost conscious to increase Quality of Service (QoS) of application.

In first part of thesis, we propose mono-processor scheduling algorithms which increase quality of service of hybrid application by maximizing execution of soft real time tasks, and by providing guarantees to hard real time tasks even in overload situations. Scheduling cost of algorithms is also reduced (in terms of reduced number of preemptions) by taking into account all explicit and implicit parameters of tasks. Reducing scheduling overheads not only increases performance of the scheduler but also minimizes energy consumption of the system. That's why, we propose to devise a technique embedded with existing DVFS (dynamic voltage and frequency scaling) techniques to minimize the switching points, as switching from one frequency to another steals processor cycles and consumes energy of system.

Multiprocessor scheduling algorithms based on fluid scheduling model (notion of fairness), achieve optimal schedulable bounds; but fairness is guaranteed at the cost of unrealistic assumptions, and by increasing preemptions and migrations of tasks to a great extent. An algorithm (ASEDZL) is proposed in this dissertation, which is not based on fluid scheduling model. It not only minimizes preemptions and migrations of tasks but relaxes the assumptions also due to not being bases on fairness notion. Moreover, ASEDZL is also propose to schedule tasks in hierarchical approach, and it gives better results than other approaches.

. . .

Résumé

par Muhammad Farooq

Les performances des algorithms d'ordonnancement ont un impact direct sur les performances du système complet. Les algorithmes d'ordonnancement temps réel possèdent des bornes théoriques d'ordonnanabilité optimales mais cette optimalité est souvent atteinte au prix d'un nombre élevé d'événements d'ordonnancement à considérer (préemptions et migrations de tâches) et d'une complexité algorithmique importante. Notre opinion est qu'en exploitant plus efficacement les paramètres des tâches il est possible de rendre ces algorithmes plus efficaces et à coût maitrisé, et ce dans le but d'améliorer la Qualité de Service (QoS) des applications. Nous proposons dans un premier temps des algorithmes d'ordonnancement monoprocesseur qui augmentent la qualité de service d'applications hybrides c'est-à-dire qu'en situation de surcharge, les tâches à contraintes souples ont leur exécution maximisée et les échéances des tâches à contraintes strictes sont garanties. Le coût d'ordonnancement de ces algorithmes est aussi réduit (nombre de préemptions) par une meilleure exploitation des paramètres implicites et explicites des tâches. Cette réduction est bénéfique non seulement pour les performances du système mais elle agit aussi positivement sur la consommation d'énergie. Aussi nous proposons une technique associée à celle de DVFS (dynamic voltage and frequency scaling) afin de minimiser le nombre de changements de points de fonctionnement du fait qu'un changement de fréquence implique un temps d'inactivité du processeur et une consommation d'énergie.

Les algorithmes d'ordonnancement multiprocesseur basés sur le modèle d'ordonnancement fluide (notion d'équité) atteignent des bornes d'ordonnanabilité optimales. Cependant cette équité n'est garantie qu'au prix d'hypothèses irréalistes en pratique du fait des nombres très élevés de préemptions et de migrations de tâches qu'ils induisent. Dans cette thèse un algorithme est proposé (ASEDZL) qui n'est pas basé sur le modèle d'ordonnancement fluide. Il permet non seulement de réduire les préemptions et les migrations de tâches mais aussi de relâcher les hypothèses imposées par ce modèle d'ordonnancement. Enfin nous proposons d'utiliser ASEDZL dans une approche d'ordonnancement hiérarchique ce qui permet d'obtenir de meilleurs résultats que les techniques classiques.

. . .

Acknowledgements

A journey is easier when we travel together. Interdependence is certainly more valuable than independence. My entry to graduate school and successful completion of this dissertation and the Ph.D. program are due to the support and help of several people. The following is my attempt at acknowledging everyone I am indebted to.

I am profoundly grateful to my advisors, Fabrice Muller and Michel Auguin, for educating and guiding me over the past few years with great care, enthusiasm, and patience. I am thankful to Michel Auguin for making me consider doing a Ph.D. and taking me under his fold when I decided to go for it. Ever since, it has been an extreme pleasure and a privilege working for Michel and Fabrice and learning from them. They reposed a lot of confidence in me, which, I should confess, was at times overwhelming, and gave me enormous freedom in my work, all the while ensuring that I was making adequate progress. They helped relieve much of the tedium, assuage my apprehensions, boost my self-esteem, and make the whole endeavor a joy by being readily accessible, letting me have their undivided attention most of the time I walked in to their offices, offering sound and timely advice, and when needed, suggesting corrective measures.

I feel honored to have had respected researchers also take the time to serve on my committee. In this regard, thanks are due to Joël Goossens, Pascal Richard, Yvon Trinquet and Robert De Simone. I am thankful to my entire committee for their feedback on my work and their flexibility in accommodating my requests while scheduling proposals and exams.

In addition, I would like to thank all my colleagues, over the past few years who have helped me become a better researcher. The list would definitely include the following people: Sebastien Icart, Khurram Bhatti, Belaid Ikbel and Zeeshan. Special thanks goes to Sebastien who helped me a lot to improve my french language skills, and special thanks also goes to Belaid ikbel, Khurram Bhatti and Bensaid Siouar whom I spent many enjoyable lunches, dinners and specially café au lait breaks, where we talk about everything other than research stuff. These wonderful breaks really helped me to kick back, relax, cool down a little and gather my energy for hard and tedious work of formal proofs of algorithms. At these times, we used to recharge our spent-up batteries.

I would like to thank to all my friends Najam, Waseem, Naveed, Umer, Khurram, Uzair, Usman, Zeeshan and Rizwaan whose company has always been a great source of encouragement for me. Being in company of such friends, I have enjoyed a lot the period of my thesis. I never felt that I was away from the family. My friends gave me the feelings of being at home.

I would like to give my special thanks to my family whose patient love and prays enabled me to complete this work. I feel a deep sense of gratitude for my father and mother who formed part of my vision and taught me the good things that really matter in life. I would like to express my gratitude to my aunt as well whose guidance, love and prayers had been like a source of light in my life.

I am fortunate to have been blessed with a loving and supportive family, who repose great trust in me. I owe it to my mother and my aunt for instilling in me a passion for learning, and to my father for his pragmatism and for enlivening even mundane things through his wit and sense of humor. I am thankful to my sisters and to my brothers for their affection and prayers. Above all, I am indebted in no small measure to my wife for having endured a lot during the past two years with no complaints. She put up with separation for several months. But for her cooperation, patience, love, and faith, I would not have been able to continue with the Ph.D. program, let alone complete it successfully. I owe almost everything to her and hope to be able to repay her in full in the coming years. Finally, I am thankful to God Almighty for the turn of events that led to this most valuable and rewarding phase of my life. . . .

Contents

List of Figures

List of Tables

Abbreviations

EDF	**E**arliest **D**eadline **F**irst
EEDF	**E**nhanced **E**arliest **D**eadline **F**irst
RM	**R**ate **M**onotonic
ERM	**E**nhanced **R**ate **M**onotonic
LLF	**L**east **L**axity **F**irst
CASH	**CA**pacity **SH**aring
MUF	**M**aximum **U**rgency **F**irst
QRAM	**Q**oS **R**esource **A**llocation **M**odel
Pfair	**P**roportionate **F**air
LLREF	**L**east **L**ocal **R**emaining **E**execution **F**irst
EDZL	**E**arliest **D**eadline until **Z**ero **L**axity
ASEDZL	**A**nticipating **S**lack **E**arliest **D**eadline until **Z**ero **L**axity
DVFS	**D**ynamic **V**oltage and **F**requency **S**caling
EDVFS	**E**nhanced **D**ynamic **V**oltage and **F**requency **S**caling
NPZ	**N**o **P**reemption **Z**one

Symbols

τ	set of tasks
τ_{par}	set of partitioned tasks
τ_g	set of global tasks
τ^i	set of tasks partitioned on processor Π_i
$\tau_r(t)$	represent set of ready task at time t
HP	hyper period of n tasks
T_i	task T_i
C_i	worst case execution time of task T_i
$C_i^{rem}(t)$	remaining execution time of T_i at time t
B_i	best case execution time of T_i
P_i	period of T_i
d_i	absolute deadline of task T_i
$d_i^{rem}(t)$	remaining time to deadline of T_i
μ_i	utilization or weight of task T_i
$U(\tau)$	sum of utilizations of all tasks
Π	set of identical processors in an architecture
Π_j	j^{th} processor
R_k	k^{th} release instant in a schedule
R_k^j	local release instant on processor Π_j
T_x^j	supertask on processor Π_j

I dedicate my dissertation to my mother, aunt and to my beloved sisters and brothers. . . .

Chapter 1

Introduction

1.1 Background

The distinguishing characteristic of a real-time system in comparison to a non-real-time system is the inclusion of timing requirements in its specification. That is, the correctness of a real-time system depends not only on logically correct segments of code that produce logically correct results, but also on executing the code segments and producing correct results within specific time frames. Thus, a real-time system is often said to possess dual notions of correctness, logical and temporal. Process control systems, which multiplex several control-law computations, radar signal-processing and tracking systems, and air traffic control systems are some examples of real-time systems.

Timing requirements and constraints in real-time systems are commonly specified as deadlines within which activities should complete execution. Consider a radar tracking system as an example. To track targets of interest, the radar system performs the following high-level activities or tasks: sends radio pulses towards the targets, receives and processes the echo signals returned to determine the position and velocity of the objects or sources that reflected the pulses, and finally, associates the sources with targets and updates their trajectories. For effective tracking, each of the above tasks should be invoked repeatedly at a frequency that depends on the distance, velocity, and the importance of the targets, and each invocation should complete execution within a specified time or deadline.

Another characteristic of a real-time system is that it should be predictable. Predictability means that it should be possible to show, demonstrate, or prove that requirements are always met subject to any assumptions made, such as on workloads.

Based on the cost of failure associated with not meeting them, timing constraints in real-time systems can be classified broadly as either hard or soft. A hard real-time constraint is one whose violation can lead to disastrous consequences such as loss of life or a significant loss to property. Industrial process-control systems and robots, controllers for automotive systems, and air-traffic controllers are some examples of systems with hard real-time constraints. In contrast, a soft real-time constraint is less critical; hence, soft real-time constraints can be violated. However, such violations are not desirable, either, as they may lead to degraded quality of service, and it is often the case that the extent of violation be bounded. Multimedia systems and virtual-reality systems are some examples of systems with soft real-time constraints.

There are several emerging real-time applications that are very complex and have high computational requirements. Examples of such systems include automatic tracking systems and tele-presence systems. These applications have timing constraints that are used to ensure high system fidelity and responsiveness, and may also be crucial for correctness in certain applications such as tele-surgery. Also, their processing requirements may easily exceed the capacity of a single processor, and a multiprocessor may be necessary to achieve an effective implementation. In addition, multiprocessors are, generally, more cost-effective than a single processor of the same capacity because the cost (monetary) of a k-processor system is significantly less than that of a processor that is k times as fast (if a processor of that speed is indeed available).

The above observations clearly underscore the growing importance of scheduling algorithms in real-time systems. In this dissertation, we focus on several fundamental issues pertaining to the scheduling of real-time tasks on mono-processor as well as on multiprocessor architecture. Before discussing the contributions of this dissertation in more detail, we briefly describe some basic concepts pertaining to real-time systems.

1.2 Background on Real Time Systems

A real-time system is typically composed of several (sequential) processes with timing constraints. We refer to these processes as tasks and set of these tasks is represented by $\tau = \{T_1, T_2, ..., T_n\}$. In most real-time systems, tasks are recurrent, i.e., each task is invoked repeatedly. The periodic task model of Liu and Layland [48] provides the simplest notion of a recurrent task. Each periodic task T_i is characterized by a phase ϕ_i, a period P_i, a relative deadline d_i, and an execution requirement C_i ($C_i < d_i$). The best case execution time is represented by B_i. Such a task is invoked every P_i time units, with its first invocation occurring at time ϕ_i. We refer to each invocation of a task as a job/instance, and the corresponding time of invocation as the job's release time. Thus, the relative deadline parameter is used to specify the timing constraints of the jobs of a periodic task. Unless stated otherwise, we assume that relative deadline of a periodic task equals its period. In other words, each job must complete before the release of the next job of the same task. We define few run time parameters of a task such as $C_i^{rem}(t)$ which represents the remaining execution time of task T_i at time t defined as $C_i^{rem}(t) = C_i - C_i^{completed}(t)$ where $C_i^{completed}(t)$ represents the completed fraction of task T_i until time t. The remaining best case execution time at time t is given by $B_i^{rem}(t)$

The remaining time to deadline of a task T_i is defined as $d_i^{rem}(t) = d_i - t$. A dynamic parameter of tasks T_i is AET_i which represents the actual execution time of task T_i.

Each task T_i has its own laxity L_i defined as:

$$L_i = P_i - C_i$$

and laxity at time t is defined as:

$$L_i(t) = d_i^{rem}(t) - C_i^{rem}(t)$$

A periodic task system in which all tasks have a phase of zero is called a synchronous periodic task system, and we have considered synchronous task system in this dissertation unless stated otherwise. Let HP represents the hyper period of all tasks which is the least common multiple of periods of all n tasks. The weight or utilization of a task T_i, denoted μ_i, is the ratio of its execution requirement to its period. We use the terms weight and utilization interchangeably in this dissertation. $\mu_i = C_i/P_i$. The weight (or utilization) of a task system is the sum of the weights of all tasks in the system. Offloading factor of a task T_i, denoted by O_i, represents the percentage of a processor that can be used to execute tasks other than T_i, and O_i is the ratio of task's laxity to its period (i.e., L_i/P_i).

In this dissertation, we assume that all tasks are synchronous and preemptive, i.e., a task can be interrupted during its execution and resumed later from the same point. Unless stated otherwise, we assume that the overhead of a preemption is zero. We further assume that all tasks are independent, i.e., the execution of a task is not affected by the execution of other tasks. In particular, we assume that tasks do not share any resources other than the processor, and that they do not self-suspend during execution.

1.3 Real-Time Scheduling Strategies and Classification

In general, real time scheduling algorithm assigns a priority to each job, and on an M-processor system, schedules for execution the M jobs with the highest priorities at any instant.

1.3.1 Feasibility and optimality

A periodic task system τ is feasible on processing platform if and only if for every possible real-time instance there exists a way to meet all deadlines. A feasibility test for a class of task systems is specified by giving a condition that is necessary and sufficient to ensure that any task system in that class is feasible.

The algorithm that is used to schedule tasks (i.e., allocate processor time to tasks) is referred to as a scheduling algorithm. A task system τ is said to be schedulable by algorithm A if A can guarantee the deadlines of all jobs of every task in τ. A condition under which all task systems within a class of task systems are schedulable by A is referred to as a schedulability test for A for that class of task systems. A scheduling algorithm is defined as optimal for a class of task systems if its schedulability condition is identical to the feasibility condition for that class.

1.3.2 On-line versus offline scheduling

In offline scheduling, the entire schedule for a task system (up to a certain time such as the the least common multiple (LCM) of all task periods) is pre-computed before the system actually runs; the actual run-time scheduling is done using a table based on this pre-computed schedule. On the other hand, an on-line scheduler selects a job for scheduling without any knowledge of future job releases. (Note that an on-line scheduling algorithm can also be used to produce an offline schedule.) Clearly, offline scheduling is more efficient at run-time; however, this efficiency comes at the cost of flexibility. In order to produce an off-line schedule, it is necessary to know the exact release times for all jobs in the system. However, such knowledge may not be available in many systems, in particular, those consisting of sporadic tasks, or periodic tasks with unknown phases. Even if such knowledge is available, then it may be impractical to store the entire precomputed schedule (e.g., if the LCM of the task periods is very large). On the other hand, on-line schedulers need to be very efficient, and hence, may need to make sub-optimal scheduling decisions, resulting in schedulability loss.

1.3.3 Static versus dynamic priorities

Most scheduling algorithms are priority-based: they assign priorities to the tasks or jobs in the system and these priorities are used to select a job for execution whenever scheduling decisions are made. A priority-based scheduling algorithm can determine task or job priorities in different ways.

A scheduling algorithm is called a static-priority algorithm if there is a unique priority associated with each task, and all jobs generated by a task have the priority associated with that task. Thus, if task T_i has higher priority than task T_j, then whenever both have active jobs, T_i's job has higher priority than T_j's job. An example of a scheduling algorithm that uses static priorities is the rate-monotonic (RM) algorithm [48]. The RM algorithm assigns higher priority to tasks with shorter periods.

Dynamic-priority algorithms allow more flexibility in priority assignments; a task's priority may vary across jobs or even within a job. An example of a scheduling algorithm that uses dynamic priorities is the earliest-deadline-first (EDF) algorithm [48]. EDF assigns higher priority to jobs with earlier deadlines, and has been shown to be optimal for scheduling periodic and sporadic tasks on uniprocessors [48, 54]. The least laxity- first

(LLF) algorithm [54] is also an example of a dynamic-priority algorithm that is optimal on uniprocessors. As its name suggests, under LLF, jobs with lower laxity are assigned higher priority.

1.4 Real-time Scheduling on Multiprocessors

In this subsection, we consider multiprocessor scheduling in some details.

1.4.1 Multiprocessor Scheduling Approaches

Scheduling of tasks on multiprocessor systems is typically solved using two different methods based on how tasks are assigned to the processors at run-time, namely partitioning and non-partitioning. In the partitioning-based method, all instances of a task are executed on the same processor, which is determined before run-time by a partitioning algorithm. In a non partitioning method, any instance of a task can be executed on a different processor, or even be preempted and moved to a different processor, before it is completed.

1.4.1.1 Partitioning

Under partitioning, the set of tasks is statically partitioned among processors, that is, each task is assigned to a unique processor upon which all its jobs execute. Each processor is associated with a separate instance of a uniprocessor scheduler for scheduling the tasks assigned to it and a separate local ready queue for storing its ready jobs. In other words, the priority space associated with each processor is local to it. The different per-processor schedulers may all be based on the same scheduling algorithm or use different ones. The algorithm that partitions the tasks among processors should ensure that for each processor, the sum of the utilizations of tasks assigned to it is at the most utilization bound of its scheduler.

Partitioning method have several advantages over non partitioning methods [4, 46, 58, 59, 71]. Firstly, the scheduling overhead associated with a partitioning method is lower than the overhead associated with a non-partitioning method. Secondly, partitioning methods allow us to apply well-known uniprocessor scheduling algorithms on each processor. Thirdly and most importantly, each processor could use resources dedicated to it i.e., distributed memory etc. The last mentioned aspect improves the performance of the system a lot, and in most cases it proves the partitioning method to be better than its counterpart. Optimal assignment of tasks to processors is known to be NP-hard, which is major drawback of partitioning method.

1.4.1.2 Global scheduling

In contrast to partitioning, under global scheduling, a single, system-wide, priority space is considered, and a global ready queue is used for storing ready jobs. At any instant, at most M ready jobs with the highest priority (in the global priority space) execute on the M processors. No restrictions are imposed on where a task may execute; not only can different jobs of a task execute on different processors, but a given job can execute on different processors at different times.

In contrast, the non-partitioning method has received much less attention [44, 49, 51, 52], mainly because it is believed to suffer from scheduling and implementation-related shortcomings, also because it lacks support for more advanced system models, such as the management of shared resources. The other important factor that makes this approach of less interest is the use of a shared memory which introduces the bottleneck for system scalability. On the better side, schedulability bounds are much better than its counterpart.

1.4.1.3 Two-level hybrid scheduling

Some algorithms do not strictly fall under either of the above two categories, but have elements of both. For example, algorithms for scheduling systems in which some tasks cannot migrate and have to be bound to a particular processor, while others can migrate, follow a mixed strategy. In general, scheduling under a mixed strategy is at two levels: at the first level, a single, global scheduler determines the processor that each job should be assigned to using global rules, while at the second level, the jobs assigned to individual processors are scheduled by per-processor schedulers using local priorities. Several variants of this general model and other types of hybrid scheduling are also possible. Typically, the global scheduler is associated with a global queue of ready, but unassigned jobs, and the per-processor schedulers with queues that are local to them.

1.5 Low-Power Embedded Operating Systems

The tremendous increase in demand for many battery-operated computing devices evidences the need for power-aware computing. As a new dimension of CPU computing, the goal of power-aware CPU scheduling is to dynamically adjust hardware to adapt to the expected performance of the current workload so that a system can efficiently save power, lengthening its battery life for useful work in the future. In addition to traditional $low-power$ designs for the highest performance delivery, the new concept focuses on enabling hardware-software collaboration to scale down power and performance in hardware whenever the system performance can be relaxed.

Power-saving State Control (PSC) and Dynamic Voltage and Frequency Scaling (DVFS) are promising examples of power-aware techniques developed in hardware. Power-saving states are like operating knobs of a device, as they consume much less power and support only partial functionalities compared to the regular active operating mode. With the prevailing power-saving trend, power states have been ubiquitously supported in processors, disks, RDRAM memory chips, wireless network cards, LCD displays, etc. Common power-saving states include standby (clock-gating), retention (clock-gating with just enough reduced supply voltage to save the logic contents of circuits), and power-down (power-gating) modes. Some devices provide specific power-states that offer more energy-efficient operating options such as the control of the back light in LCD displays and the control of the modulation scheme and transmission rate in wireless network cards.

DVFS techniques are deployed in many commercial processors such as Transmeta's Crusoe, Intel's XScale processors and Texas Instruments OMAP3430. Due to the fact that dynamic power in CMOS circuits has a quadratic dependency on the supply voltage, lowering the supply voltage is an effective way to reduce power. However, this voltage reduction also adversely affects the system performance through increasing delay. Therefore, efficient DVFS algorithms must maintain the performance delivery required by applications.

No matter how advanced power-aware strategies in circuit designs and hardware drivers may become, they must be integrated with applications and the operating systems, and knowledge of the applications intents is essential. Advanced Configuration and Power Interface (ACPI) [23] is an open industry specification that establishes interfaces for the operating system (OS) to directly configure power states and supply voltages on each individual device. ACPI was developed by Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba. The operating system thus can customize energy policies to maintain the quality of service required by applications and assign proper operating modes to each device at runtime.

1.5.1 DVFS

The field of dynamic voltage and frequency scaling (DVFS) is currently the focus of a great deal of power-aware research. This is due to the fact that the dynamic power consumption of CMOS circuits [28, 72] is given by:

$$PW = \beta C_L V_{DD}^2 f \tag{1.1}$$

where β is the average activity factor, C_L is the average load capacitance, V_{DD} is the supply voltage and f is the operating frequency. Since the power has a quadratic dependency on the supply voltage, scaling the voltage down is the most effective way to minimize energy. However, lowering the supply voltage can also adversely affect the system performance due to increasing delay. The maximum operating speed is a direct consequence of the supply voltage given by:

$$f = \frac{K(V_{DD} - V_{th})^{\beta}}{V_{DD}} \tag{1.2}$$

where K is a constant specific to a given technology, V_{th} is the threshold voltage and β is the velocity saturation index, $1 \leq \beta \leq 2$

DVFS algorithms that target real-time systems instead assume the knowledge of timing constraints of real-time tasks which are specified by users or application developers in $< C, P, d >$ tuple. Pillai and Shin [35] proposed a wide-ranging class of voltage scaling algorithms for real-time systems. In their static voltage scaling algorithm, instead of running tasks with different speeds, only one system frequency is determined and used. Their operating frequency is determined by equally scaling all tasks to complete the lowest-priority task as late as possible. This turns out to be pessimistic. Since the amount of preemption by high-priority tasks is not uniformly distributed when there are multiple task periods, a task can encounter less preemption relative to its own computation and save more energy if it completes earlier than its deadline.

Aydin et al. [10] proposed the optimal static voltage scaling algorithm using the solution in reward-based scheduling. Their approach assigns a single operating frequency for each task and focuses on the EDF scheduling policy. DVFS effectively minimizes the energy consumption but cost of voltage/frequency switch is high in some architectures. One example for these systems is the Compaq IPAQ, which needs at least 20 milliseconds to synchronize SDRAM timing after switching voltage (frequency). Voltage switch cost on ARM11 implemented in the IMX31 architecture of Freescale is about 4 milliseconds). For such systems, switching voltage at every point when task finishes its execution before its worst case execution time is unacceptable. In this dissertation, we focus on minimizing the voltage/frequency switching points but not at the cost of wasting unused processor cycles.

Until now, We have considered a task model where tasks can execute only on one processor at a time, but with the invent of new programming paradigms like MPI, open-MP and Snet we are able to execute more than one threads of a task in parallel on a multiprocessor or on multi-core (chip multiprocessing) systems even on FPGAs as well. The degree of parallelism at different levels of a task is different which must be taken care of while reserving resources for tasks. This can be tackled by allocating resources dynamically, but if architecture can improve its processing capabilities at run time to well suit the varying demands of application, then there is a need to schedule tasks by a self adaptive approach. Moreover, if demands of applications changes at run time, then we are left with no solution of scheduling such task but self adaptive scheduling technique.

1.6 Self Adaptive scheduling

Scheduling of adaptively parallel jobs for which the number of processors which can be used without waste changes during runtime can be achieved by scheduler where number of processors allocated to tasks are static. Parallelism at each level of task is not same and task can self-optimize itself during runtime as well, thus offering different degree of parallelism at different than calculated before. So, there is a need to schedule tasks dynamically and adaptively. Most prior work on task scheduling for multi-tasked jobs deals with nonadaptive scheduling, where the scheduler allots a fixed number of processors to the job for its entire lifetime. For jobs whose parallelism is unknown in advance and which may change during execution, this strategy may waste processor cycles, because a job with low parallelism may be allotted more processors than it can productively use. Moreover, in a multiprogrammed environment, nonadaptive scheduling may not allow a new job to start, because existing jobs may already be using most of the processors. With adaptive scheduling, the job scheduler can change the number of processors allotted to a job while the job is executing. Thus, new jobs can enter the system, because the job scheduler can simply recruit processors from the already executing jobs and allot them to the newly arrived tasks.

1.7 Contributions

The main thesis supported by this dissertation is the following.
Real time scheduling algorithms have schedulable bounds equal to the capacity of architecture (based on some assumptions) but can be made more efficient by minimize scheduling overheads to increase QoS (Quality of Service) of application, and this efficiency is attained by taking into account task's implicit run time parameters. Moreover, the schedulable bounds equal to capacity of architecture can be achieved by relaxing these assumptions.

In the following subsections, we describe the contributions of this dissertation in more detail. In addition to optimal schedulable bounds (which is clearly important), efficiency is essential in several of the emerging real-time applications to improve QoS of application and to minimize power consumption of the system. The objective is to propose scheduling algorithms, which have very low run time complexity in terms of algorithmic complexity and number of the invocations during execution of tasks i.e., reduced scheduling events. Moreover, algorithms, which increase the number of tasks executed for given time by exploiting runtime parameters of tasks, are also considered more efficient, as QoS of application is directly related with number of tasks executed over a given period. In this dissertation, we address the efficiency issues by introducing significant modifications to existing scheduling algorithms to minimize scheduling events, cost of one scheduling event and to improve QoS of application (briefly described in Section 1.7.1). We also address the issues related to power efficiency of architecture and provide schemes to minimize power consumption of the system without compromising on the schedulability bound (Section 1.7.2). In case of multiprocessor system, schedulable bounds are achieved by making unrealistic assumptions about application, for which runtime complexity, preemptions and migration of tasks are quiet high. We handle this issue of runtime complexity and preemptions (and migrations) on multiprocessor architecture, and devise technique which relaxes these assumptions and reduces preemptions to a great extent(Section 1.7.3). We also study hybrid scheduling algorithms, and present three approaches which have optimal schedulable bounds and reduced cost(section1.7.4).

1.7.1 RUF scheduling algorithm

If task set is composed of critical and non critical tasks, and overall load of all tasks is greater than 100%, then there is a need to devise a technique where all critical tasks are guaranteed to meet their deadlines, and execution of non critical tasks is maximized as well. Existing algorithms (EDF,MUF,CASH) either do not provides guarantees to critical tasks in transient overload situation, or minimize the execution of non critical tasks. We propose to exploit the run time parameters of tasks i.e., time slot of processor available at runtime, $C_i - AET_i$ which is the difference beteween actual and worst case execution of task, not only to provide guarantees to critical tasks but to maximize the execution of non critical tasks also. To handle this issue, a new scheduling algorithm called RUF (Real Urgency First) is proposed with objective to increase the QoS of application. We define an admission control mechanism for non critical tasks to maximize their execution but without compromising on deadline guarantees of critical tasks. This admission of non critical task at time t depends upon the run time parameters of all those critcal tasks which have finsihed their execution until t. We illustrate the prinicple through an example and demonstrate experimentally that proposed algorithm incurs less preemptions than all existing algorithms in terms of maximizing executions of critical tasks.

1.7.2 Power Efficiency

In case of mono processor system, real time scheduling algorithms have optimal schedulable bounds but they have either high run time complexity or incurs a lot of preemptions of tasks (increased scheduling events). These increased preemptions and high runtime complexity of scheduling algorithms not only decrease the schedulable bounds (practically), but also increase energy consumption of the system. These algorithms do not exploit all implicit and runtime parameters of a task to minimize these overheads and energy of the system . Laxity is an implicit parameter of task, which provides certain level of flexibility to scheduler. Scheduler can exploit this flexibility to relax certain rules of scheduling algorithm that can help to minimize scheduling overheads and preemptions of tasks. We propose to exploit this implicit parameter to define an algorithm which significantly reduces preemptions of tasks. Two variants of the approach are proposed to exploit run time parameters of tasks to minimize preemptions, one is static and the other is dynamic. Moreover, we also propose to exploit these parameter both in case of dynamic and static priority scheduling algorithms i.e., for EDF and RM.

We also observed that preemptions of tasks are directly related with load of processor and minimizing number of preemptions is beneficial to reduce the energy consumption of the system. However, DVFS algorithms minimize the frequency of processor to decrease power consumption which has its adverse effect on increasing the number of preemptions. We propose an algorithm, where frequency of the processor is decreased only at those instants when it does not have its impact on the number of preemptions. Moreover, it is also ensured that switching points of frequency (where frequency of processor is changed) are also minimized, as changing from one level of frequency to another level consumes energy and processor time.

1.7.3 ASEDZL scheduling Algorithm

In case of multiprocessor systems, scheduling algorithms have been proposed which are optimal but they are based on fluid schedule model or fairness notion. Algorithms based on notion of fairness, increase scheduling overheads (preemptions and scheduler invocations) to such a great extent that sometimes they are impractical. To minimize these scheduling overheads, we propose a scheduling algorithm which is not based on fluid schedule (or fairness notion).

We propose an algorithm called Anticipating Slack Earliest Deadline First until zero laxity(ASEDZL). According to this algorithm, tasks are not allocated processor time in proportion to their weights. Tasks are selected to execute between two consecutive task's release instants and tasks with the earliest deadlines are ensured to execute on all processors until next release instant. This proposed algorithm provides better results in terms of minimum number of preemptions and migrations of tasks for not being based on fluid scheduling model. We illustrate the principle through couple of examples, and we also demonstrate through simulation results that it performs better than all those algorithms which have schedulable bound equal to capacity of the architecture.

1.7.4 Hierarchical Scheduling Algorithms

Hierarchical scheduling or hybrid scheduling algorithms guarantees better results than partitioning or global scheduling algorithms in case of distributed shared memory architecture but very few hybrid scheduling algorithms are proposed which have schedulable bound equal to number of processors (with some assumptions, detailed in chapter 5). Moir et al. [53] proposed a technique of supertasking approach for hybrid scheduling algorithm, where Pfair scheduling algorithm is used as global scheduler, but it was required to define weight bound condition between local tasks (partitioned tasks) and its corresponding supertask to achieve optimal schedulable bound, we establish that condition. We also propose to use ASEDZL scheduling algorithm as global scheduler, as it does not impose any condition on weights of supertask and local tasks. We compare, analytically, this proposed algorithm with existing approaches to illustrate that it promises better results. A novel hybrid scheduling algorithm is also proposed, where time slots are reserved for execution of local and global tasks.

1.7.5 ÆTHER: Self-adaptive Middleware

Self-adaptive scheduling algorithms deal with tasks for which degree of parallelism is calculated statically, and resources are allocated dynamically to cope different degree of parallelism at different levels of sequential execution of a task to minimize wastage. These algorithms also deal with dynamic creation and deletion of tasks, but they don't deal where tasks can self-optimize at run time and due to this self-optimization, degree of parallelism at different levels of sequential execution of a task may differ from that of calculated statically. Moreover, architecture can also self-optimize to execute parallel threads on one resource more efficiently than executing each thread on separate resource. These run time optimizations of architecture and application requires an intelligent self-adaptive resource management mechanism that can tackle all these run time variations. We propose a self-adaptive scheduling model, which takes care of all these run time optimization of tasks and architecture to schedule tasks on resources, and provide deadline guarantees to real time tasks.

1.8 Organization

In Chapter 2, we exploit the runtime parameter of a task which is $C_i - AET_i$ to improve the QoS of an application, while in Chapter 3, we propose to work on both implicit and runtime parameters of a task which are its laxity and release times respectively to minimize the preemptions of tasks in schedule. In Chapter 4, we aim to take care of both explicit and implicit parameters of a task i.e., C_i, P_i and L_i, to propose an optimal global multiprocessor scheduling algorithm. Chapter 5 covers the proposed hierarchical scheduling algorithms, where we have presented results using Pfair and ASEDZL scheduling algorithm as a global scheduler. In each chapter, related state of art is also presented before describing our work and results. In the last chapter, we summarize our contributions and discuss directions for future research.

Introduction

1.1 Généralités

La différence principale entre un système temps réel par rapport à un système transactionnel ou même hautes performances est l'introduction de conditions temporelles dans ses spécifications. En d'autres termes, un système temps réel est correct non seulement si le code applicatif est logiquement correct et produit des résultats logiquement corrects, mais également si ces résultats sont produits dans des intervalles de temps précis. Ainsi, un système temps réel possède des notions duales d'exactitude relatives aux aspects logique et temporel. Les systèmes de contrôle de processus qui multiplexent plusieurs calculs de lois de commande tels du traitement de signal radar et de suivi de cibles dans un système de contrôle de trafic aérien est un exemple de systèmes temps réel.

Les conditions et les contraintes temporelles dans les systèmes temps réel sont généralement spécifiées sous forme d'échéances avant lesquelles les activités du système doivent terminer leurs exécutions. Considérons comme exemple un système de suivi radar. Pour suivre des cibles d'intérêt, le système radar effectue les activités ou tâches suivantes : envoi des impulsions par radio vers les cibles, réception et traitement des signaux d'écho reçus pour déterminer la position et la vitesse des objets ou des sources qui ont renvoyé les impulsions, et enfin association des sources aux cibles et mise à jour de leurs trajectoires. Pour un suivi efficace, chacune des tâches ci-dessus doit être exécutée périodiquement à une fréquence qui dépend de la distance, de la vitesse, et de l'importance des cibles et, à chaque invocation, doit exécuter la tâche dans un délai ou suivant une échéance spécifique. Une autre caractéristique d'un système temps réel est qu'il devrait être prévisible. La prévisibilité signifie qu'il devrait être possible de montrer ou de prouver que les exigences des tâches sont toujours satisfaites quelles que soient les hypothèses faites, comme la charge de calcul sur le ou les processeurs.

En fonction des conséquences induites par un non respect des exigences des tâches, les contraintes de temps sont généralement classées comme dures ou souples. Une contrainte de temps dure correspond par exemple à échéance dont la violation peut entrainer des conséquences désastreuses tels que des risques pour des vies humaines ou des dégâts matériels importants. Des systèmes de contrôle de procédés industriels, des contrôleurs embarqués dans l'automobile ou des systèmes de contrôle aérien en aéronautique sont autant d'exemples de systèmes soumis à des contraintes de temps réel dur. Par comparaison, un système temps réel souple possède des contraintes moins critiques, le dépassement de contraintes souples est ainsi possible. Cependant, ces dépassements ne sont en général pas souhaitables du fait que la qualité de service du système se dégrade avec le nombre de dépassements observés et par conséquent, ce nombre est généralement borné. Les systèmes multimédia ou de réalité virtuelle sont des exemples de systèmes à contraintes de temps souples.

Il existe des applications émergentes à la fois temps réel et de grande complexité algorithmique. Des exemples sont les systèmes de suivi automatique de cibles et les systèmes de télé-surveillance. Ces applications ont des contraintes temporelles qui visent à produire une grande réactivité du système et peuvent être aussi nécessaires pour assurer un comportement correct voire crucial dans certaines applications telle que la télé-chirurgie. En outre, leurs besoins de traitement peuvent facilement dépasser la capacité de calcul d'un simple processeur et dans ce cas, un système multiprocesseur peut être nécessaire pour réaliser une exécution efficace de l'application. Par ailleurs, les architectures multiprocesseurs peuvent être plus rentables qu'un processeur unique de même capacité de calcul du fait que le coût d'un système à k-processeurs peut être moins élevé que celui d'un processeur k fois plus rapide (dans l'hypothèse où un processeur de cette performance est en effet disponible).

Ces observations soulignent l'importance croissante des architectures multiprocesseurs dans les systèmes temps réel et celle des algorithmes d'ordonnancement de tâches adaptés à ces architectures. Dans cette thèse, nous nous concentrons sur plusieurs points centraux relatifs à cette problématique. Avant d'introduire les contributions développées dans la thèse, nous décrivons brièvement quelques concepts de base concernant les systèmes temps réel.

1.2 Les systèmes temps réel : concepts de base

Un système temps réel se compose typiquement de plusieurs processus (séquentiels) munis de contraintes temporelles. Nous appelons ces processus des tâches. Dans la plupart des systèmes temps réel, les tâches sont récurrentes, c'est à dire que chaque tâche est appelée répétitivement tant que le système fonctionne. Le modèle de tâches périodiques introduit par Liu et Layland [48] fournit la notion la plus simple d'une tâche récurrente. Chaque tâche périodique T_i est caractérisée par une phase ϕ_i, une période P_i, une échéance relative d_i, et un temps d'exécution pire cas C_i ($C_i \leq d_i$). La quantité B_i représente le temps d'exécution minimum d'une tâche. Une telle tâche est activée toutes les P_i unités de temps, et sa première activation se produit à l'instant ϕ_i.

Nous désignons par le terme ”travail” chaque invocation d’une tâche, et l’instant correspondant à une invocation d’une tâche comme instant d’activation du travail. Ainsi, le paramètre relatif à l’échéance est employé pour spécifier les contraintes temporelles des travaux d’une tâche périodique. Sauf indication contraire, nous supposons que l’échéance relative d’une tâche périodique est égale à sa période. En d’autres termes, chaque travail doit terminer son exécution avant l’activation du prochain travail de la même tâche. Nous définissons les différents paramètres d’exécution d’une tâche suivants. Le paramètre $C_i^{rem}(t)$ représente la durée d’exécution restante de la tâche T_i à l’instant t définie par $C_i^{rem}(t) = C_i - C_i^{completed}(t)$ où $C_i^{completed}(t)$ représente la fraction du temps d’exécution réalisée de la tâche T_i jusqu’à l’instant t. Le paramètre $B_i^{rem}(t)$ représente le temps d’exécution minimum restant à l’instant t. La quantité $d_i^{rem}(t) = d_i - t$, définit la durée restante à l’échéance de T_i. Le paramètre AET_i associé à une tâche T_i représente le temps d’exécution effectif d’un travail relatif à une invocation de cette tâche, il s’agit d’un paramètre dynamique.

A chaque tâche T_i on associe une latence L_i définie par:

$$L_i = P_i - C_i$$

La latence d’une tâche T_i à l’instant t est définie par :

$$L_i(t) = d_i^{rem}(t) - C_i^{rem}(t)$$

Un système de tâches périodiques dans lequel toutes les tâches ont une phase nulle est un système de tâches périodiques synchrones. Sauf indications contraires, nous considérons dans la suite de la thèse des tâches de cette nature. Le terme HP désigne l’hyperpériode de toutes les tâches c’est à dire le plus petit commun multiple des périodes des n tâches. Le poids ou l’utilisation d’une tâche T_i, noté μ_i, est le rapport entre son temps d’exécution pire cas et sa période (C_i/P_i). Le poids d’une tâche détermine la fraction de temps d’un processeur que la tâche requiert pour son exécution pendant sa période. Le poids d’un système de tâches est la somme des poids de toutes les tâches dans le système.

Nous supposons que toutes les tâches sont préemptives, c’est à dire qu’une tâche peut être interrompue pendant son exécution et reprise plus tard à partir du même point d’arrêt. Nous supposons que les coûts en temps liés à la préemption d’une tâche sont négligeables. Nous supposons également que toutes les tâches sont indépendantes, en d’autres termes, l’exécution d’une tâche n’est pas conditionnée par un ou des résultats fournis par d’autres tâches. En particulier, nous supposons que les tâches ne partagent aucune ressource autre que le processeur, et qu’elles ne s’auto-suspendent pas pendant leur exécution.

1.3 Les stratégies d'ordonnancement temps réel et leur classification

Généralement, un algorithme d'ordonnancement temps réel affecte une priorité à chaque travail, et dans le cas d'un système à M processeurs, planifie l'exécution des M travaux ayant les priorités les plus élevées à chaque instant.

1.3.1 1.3.1 Faisabilité et optimalité

Un système de tâches périodiques est faisable sur une architecture mono ou multiprocesseur si et seulement si pour chaque invocation de chaque tâche, il est possible de respecter toutes les échéances. Un test de faisabilité pour une classe de systèmes de tâches est défini comme une condition nécessaire et suffisante à vérifier pour s'assurer que tout système de tâches dans cette classe soit faisable.

Un algorithme employé pour planifier des tâches (c'est à dire, allouer un temps du processeur aux tâches) est désigné sous le nom d'algorithme d'ordonnancement. Un système de tâches τ est ordonnançable par l'algorithme A si A peut garantir les échéances de tous les travaux de chaque tâche dans τ. Une condition suivant laquelle tous les systèmes de tâches dans une classe sont ordonnançables par A est un test d'ordonnançabilité pour A pour cette classe de systèmes de tâches. Un algorithme d'ordonnancement est considéré comme optimal pour une classe de systèmes de tâches si sa condition d'ordonnançabilité est identique à la condition de faisabilité pour cette classe.

1.3.2 Ordonnancement en ligne et hors ligne

Dans un ordonnancement hors ligne, l'ordonnancement complet d'un système de tâches (jusqu'à un temps défini par l'hyperpériode des tâches) est pré-calculé avant que le système ne soit exécuté réellement; l'ordonnancement des exécutions des tâches est effectué en utilisant une table basée sur cet ordonnancement pré-calculé. Afin de produire un ordonnancement hors ligne, il est nécessaire de connaître les temps exacts d'activation de tous les travaux du système. Cependant, une telle connaissance n'est pas toujours disponible, en particulier pour ceux qui contiennent des tâches sporadiques, ou périodiques avec des phases inconnues. Même si une telle connaissance était disponible, alors il peut être irréaliste de mémoriser l'ordonnancement complet dans le cas où le PPCM des périodes des tâches est très grand. Par ailleurs, un ordonnanceur en ligne choisit un travail à ordonnancer sans aucune connaissance des activations des travaux futurs. On peut noter qu'un algorithme d'ordonnancement en ligne peut être également employé pour produire un ordonnancement hors ligne. De manière évidente, un ordonnancement hors ligne est plus efficace à l'exécution qu'un ordonnancement en ligne, cependant la flexibilité est réduite dans le cas hors ligne. Les ordonnanceurs en ligne doivent être efficaces, mais peuvent avoir à prendre des décisions d'ordonnancement sous-optimales.

1.3.3 Priorités Statiques et Dynamiques

La plupart des algorithmes d'ordonnancement est basée sur le principe de priorité: ils attribuent des priorités aux tâches ou aux travaux dans le système et ces priorités sont utilisées pour sélectionner un travail à exécuter en fonction des décisions d'ordonnancement prises. Un algorithme d'ordonnancement basé sur les priorités peut déterminer les priorités des tâches ou de travaux de différentes manières. Un algorithme d'ordonnancement est à priorités statiques si une priorité unique est associée à chaque tâche, et tous les travaux issus d'une même tâche possèdent la priorité liée à cette tâche. Ainsi, si la tâche T_i a une priorité plus élevée que la tâche T_j , alors lorsque toutes les deux deviennent actives, le travail de T_i aura une priorité plus élevée que celui de T_j . Un exemple d'algorithme d'ordonnancement basé sur des priorités statiques est l'algorithme Rate-Monotonic (RM) [48]. L'algorithme RM affecte les priorités les plus élevées aux tâches ayant des périodes les plus courtes. Les algorithmes à priorité dynamique permettent une plus grande flexibilité dans l'attribution des priorités aux tâches. La priorité d'une tâche peut varier entre les travaux ou même au sein du même travail. Un exemple d'algorithme d'ordonnancement à priorités dynamiques est l'algorithme Earliest Deadline First (EDF) [48]. Cet algorithme associe une priorité plus élevées aux travaux ayant des échéances plus proches. Il est prouvé que cet algorithme est optimal pour ordonnancer des tâches périodiques et sporadiques sur une architecture monoprocesseur [48, 54]. L'algorithme Least Laxity-First (LLF) [54] est également un exemple d'algorithme à priorités dynamiques - il est également optimal pour des architectures monoprocesseurs. Comme son nom le suggère, sous LLF, les travaux ayant des laxités les plus faibles ont les priorités les plus élevées.

1.4 Ordonnancement temps-réel multiprocesseur

Dans cette ce paragraphe, nous détaillons des approches d'ordonnancement développées dans le cas multiprocesseur.

1.4.1 Approches d'ordonnancements multiprocesseurs

Deux approches traditionnelles considérées pour l'ordonnancement sur architectures multiprocesseurs sont l'ordonnancement avec partitionnement et l'ordonnancement global.

1.4.1.1 Approche par partitionnement

Sous le terme ordonnancement par partitionnement, il faut considérer un ensemble de tâches (statiquement) partitionnées sur les processeurs. Chaque tâche est allouée à un processeur unique sur lequel elle s'exécute. Chaque processeur possède son propre ordonnanceur qui lui permet de sélectionner localement les tâches à exécuter parmi celles qui lui ont été attribuées. Il possède également une file locale des tâches dans laquelle sont rangés les travaux prêts à être exécutés. En d'autres termes, l'espace des priorités affecté à chaque tâche est local à chaque processeur. De plus, les ordonnanceurs de chaque processeur peuvent être tous basés sur le même algorithme d'ordonnancement ou chacun peut utiliser son propre algorithme. Enfin, le partitionnement des tâches sur l'ensemble des processeurs doit s'assurer que pour chaque processeur, la somme des utilisations des tâches allouées au processeur est au plus égale à la limite d'utilisation possible de son ordonnanceur.

1.4.1.2 L'approche par ordonnancement global

Par opposition à l'ordonnancement par partitionnement, l'ordonnancement global utilise un espace unique pour les priorités. Il n'existe qu'une seule file globale des tâches prêtes. A chaque instant, les M tâches prêtes qui possèdent les plus grandes priorités dans l'espace global des priorités s'exécutent sur les M processeurs. Aucune restriction n'est imposée ici sur le choix du processeur qui exécute une tâche prête et sélectionnée. En particulier, l'exécution d'une tâche peut être amenée à migrer d'un processeur à un autre en fonction des décisions d'ordonnancement.

1.4.1.3 L'ordonnancement hybride à deux niveaux

Certains algorithmes d'ordonnancement ne répondent pas strictement aux deux approches présentées ci-dessus mais utilisent les deux approches à la fois. Par exemple, des algorithmes dans lesquelles certaines tâches ne peuvent pas migrer et doivent donc être affectées à un processeur spécifique alors que d'autres tâches ont la possibilité de migrer. Ces algorithmes suivent une stratégie mixte. En général, un ordonnancement basé sur une stratégie mixte est réalisé sur deux niveaux. Au premier niveau, un ordonnanceur global et unique détermine le processeur sur lequel les tâches devraient être allouées et ce en fonction de règles globales. Au second niveau, les tâches allouées à un processeur sont ordonnancées localement en fonction de priorités locales. Plusieurs variantes de ce modèle général ainsi que d'autres types d'ordonnancements hybrides sont également possibles.

1.5 Systèmes d'exploitation embarqués pour la gestion de l'énergie

L'accroissement important de la demande pour l'exécution d'applications sur les systèmes autonomes (sur batterie) met en évidence un besoin d'exécuter des applications en tenant compte de la consommation d'énergie engendrée. Avec les nouvelles générations de processeurs basse consommation, l'objectif est d'ajuster dynamiquement et de manière matérielle les performances souhaitées du ou des processeurs de sorte à satisfaire la charge induite par l'exécution des tâches tout en minimisant la consommation d'énergie et ce afin d'allonger la durée de fonctionnement entre deux recharges de la batterie. Par rapport aux conceptions traditionnelles basse consommation qui visent à obtenir les hautes performances, il est ici nécessaire de considérer une collaboration Entre matériel et logiciel pour réduire la consommation à chaque fois que les performances du système peuvent être relâchées.

Les techniques DVFS (Dynamic Vlotage and Frequency Scaling) sont intégrées dans de nombreux processeurs commerciaux tels que les processeurs Transmetas, Crusoe, Intels XScale et Texas Instruments OMAP3430. Du fait que la puissance dynamique (celle liée à l'exécution des traitements) dans les circuits CMOS dépende de manière quadratique de la tension d'alimentation du circuit, l'abaissement de la tension est un moyen efficace pour réduire la consommation de puissance. Cependant, cette réduction affecte également et ce de manière négative les performances du système en augmentant les temps de réponse. Par conséquent, des algorithmes efficaces de gestion du DVFS doivent permettre de maintenir dynamiquement les performances requises par les applications.

Quelque soit la stratégie basse consommation utilisée pendant la conception de circuits, cette stratégie doit être associée avec les applications et le système d'exploitation et de plus, la connaissance du comportement des applications est aussi essentielle. La spécification industrielle Advanced Configuration and Power Interface (ACPI)[23] établit des interfaces dans les systèmes d'exploitation afin de permettre de configurer directement les états de sauvegarde d'énergie et les tensions d'alimentation de chaque composant du système et ce de manière individuelle. Cette spécification ACPI a été développée par Hewlett-Packard, Intel, Microsoft, Phoenix et Toshiba. Le système d'exploitation peut ainsi personnaliser des politiques de consommation d'énergie pour maintenir la qualité de service demandée par les applications et définir ainsi pour chaque composant son propre mode d'exécution en fonction des besoins.

1.5.1 DVFS

La problématique liée au changement dynamique en fréquence et en tension (DVFS) est actuellement un domaine ou de nombreuses recherches sont menées pour obtenir des réductions efficaces de la consommation d'énergie. La puissance dynamique consommée par un circuit CMOS est donnée par [28, 72] :

$$PW = \beta C_L V_{DD}^2 f \tag{1.3}$$

Où β est un facteur d'activité moyen du circuit, C_L est la capacité globale du circuit chargée et déchargée à chaque transition, V_{DD} est la tension d'alimentation et f est la fréquence utilisée. Puisque la puissance dépend de manière quadratique de la tension d'alimentation, la réduction de la tension est la méthode la plus efficace pour minimiser l'énergie. Cependant, l'abaissement de la tension réduit également les performances du système du fait que la fréquence maximale du circuit est directement liée à la tension

$$f = \frac{K(V_{DD} - V_{th})^{\beta}}{V_{DD}} \tag{1.4}$$

La constante K est spécifique pour la technologie donnée, V_{th} est la tension de seuil de basculement et β est l'index de saturation de la vélocité,$1 \leq \beta \leq 2$

Les algorithmes DVFS développés pour les systèmes temps réel supposent en général la connaissance a priori des caractéristiques des tâches. Ces caractéristiques sont spécifiées par l'utilisateur ou le concepteur. Pillai et Shin [35] proposent une large gamme d'algorithmes pour les systèmes temps réel. Dans leurs algorithmes " statiques ", une seule fréquence est calculée puis utilisée pour exécuter toutes les tâches. Cette fréquence est déterminée en allongeant de manière identique les temps d'exécution de toutes les tâches de telle sorte que la tâche la moins prioritaire se termine le plus tard possible tout en vérifiant son échéance. Cette approche s'avère pessimiste du fait que le nombre total de préemptions par les tâches de plus hautes priorités n'est pas uniformément distribué quand des tâches ont des périodes différentes. De même, une tâche peut rencontrer moins de préemptions relativement à sa propre exécution si elle se termine plus tôt que son échéance, ce qui a pour effet de réduire la consommation en énergie.

Aydin et al. [10] proposent un algorithme optimal avec des points de tensions statiques utilisant la solution d'un ordonnancement basé sur la notion de récompense. Leur approche fixe une fréquence unique pour chaque tâche et se focalise sur une politique d'ordonnancement de type EDF. La technique DVFS minimise efficacement la consommation d'énergie. Cependant le coût de changement de couple tension/fréquence est élevé dans certaines architectures. Un exemple d'un tel système est le Compaq IPAQ, qui nécessite au moins 20 ms pour synchroniser l'horloge de la SDRAM après un changement en tension/fréquence. De même, le changement de tension/fréquence sur un ARM1136 (architecture Freescale IMX31) est de l'ordre de 4 ms. Pour ces systèmes, le changement en tension/fréquence aux instants définis par les terminaisons des tâches qui peuvent précéder celle relative à leur temps exécution maximum est inapplicable. Dans ce mémoire, on s'attache à minimiser les instants de changement de tension/fréquence mais pas au prix d'un nombre élevé de cycles gaspillés lorsque le processeur est inactif.

1.6 Ordonnancement Auto-adaptatif

L'ordonnancement de travaux où chaque travail possède un degré de parallélisme variable durant son exécution peut permettre d'adapter le nombre de processeurs utilisés et ce en définissant de manière statique un nombre de processeurs alloués à chaque changement de degré de parallélisme. Cette adaptation peut être aussi réalisée de façon dynamique par l'ordonnanceur afin de réduire l'inactivité de processeurs induite par une allocation statique de processeurs à chaque tâche. Le parallélisme disponible au cours de l'exécution d'une tâche peut être appréhendé avec une granularité variable, aussi on peut imaginer d'ordonnancer des tâches dynamiquement et de manière adaptative. Pour les travaux dont le degré de parallélisme est inconnu à l'avance (création dynamique de tâches) et/ou ce degré de parallélisme est non constant pendant l'exécution, la stratégie d'associer un nombre fixe de processeurs par travail peut engendrer un important gaspillage en nombre de cycles processeur du fait qu'un travail ayant un faible parallélisme effectif a pu se voir attribué plus de processeurs qu'il n'a pu en utiliser. Par ailleurs, dans un environnement multiprogrammation, un ordonnancement non adaptatif peut ne pas permettre l'exécution d'un nouveau travail du fait que l'allocation statique des processeurs aux travaux en cours d'exécution peut conduire à une réservation de tous les processeurs même si ceux-ci ne sont pas réellement utilisés. Avec un ordonnancement adaptatif, l'ordonnanceur de travaux peut changer dynamiquement le nombre de processeurs associés à un travail, pendant l'exécution de celui-ci. Ainsi, un nouveau travail peut éventuellement entrer dans le système tant que l'ordonnanceur peut allouer des processeurs non utilisés.

1.7 Contributions

Les études menées dans cette thèse ont pour dénominateur commun le constat suivant: *les algorithmes d'ordonnancement temps-réel ont des bornes d'ordonnançabilité au plus égales à la capacité de traitement de l'architecture (et ce suivant certaines hypothèses). Cependant, ces algorithmes peuvent gagner en efficacité dès lors que leur impact sur les temps de gestion est réduit et ainsi augmenter la Qualité de Service (QoS) des applications. Cet accroissement en efficacité peut être obtenu par une meilleure prise en compte de paramètres implicites des tâches. De plus, les bornes d'ordonnançabilité égales à la capacité de traitement de l'architecture peuvent être atteintes en relâchant certaines des hypothèses classiquement considérées.*

Dans la suite nous décrivons les contributions de cette thèse de manière plus détaillée. Les bornes d'ordonnançabilité sont de toute évidence importantes à identifier mais l'efficacité est aussi essentielle dans un grand nombre d'applications émergentes et temps réel et ce afin d'améliorer la QoS de l'application et de minimiser la consommation d'énergie du système. L'objectif est de proposer des algorithmes d'ordonnancement qui possèdent un temps d'exécution faible du fait d'une complexité algorithmique réduite liée en particulier à une diminution du nombre d'invocations de l'ordonnanceur, c'est-à-dire la réduction du nombre d'événements d'ordonnancement. Par ailleurs, des algorithmes d'ordonnancement qui augmentent le nombre de tâches exécutées dans un temps imparti en exploitant des paramètres dynamiques des tâches en cours d'exécution, sont a priori efficaces puisque la QoS de l'application est directement liée au nombre de tâches exécutées pendant ce temps. Dans ce mémoire, nous abordons les problèmes d'efficacité d'ordonnancement en apportant des modifications significatives aux algorithmes d'ordonnancement existants et ce en cherchant à minimiser le nombre d'événements d'ordonnancement et en diminuant le coût de gestion d'un événement d'ordonnancement ce qui a pour résultat d'améliorer la QoS de l'application (partie décrite brièvement dans le paragraphe 1.7.1). Nous abordons également les problèmes liés à la consommation d'énergie de l'architecture cible et proposons une démarche pour la minimiser sans compromettre les bornes d'ordonnançabilité (paragraphe 1.7.2). Dans le cas de systèmes multiprocesseurs, la borne d'ordonnançabilité égale au nombre de processeur peut être atteinte mais au prix d'hypothèses souvent irréalisables vis-à-vis de l'application du fait des coûts élevés induits par la complexité de gestion en ligne et les nombres de préemptions et migrations de tâches. Nous apportons une réponse à ces problèmes dans le cas d'architectures multiprocesseurs en proposant des techniques qui relâchent les hypothèses et permettent ainsi de réduire le nombre de préemptions de manière importante (paragraphe 1.7.3). Nous avons également étudié des algorithmes d'ordonnancement hybrides et présentons trois approches ayant des bornes optimales d'ordonnançabilité avec un coût maîtrisé (paragraphe 1.7.4).

1.7.1 Algorithmes d'ordonnancement RUF

Considérons un ensemble de tâches composé de tâches critiques et non critiques tel que la charge totale de cet ensemble est supérieure à 100%. Il se pose alors le problème de définir une technique d'ordonnancement qui garantit une terminaison de toutes les tâches critiques avant leurs échéances tout en maximisant les exécutions des tâches non critiques. Des algorithmes existants (tels EDF, MUF, CASH) n'apportent pas de garanties vis-à-vis des tâches critiques lors de situations transitoires de surcharge ou bien ne maximisent pas l'exécution des tâches non critiques. Dans ce contexte, nous proposons d'exploiter les paramètres dynamiques (en ligne) des tâches, c'est à dire les intervalles de temps $C_i - AET_i$ où le processeur est disponible à chaque exécution d'une tâche lorsque le temps d'exécution effectif de la tâche est inférieur à son temps d'exécution pire cas. L'objectif n'est pas seulement de fournir des garanties aux tâches critiques mais aussi de maximiser l'exécution des tâches non critiques. Pour répondre à ce problème, un nouvel algorithme d'ordonnancement appelé RUF (Real Urgency First) est proposé. Nous définissons un mécanisme de contrôle des tâches non critiques afin de maximiser leurs exécutions et ceci sans compromettre les échéances des tâches critiques. Ce contrôle des tâches non critiques à un instant t dépend des paramètres dynamiques des tâches qui ont terminé leurs exécutions avant l'instant t. Nous illustrons le principe à travers un exemple et nous montrons expérimentalement que l'algorithme proposé génère moins de préemptions que les algorithmes existants tout en maximisant les exécutions des tâches non critiques.

1.7.2 Optimisation de la consommation

Dans le cas de systèmes monoprocesseurs, des algorithmes d'ordonnancement temps réel ont des bornes d'ordonnançabilité optimales mais au prix d'une complexité en ligne qui peut être élevée, avec pour conséquence un nombre important de préemptions de tâches (ou d'événements d'ordonnancement). Ce nombre élevé de préemptions diminue en pratique l'ordonnançabilité des tâches, mais également augmente la consommation d'énergie du système. Ces algorithmes n'exploitent pas en général tous les paramètres implicites et dynamiques (en ligne) des tâches aussi des optimisations sont possibles afin de minimiser les coûts de gestion par l'ordonnanceur et l'énergie du système. La laxité est par exemple un paramètre implicite d'une tâche qui fournit une certaine flexibilité à l'ordonnanceur. L'ordonnanceur peut exploiter cette flexibilité pour relâcher certaines règles de l'algorithme d'ordonnancement et ainsi aider à réduire le coût lié aux préemptions des tâches. Nous proposons d'utiliser ce paramètre implicite dans un algorithme d'ordonna-ncement afin de réduire très significativement le nombre de préemptions des tâches. Deux variantes de cette approche sont proposées, l'une est statique, l'autre est dynamique, toutes deux sont appliquées aux politiques d'ordonnancement EDF et RM. On peut remarquer également que les préemptions des tâches sont en général liées au taux de charge du processeur par les tâches. L'utilisation de la technique de DVFS sur le processeur dont l'objectif est de minimiser sa fréquence en vue de diminuer la consommation d'énergie a également pour effet inverse d'augmenter la charge relative du processeur et pas conséquence le nombre de préemptions. Ainsi, il est nécessaire de réduire dans le même temps le nombre de changements de fréquence de fonctionnement du processeur, puisque chaque changement implique une consommation d'énergie et un temps d'inoccupation du processeur.

1.7.3 Algorithme d'ordonnancement ASEDZL

Dans le cas de systèmes multiprocesseurs, des algorithmes d'ordonnancement optimaux ont été proposés, basés sur un modèle d'ordonnancement fluide avec une idée d'équité sous-jacente. Les algorithmes basés sur cette approche d'équité impliquent un coût de gestion élevé dû aux préemptions et aux invocations de l'ordonnanceur ce qui les rend parfois inutilisables. Pour minimiser ce coût de gestion, nous proposons un algorithme d'ordonnancement qui n'est justement pas basé sur un ordonnancement équitable. Dans cet algorithme appelé ASEDZL (Anticipating Slack Earliest Deadline first until Zero Laxity), les tâches ne sont pas allouées à un processeur pour un temps fonction de leurs poids, elles sont sélectionnées pour s'exécuter entre deux requêtes consécutives de tâches de telle sorte que les tâches ayant les échéances les plus proches sont assurées de s'exécuter jusqu'à la prochaine requête. Cet algorithme fournit de meilleurs résultats en termes de nombre de préemptions et de migrations de tâches par rapport aux algorithmes à ordonnancement fluide. Nous illustrons les principes par l'intermédiaire de deux exemples et nous montrons également par simulation que les résultats obtenus sont meilleurs (en préemptions et migrations) que les algorithmes classiques qui ont également une borne d'ordonnançabilité égale à la capacité de traitement de l'architecture.

1.7.4 Algorithme d'Ordonnancement Hiérarchique

Les algorithmes d'ordonnancement hybrides ou hiérarchiques garantissent en général de meilleurs résultats que les algorithmes d'ordonnancement globaux ou partitionnés dans le cas d'architectures à mémoire partagée et distribuée. Cependant, un très petit nombre d'algorithmes d'ordonnancement hybrides sont proposés ayant une borne d'ordonnançabilité égale au nombre de processeurs (en considérant les hypothèses précisées dans le Chapitre 5). Une technique de supertâche est proposée par Moir et al. [53] dans un algorithme d'ordonnancement hybride où l'algorithme Pfair est utilisé comme un ordonnanceur global. Cependant, il est nécessaire de définir une condition limite pondérée entre les tâches locales (celles partitionnées) et les supertâches correspondantes pour atteindre la borne maximum d'ordonnançabilité. Nous établissons cette condition dans le chapitre cinq. Nous proposons également d'utiliser l'algorithme ASEDZL comme ordonnancement global avec l'intérêt qu'il n'impose aucune condition sur les poids respectifs des supertâches et des tâches locales. Nous comparons l'algorithme proposé avec des approches existantes pour illustrer que de meilleurs résultats peuvent être obtenus. Un nouvel algorithme d'ordonnancement hybride est également proposé où des intervalles de temps sont réservés pour l'exécution des tâches locales ou globales.

1.7.5 ÆTHER: Self-adaptive Middleware

Les algorithmes d'ordonnancement auto-adaptatifs considèrent généralement des tâches avec un degré de parallélisme déterminé statiquement et une allocation de ressources dynamique afin d'une part, de tenir compte de la variation du parallélisme apparaissant lors de l'exécution de la tâche suivant son flot de contrôle et d'autre part, de réduire le gaspillage en ressources. Ces algorithmes peuvent prendre en compte la création ou la suppression dynamique de tâches mais ils ne sont pas capables de traiter le cas de tâches ayant des capacités à s'auto-optimiser à l'exécution ce qui conduit à un degré de parallélisme effectif pendant l'exécution différent de celui calculé statiquement. Par ailleurs, l'architecture peut aussi s'auto-adapter pour exécuter séquentiellement des tâches a priori parallèles sur une ressource spécifique plutôt que de les exécuter en parallèle sur différentes ressources standardisées. Ces optimisations à l'exécution de l'architecture et de l'application nécessitent un mécanisme auto-adaptatif de gestion de ressources capable d'appréhender ces variations à l'exécution. Nous proposons un modèle d'ordonnancement auto-adaptatif qui prend en compte ces optimisations à l'exécution et fournit les garanties de respect d'échéances pour les tâches temps réel critique. L'ensemble de toutes les contributions introduites ci-dessus est détaillé dans les chapitres suivants. Dans chaque chapitre, un état de l'art associé au thème abordé est présenté dans un premier temps avant de décrire les approches proposées et les résultats obtenus. Dans le dernier chapitre, nous résumons nos différentes contributions et proposons des pistes pour de futures recherches.

Chapter 2

Real Urgency First Scheduling Algorithm

2.1 Introduction

In real time systems all tasks have some certain deadline constraints which they have to meet in order to provide required level of Quality of Service (QoS). In case of hard real time systems, tasks must meet their deadlines to avoid catastrophic happenings. Many scheduling algorithms have been proposed that tackle these problems with different ways. Real-time scheduling algorithms may assign priorities statically, dynamically, or in a hybrid manner, which are called fixed, dynamic and hybrid scheduling algorithms, respectively.

These scheduling algorithms are optimal on mono-processor architecture i.e., if utilization of task set is less than or equal to 100% then all tasks are guaranteed to meet their deadline constraints, but when these algortihms are applied in transient overload situations, they perform poorly. For example, EDF and LLF does not guarantee that which task will fail in overload situations. As a result, it is possible that a very critical task may fail at the expense of a lesser important task.

The maximum urgency first algorithm MUF [74] solves the problem of unpredictability during a transient overload for EDF, LLF and MLLF [57] algorithms. The MUF algorithm is a combination of fixed and dynamic priority scheduling, also called mixed priority scheduling. With this algorithm, each task is given an urgency which is defined as a combination of two fixed priorities (criticality and user priority) and a dynamic priority that is inversely proportional to the laxity. The criticality has higher precedence over the dynamic priority while user priority has lower precedence than the dynamic priority. MUF algorithm provides guarantees, that no critical task misses its deadline in a transient overload. But this algorithm has one serious shortcoming, that it is not a fair scheduler. It provides guarantees to critical tasks, but may unnecessarily cause many non critical tasks to miss their deadlines. It is due to assigning higher static priorities to critical tasks over non critical tasks.

In imprecise computation techniques [11, 43], each task is divided into two subtasks; mandatory subtask and optional subtask, instead of grouping tasks into two subsets. If these tasks are terminated before their completion but after executing mandatory units, the results are acceptable, and executing a part of optional subtask does not decrease the performance. This model is quiet different from the basic model and requires a lot of support from programmers. Moreover, these algorithms schedule optional subtasks in the same way as non critical tasks are scheduled by MUF algorithm.

Quality of service based resource allocation model QRAM [69] tries to allocate resources to different applications, which may need to satisfy more than one QoS requirements. Utility function is defined for each task to allocate resources to applications to increase its performance, but it does not explain that how QoS of a real time application can be increased by assigning more resources of processor to non critical tasks while providing guarantees to critical tasks.

Capacity sharing algorithm (CASH)[20] is based on the idea of resource reservation. Resource reservation mechanism bounds effects of task interference. CASH algorithm also performs the efficient reclaiming of unused computation time to relax the utilization constraints imposed by isolation. It provides guarantees to both critical tasks and non critical tasks if and only if

$$\sum_{i=1}^{k} \frac{C_i}{P_i} + \sum_{j=1}^{m} \frac{C_j}{P_j} \leq 1 \tag{2.1}$$

where
$\sum_{i=1}^{k} \frac{C_i}{P_i}$ denotes sum of utilization of critical tasks and
$\sum_{j=1}^{m} \frac{C_j}{P_j}$ represent sum of utilization of non critical tasks

If sum of utilization of critical tasks approaches 100%, then no non critical task can be added in the system. If sum of utilization of critical tasks and sum of utilization of non critical tasks increases more than 100%, then CASH can not provide guarantee to critical tasks.

We propose a scheduling algorithm called RUF (Real Urgency First), where the critical tasks are always guaranteed to meet their deadlines. It also improves the execution of non critical tasks. The proposed scheduling algorithm is an extended model of MUF algorithm. MUF is biased towards critical tasks, as MUF assigns higher static priorities to critical tasks over non critical tasks. It may, unnecessarily, cause many non critical tasks to miss their deadlines, which could be executed without causing critical tasks to miss their deadlines. We divide a task set into two groups. The first group is the set of critical tasks, and second group is set of non critical tasks. In our algorithm, critical tasks are not assigned higher static priorities over not critical tasks, rather they are placed in two different queues i.e., ready queue and temporary ready queue. Scheduler selects tasks from ready queue to execute, and non critical tasks are moved from temporary ready queue to ready queue at runtime (Fig:2.1). Non critical tasks are selected to execute in presence of ready critical tasks.

2.2 Approach Description

We assume that there are k critical tasks and m non critical tasks such that $k + m = n$. In the task set τ, tasks are sorted based on their period lengths in an ascending (non-decreasing) order, i.e., the period of T_i is smaller than that of T_{i+1}.

2.2.1 Dynamic Slack

Processor utilization of each task is calculated by considering its worst case execution time, which happens rarely in actual execution of a task. Moreover if all critical tasks take processor time equal to C_i during each interval $[(g-1).P_i, g.P_i)$ where g is a positive integer, then no non critical task can be executed (if sum of utilization of critical task is 100%). Non critical tasks are executed if there is positive slack (difference between C_i and AET_i) offered by critical tasks. We call this dynamic slack as available *time_slot*, and $time\_slot_i$ is defined as:

$$time\_slot_i = C_i - AET_i$$

Whenever a task completes its execution, and it has used processor for time units less than its C_i, it increases available *time_slot*. This increment in $time\_slot_i$ may vary from zero to the difference between its C_i and B_i. The total available *time_slot* is updated whenever a task T_i finishes its execution:

$$time\_slot = time\_slot + (C_i - AET_i)$$

We define a task called virtual task which is not a real task and is only used to drive admission control mechanism for non critical tasks, as shown in Fig:2.1. Whenever AET_i of a critical task is less than its C_i, there is a $time\_slot_i$ and scheduler generates/updates parameters of virtual task T_{vt}. The generation of virtual tasks implies that non critical tasks may be admitted into ready queue.

2.2.2 Virtual task

A virtual task has dynamic parameters which depend upon the runtime parameters of critical tasks. Worst case execution time of virtual task, C_{vt}, is defined as *time_slot* while absolute deadline, d_{vt}, of this virtual task is the latest deadline of all those critical tasks

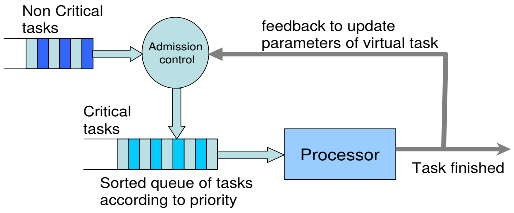

Non Critical tasks
Admission control
feedback to update parameters of virtual task
Critical tasks
Processor
Task finished
Sorted queue of tasks according to priority

FIGURE 2.1: Approach Description

which have contributed to *time_slot* of the system:

$$C_{vt} = time\_slot$$
$$d_{vt} = \max(d_i)$$

Whenever a non critical task T_{nc} is moved from temporary ready queue to ready queue, *time_slot* is recalculated in the following way: (C_{nc} : WCET of non critical task)

$$time\_slot = time\_slot - C_{nc}$$

2.2.3 Dummy Task

If sum of processor utilization of all critical tasks is not 100%, then a dummy task ,T_d, is added to the set of critical tasks. Time period, P_d, of this dummy task is equal to time period of the most frequent non critical task, and its worst case execution time, C_d, is such that sum of processor utilizations of all critical tasks plus processor utilization of dummy task equals to 100%. The dummy task is defined by:

$$P_d = \min(P_1, P_2, P_3, ..., P_m)$$

where $(P_1, P_2, ..P_m)$ are periods of non critical tasks and

$$C_d = P_d \times \left(1 - \sum_{i=1}^{k} \frac{C_i}{P_i}\right)$$

where $\sum_{i=1}^{k} \frac{C_i}{P_i}$ is sum of processor utilizations of critical tasks. Whenever this dummy task executes it takes zero time to execute, and time allocated to it, C_d, is used to execute most appropriate non critical task. When dummy task executes, the value *time_slot* is updated in following way:

$$time\_slot = time\_slot + C_d$$

2.2.4 Management of non critical tasks

Tasks from ready queue are selected by scheduler to execute on the processor. Tasks from temporary ready queue are moved to ready queue depending upon value of *time_slot* and C_i of highest priority non critical task.

2.2.4.1 Selection of Non Critical task

A non critical task, T_{nc}, is inserted into the ready queue to benefit from the available *time_slot*. As it may cause critical task to miss its deadline, there is a need to define an appropriate mechanism, which provides guarantees to the set of critical tasks even in the presence of non critical task in ready queue. There are two approaches to add a non critical task into ready queue, and these approaches are based on parameters of virtual tasks.

Best Effort Approach: The highest priority non critical task T_i is added to ready queue if *time_slot* is greater than C_i of highest priority non critical task. Whenever non critical task is added to ready queue, it is assigned a virtual deadline which may be greater than or equal to its real absolute deadline (Fig:2.2.a). In this case, addition of non critical task to ready queue does not cause critical tasks to miss their deadlines, but successful completion of non critical task is not guaranteed in this case.

Guaranteed Approach: In this approach, a non critical task which has its deadline either equal or greater than absolute deadline of virtual task is selected to add into ready queue (Fig:2.2.b). The worst case execution time of this non critical task is smaller than or equal to available *time_slot*. In this case, addition of non critical task neither causes critical tasks to miss their deadlines nor misses its own deadline.

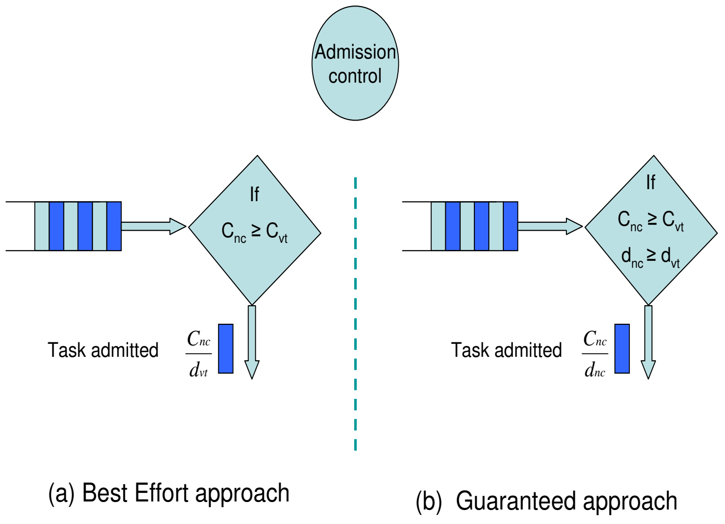


Figure 2.2: Two types of Admission Control

2.2.4.2 Inserting Points of Non Critical Task

Non critical tasks are inserted into the ready queue to benefit from the available *time_slot* at time when it could be executed without making critical task to miss their deadline. A non critical task is inserted into ready queue in two cases.

Sufficient *time_slot*: A non critical task is inserted into ready task queue whenever available *time_slot* is sufficiently high to execute a non critical task. Once a non critical task is added into ready queue, it is selected to execute depending upon its dynamic priority.

Urgency of the *time_slot*: A non critical task is inserted into ready queue at the point when urgency of available *time_slot* has reached. Urgency of available *time_slot* is defines as a scheduling instant when there is no critical task in ready queue, and available *time_slot* is not sufficient enough to execute a non critical task. At this time, a non critical task is inserted into ready queue. This non critical task executes and keeps on executing until either it has finished its execution, or a critical task has been released for its next instant. If a critical task has been released and non critical task has not yet finished its execution, non critical task is not only preempted but it is also removed from the ready queue.

A non critical task added to ready queue at instant of urgency can not cause critical tasks to miss their deadlines, as it is removed from ready queue whenever a critical task is released. But a non critical task admitted to ready queue at the instant when available *time_slot* was sufficient to execute this non critical task, stays in ready queue until either it has finished its execution or its laxity has reached a value less than zero.

Approach described in this subsection allows non critical task to execute only at those points when parameters of virtual task are sufficient to execute a non critical task. This approach is pessimistic in the sense, as it requires worst case execution time of virtual task to be greater than that of non critical task. Non critical task may take processor time less than its worst case execution time to execute. Moreover, the deadline of virtual task is defined as the maximum of deadlines of all those tasks that have contributed to worst case execution time of virtual task. In the following subsection, we present another approach, where *time_slot* is not accumulated and scheduler generates multiple virtual tasks based on *times_slot* of each task.

2.2.5 Multiple Virtual tasks

Whenever actual execution time of a critical task is less than its C_i, there is a *time_slot* for the scheduler which in turn generates a virtual task. In other words, whenever there is a *time_slot*, there is a corresponding virtual task. The worst case execution time of this virtual task is equal to difference of C_i and AET_i of critical task that has finished its execution. The value *time_slot* is not accumulated in this approach. In this approach,

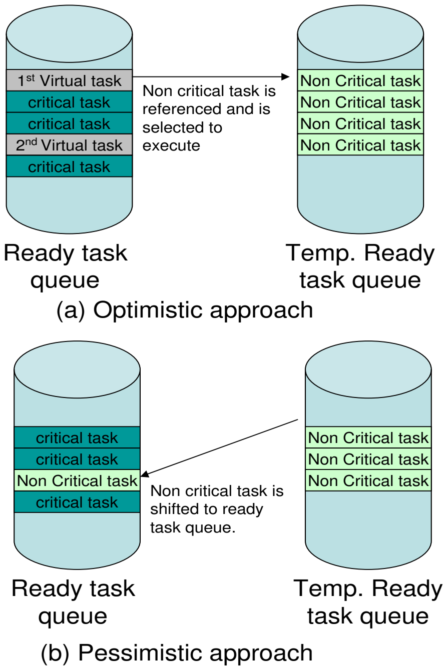


FIGURE 2.3: Comparison of variants of approach

non critical tasks are not shifted from temporary ready queue to ready queue, but are selected to execute when one of virtual task in ready queue gets highest priority. Non critical tasks are never shifted from temporary ready queue to ready queue of critical task. Whenever a virtual task gets the highest priority, scheduler selects the highest priority non critical task to execute. In this case, this task is preempted when *time_slot* of the virtual task is exhausted.

In Fig:2.3.b, a non critical task is shifted to ready queue by replacing two virtual tasks. Non critical task is inserted into ready queue if sum of worst case execution times of 1^{st} and 2^{nd} virtual task is greater than that of non critical task. Moreover it is placed at the position of 2^{nd} virtual task which has lower dynamic priority than that of 1st virtual task to ensure deadline guarantees to critical task. This approach is pessimistic in two aspects, firstly, it is assumed that non critical task takes processor for a time equal to its C_i, secondly the non critical task replaces a virtual task that has the lowest dynamic priority.

2.3 Feasibility Analysis For Critical Tasks

Insertion of non critical task into ready task queue does not make critical tasks to miss their deadlines. Non critical task, T_{nc}, is assigned parameters of virtual task when it is inserted into the ready task queue:

$$C_{nc} \leq time\_slot$$

$$d_{nc} \geq d_{vt}$$

Now, we prove that load of processor does not exceed 100% after insertion of non critical task T_{nc} , hence it does not cause critical tasks to miss their deadlines. At any given time t, total utilization of critical task set is:

$$\sum_{i=1}^{k} \frac{C_i}{P_i} \leq 1$$

$$\sum_{i=1}^{j} \frac{C_i}{P_i} + \sum_{i=j+1}^{k} \frac{C_i}{P_i} \leq 1$$

Let g be equal to $\sum_{i=1}^{j} \frac{C_i}{P_i}$, in this case we have:

$$\sum_{i=j+1}^{k} \frac{C_i}{P_i} \leq (1-g)$$

Let us assume that at given instant $t = t_z$, when a non critical task, T_{nc}, is moved in the ready queue, tasks $T_1, T_2, ..., T_j$ are those critical tasks which have finished their executions and $T_{j+1}, T_{j+2}, ...T_k$ are critical tasks which are ready to execute.
We know that:

$$C_i = AET_i + time\_slot_i$$

Thus:

$$\sum_{i=1}^{j} \frac{AET_i + time\_slot_i}{P_i} = g$$

$$\sum_{i=1}^{j} \frac{AET_i}{P_i} + \sum_{i=1}^{j} \frac{time\_slot_i}{P_i} = g \tag{2.2}$$

Now, we show that insertion of a non critical task does not increase processor utilization. If the worst case execution time C_{nc} of non critical task is such that:

$$time\_slot \geq C_{nc}$$

and

$$\frac{C_{nc}}{P_{nc}} + \sum_{i=1}^{j} \frac{AET_i}{P_i} \leq g$$

then all critical tasks respect their deadlines. The above statement is true if:

$$\frac{C_{nc}}{P_{nc}} \leq \sum_{i=1}^{j} \frac{time\_slot_i}{P_i}$$

$$d_{vt} \geq= \max(d_1, d_2, d_3, ..., d_j)$$

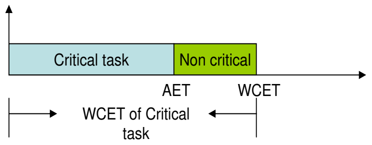


FIGURE 2.4: execution scenario of non critical task.

and

$$d_{nc} \geq d_{vt}$$

We know that:

$$time\_slot = \sum_{i=1}^{j} time\_slot_i$$

$$\sum_{i=1}^{j} \frac{time\_slot_i}{P_i} = \frac{\sum_{i=1}^{j} \frac{P_{nc}}{P_i} \times time\_slot_i}{P_{nc}}$$

Since $\frac{P_{nc}}{P_i} \geq 1$ we get:

$$\sum_{i=1}^{j} \frac{time\_slot_i}{P_i} \geq \frac{time\_slot}{P_{nc}} \Rightarrow$$

$$\frac{C_{nc}}{P_{nc}} \leq \sum_{i=1}^{j} \frac{time\_slot_i}{P_i} \Rightarrow$$

$$\frac{C_{nc}}{P_{nc}} + \sum_{i=1}^{j} \frac{AET_i}{P_i} \leq g$$

Hence, RUF algorithm provides deadline guarantees to critical tasks, because a non critical task is executed in the same way as critical task has used processor time equal to its C_i instead of AET_i. (Fig:2.4).

2.4 Algorithm

As we have explained earlier, that non critical tasks are inserted into ready task queue only at appropriate scheduling instants. These instants are scheduling points when a critical task finishes its execution, and at these instants *time_slot* is recalculated and a non critical task is added into the ready queue either when *time_slot* is found sufficient to execute a non critical tasks or there is no critical task in ready queue. We call this non critical task as T_{nwf}. This non critical task is removed from the critical task queue and is placed back in non critical task queue, whenever one of the critical task becomes ready.

Algorithm 1 Insertion of critical task in ready queue

```
Whenever a critical task $T_i$ is released
if ready-queue =! empty then
    if first-task-in-queue == $T_{nwf}$ then
        remove task $T_{nwf}$ from ready-queue;
        $T_{nwf}$ = 'nothing';
    end if
    $time\_slot = time\_slot$-(current-time-$t_{nwf}$);
end if
insert $T_i$ into ready-queue;
```

t_{nwf} represents the time when T_{nwf} was added in ready queue.

Whenever a non critical task is released, it is added only in temporary ready queue. The highest priority task in temporary ready queue is represented by T^f_{nc}. If available *time_slot* is greater than C^f_{nc} of T^f_{nc}, then it is added into ready queue. Whenever a task completes its execution, *time_slot* is updated and scheduler calculates if available *time_slot* is sufficient enough to execute the highest priority non critical task. If it is sufficient, then the highest non critical task is added in ready queue. Otherwise scheduler waits for another task to finish.

Algorithm 2 Insertion of non critical task in temporary ready queue on its release

```
Whenever $T_{nc}$ is ready
insert $T^f_{nc}$ into temp-ready-queue;
if $time\_slot \geq C^f_{nc}$ then
    insert $T^f_{nc}$ into ready-queue;
    $time\_slot = time\_slot$-$C^f_{nc}$;
end if
```

Algorithm 3 Insertion of non critical task in ready queue when a task finishes its execution

```
Whenever a Task $T_i$ completes its execution
$time\_slot = time\_slot + time\_slot_i$
if temp-ready-queu =! empty then
    if $time\_slot \geq C^f_{nc}$ then
        if $d_{vt} < d_i$ then
            $d_{vt} = d_i$;
        end if
        insert $T_{nc}$ into ready-queue;
        $time\_slot = times\_slot$-$C^f_{nc}$;
    else if ready-queue == empty then
        insert $T^f_{nc}$ into ready-queue;
        $T_{nwf} = T_{nc}$;
    end if
    $time\_slot = time\_slot$-$C^f_{nc}$;
end if
```

2.5 Experimental Results

In this section, we compare the performance of our proposed algorithm RUF with MUF algorithm. The simulation runs on Cofluent Studio simulator[1] with a task set composed of critical and non critical tasks. The worst case execution times and time periods of all these tasks are generated randomly. The time unit of execution time and period is

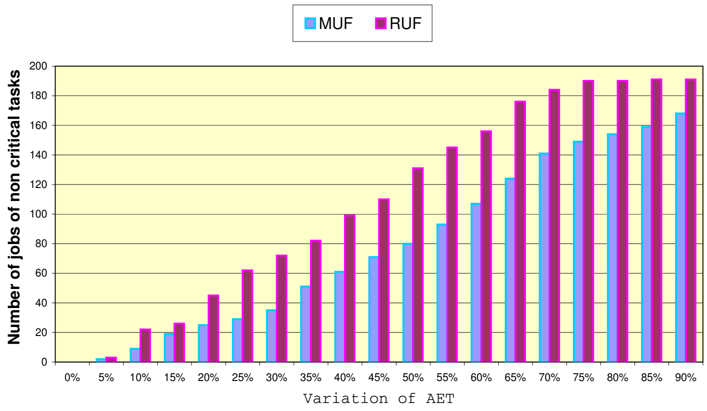


FIGURE 2.5: Execution of non critical tasks over hyper-period of critical tasks

in micro-second. We assume that each task's deadline is equal to its period. In the experiment, the system runs for time units equal to hyper-period of critical tasks. We observe the number of non critical tasks executed in one hyper-period by varying the actual execution time of critical tasks. A 10% variation of AET means that execution times of all tasks may be decreased by 10% at maximum (execution time of all tasks is decreased randomly during execution, but it is never decreased more than the value specified). In these experiments, we have a task set composed of 5 critical tasks with 100% utilization, and 3 non critical task with overall weight of 50%. We plot the results in Fig:2.5 and Fig:2.6. We could observe that RUF executes more number of non critical tasks over a hyper period than number of non critical tasks executed by MUF scheduling algorithm.

Moreover, if the number of tasks in the system increases, then chances of executing more non critical tasks increase as compared with MUF algorithm. This is due to the fact that the chances of presence of at least one critical task in ready queue increases , and MUF scheduling keeps non critical tasks out of execution while in case of RUF algorithm, the recalculation points, when parameters of virtual task are modified, increases and hence the chances of executing non critical tasks are augmented as well (Fig:2.7).

Example 2.1. *We compare our algorithm RUF with MUF to illustrate the different steps of proposed algorithm. This example explains the points when decisions are made about admission of non critical task into ready queue and what are parameters assigned to this non critical task at time of admission. this algorithm does provide guarantees to critical task, and improves execution of non critical tasks as well. The task set considered for this example is given in Table 2.1.*

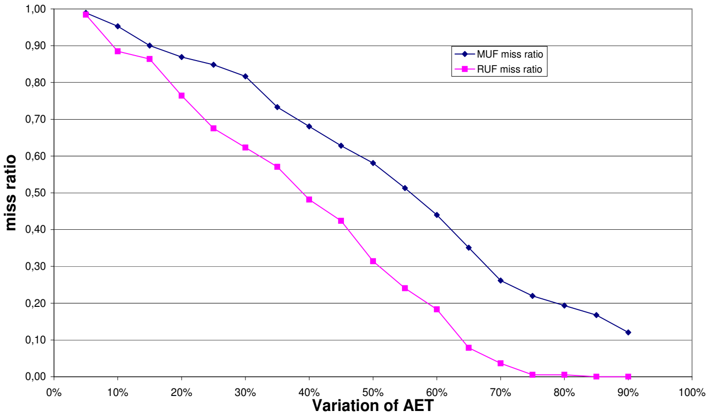


Figure 2.6: Miss ratio of non critical tasks over hyper-period

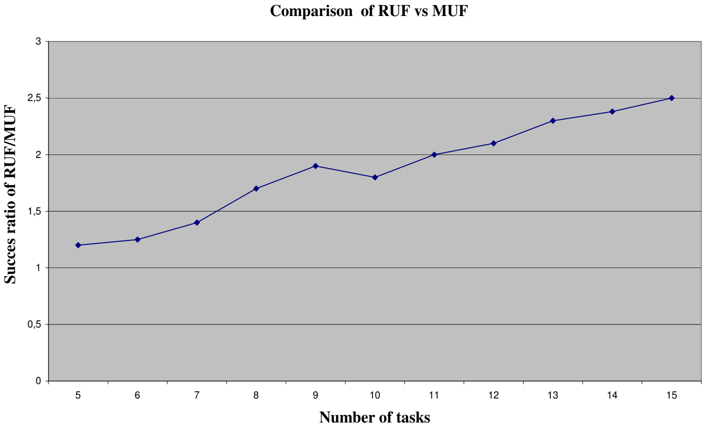


Figure 2.7: Success ratio of RUF vs MUF

Task	Criticality	Period	WCET	AET	Utilization
T_1	High	6	2	1	33%
T_2	High	10	4	3	40%
T_3	High	12	3	3	25%
T_4	Low	15	4	4	27%

Table 2.1: Task Parameters

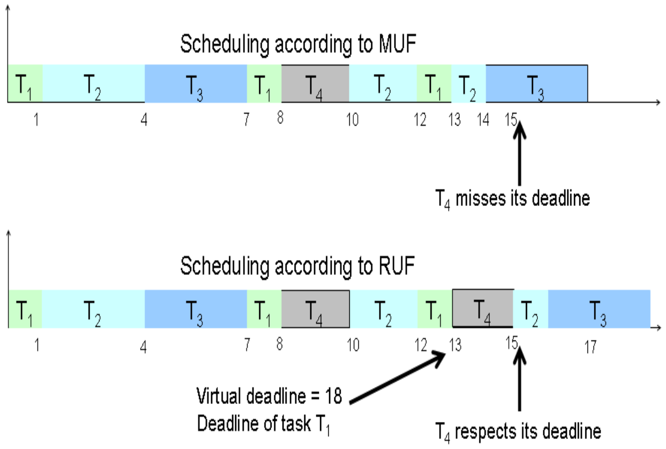


FIGURE 2.8: Example comparing MUF and RUF.

Here in this example (Fig:2.8), RUF algorithm finds at instant $t = 13$*, that available time_slot is sufficient enough to add* T_4 *into ready queue. The insertion of this non critical task* T_4 *does not make critical tasks to miss their deadlines, as we assign a virtual deadline equal to deadline of* T_4*. This virtual deadline is equal to latest deadline of all tasks that has contributed to time_slot. In this case, it is* 18 *(deadline of task* T_1*). So at* $t = 13$*,* T_4 *has higher priority over* T_3 *and* T_2*, as (virtual) deadline of* T_4 *is before than that of* T_2 *and* T_3*.*

2.6 Conclusions

In this chapter, we have presented a novel scheduling algorithm called RUF, which improves QoS by executing more non critical tasks as compared to other approaches of same domain. The performance of the RUF was compared to the well known MUF algorithm, and showed to be superior. We have not compared our algorithm with CASH as CASH does not provide guarantees to critical tasks, if processor utilization of critical and non critical task comes out to be more than 100%.

Scheduling on multiprocessor system using a global scheduling approach is similar to scheduling of tasks in a mono-processor system. The RUF scheduling algorithm could be a good candidate for a global scheduler, since a single system-wide priority space is considered in RUF as well. Future work could be to investigate the capability of RUF algorithm to schedule critical and non critical tasks efficiently on a multiprocessor architecture.

Chapter 3

Power Efficient Middleware

3.1 Introduction

Scheduling theory has been widely studied in the last twenty years. Since a milestone in the field of hard real-time scheduling pioneered by Liu et al.[48], two classes of algorithms (i.e., dynamic priority and fixed priority) have been studied separately and a lot of results such as optimality, feasibility condition, response time, admission control have been established.

The Earliest Deadline First (EDF) [48] scheduling algorithm is a dynamic priority algorithm which always executes a task whose deadline is the most imminent. This algorithm has been proved to be optimal in case of mono processor system [48]. At the instant of release of a higher priority task, EDF scheduling algorithm preempts the active task and starts executing the newly released higher priority task. It does not whether it is possible to delay the execution of higher priority task and let the low priority task completes its execution. Thus, EDF may, unnecessarily, increase task preemptions.

Reducing the number of preemptions can also be beneficial from energy point of view in systems with low power consumption demands. When a task is preempted, there is a strong probability that its contents in the cache are partially or totally lost. When the execution of the task is again resumed, it causes a lot of energy consuming accesses to off-chip memory. An access to off-chip memory is typically 10-100 times more expensive than that of an on-chip cache access in terms of energy consumption. Reducing number of preemptions reduces these additional expensive memory accesses by reducing the cache pollution.

Radu Dorbin [25] has proposed an algorithm, which minimizes preemptions for fixed priority scheduling but it does not address the dynamic scheduling algorithms. Sung-Heun [57] proposed an algorithm, which solves the problem of laxity tie appearing between two tasks at run time. This laxity tie introduces serious problem which made LLF impractical. But this algorithm does not benefit from the slackness present in each task. Woonseok Kim

[41] increases frequency of processor to minimize the number of preemptions. It reduces the energy consumption by minimizing the number of preemptions but on the other hand it increase energy consumption by operating processor at higher frequencies. Sanjoy K. Baruah [15] proposed an algorithm where basic idea of minimizing preemptions is similar but technique to calculate the No Preemption Zone(NPZ) parameter of a task is different. It does not work if tasks leave or join the system dynamically. Moreover, this work does not deal with dynamic calculation of NPZ. NPZ for a task calculated dynamically can have a value greater than or equal to NPZ calculated statically thereby further reducing the preemptions. We also observed that preemptions of tasks also depends upon the load of processors. If processor load is low then number of preemptions are fewer too. DVFS (Dynamic voltage and frequency scaling) techniques decrease the frequency of processor if processor load is not 100%, to minimize energy consumption but decreasing the frequency of processor has an effect of increasing processor load thereby increasing preemptions of tasks.

We propose modification in the existing DVFS algorithms to minimize the number of preemptions and frequency switching points. We intend to decrease the frequency of the processor only at those instants when it has no impact on number of preemptions. Numbers of preemptions are fewer if static/dynamic charge of the processor is low. Since dynamic power is a quadratic function of the voltage, reducing the supply voltage effectively minimize the dynamic power consumption [19, 26, 29–31, 42, 50, 62–64, 66, 67, 76].

In terms of reducing overall energy consumption, many newly developed scheduling techniques, e.g. [36–39, 55, 68], are constructed based on the DVFS schedule. For example, Yan et al. [77] proposed to first reduce the processor speed such that no real-time task misses its deadline and then adjust the voltage supply and body biasing voltage based on the processor speed in order to reduce the overall power consumption. Irani et al. [36] showed that the overall optimal voltage schedule can be constructed from the traditional DVFS voltage schedule that optimizes the dynamic energy consumption.

Pillai et al [65] has proposed two approaches. First, cycles conserving DVFS minimizes energy cost but it increases unnecessarily switching points. Second, look-ahead approach reduces the switching points but complexity is high to analyze the deferred work and to calculate slow down factor.

From these remarks, we propose that frequency of the processor should not be changed at all those instants when dynamic load is less than static load of the system. We suggest to decrease the frequency of the processor only at those instants when it does not increase the number of preemptions. This approach not only decreases the number of preemptions but also decreases the switching points and hence the energy consumption. In this chapter, we propose two techniques to minimize the energy consumption of the systems:

1. Minimizing number of preemptions of tasks

2. Minimizing number of switching points where frequency of the processor is changed

We define another parameter of a task called No Preemption Zone (NPZ) that helps minimizing preemptions. NPZ_i of Task T_i is the time duration for which the current running task T_i can keep on executing even if there exists a task (or tasks) whose dynamic priority is higher than that of current running task T_i. In the task set τ, tasks are sorted based on their period lengths in an ascending (non-decreasing) order, i.e., the period of T_i is smaller than that of T_{i+1}.

3.2 Minimizing number of preemptions

Most scheduling algorithms are priority-based: they assign priorities to the tasks or jobs in the system and these priorities are used to select a job for execution whenever scheduling decisions are made.

Dynamic-priority algorithms allow more flexibility in priority assignments; a task's priority may vary across jobs or even within a job. An example of a scheduling algorithm that uses dynamic priorities is the earliest-deadline-first (EDF) algorithm. Static priority algorithm RM is considered as an optimal fixed priority scheduling algorithm, and it has very low run time complexity of algorithm, while EDF has optimal schedulable bound on mono-processor systems. That's why, we have considered these two algorithms to be modified such that number of preemptions are minimized. We start with investigating those points when a task T_i is preempted by a high priority task. We also identify those tasks that can potentially preempt a task T_i, and propose to use laxity of T_i to help delaying this preemption if possible, as laxity of a task gives the flexibility to delay it.

Theorem 3.1. *Laxity L_i of task T_i can be shifted anywhere in its time period length without causing Task T_i to miss its deadline.*

Proof. Task T_i will never miss its deadline if it is assigned C_i unit of processor time during time duration of length P_i. This implies that laxity of a task can be shifted anywhere during time period of Task T_i. □

Theorem 3.2. *Task T_i can be preempted only by the task which has time period less than that of T_i (Both for RM and EDF scheduling).*

Proof. Only higher priority tasks can preempt low priority tasks. In case of RM scheduling algorithm, tasks having lower periods have higher priorities than tasks having higher time periods. But in case of EDF scheduling algorithm, priorities are defined on the basis of closeness of absolute deadlines of the tasks. If Task T_i is preempted during its n^{th} instant by task T_j ready for k^{th} instant it implies that n^{th} instant of task T_i was released before k^{th} instant of task T_j(i.e., $r_i^n < r_j^k$). As Task T_i is preempted during its n^{th} instant by task T_j ready for k^{th} instant, it implies that deadline of task T_j is earlier than task T_i's

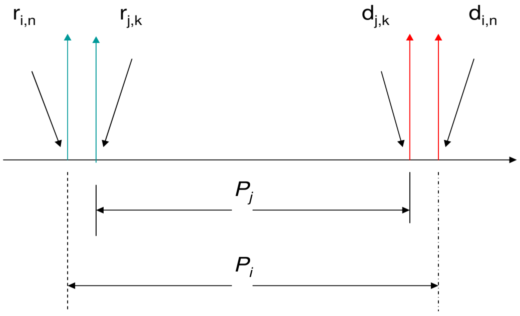


FIGURE 3.1: task having higher frequency can preempt low frequent tasks

deadline.
if

$$r_i^n < r_j^k$$

and

$$d_j^k < r_i^n$$

then

$$P_j < P_i$$

Hence it is proved that task T_i can be preempted by that task only which has time period less than time period of task T_i(Fig:3.1). □

We provide techniques to calculate NPZ of a task for both scheduling algorithms i.e., RM and EDF

3.2.1 Algorithm EEDF

We propose an algorithm aiming to reduce the number of preemptions. Preemptions are directly related with the Context switch. This algorithm works exactly like classical EDF scheduling algorithm except at those scheduling instants where preemption of active task is proposed by scheduler. If active (running) task is going to be preempted by newly arrived higher priority task, then this new algorithm EEDF (Enhanced EDF) comes into action and tries to figure out if it is really necessary to preempt this running task or preemption can be delayed for sometime i.e., NPZ.

This approach is similar to one proposed to by Baruah [15] where he has considered sporadic task set. According to this approach, each task is assigned another parameter called NPZ and preemption of the running task is delayed for time equal to NPZ. This approach presented by Baruah is based on static parameters of tasks and it does not support dynamic creation and deletion of tasks. In this approach, he has used the demand bound function to calculate the value of NPZ for each task and to prove the feasibility of task set. The demand bound function which provides sufficient and necessary condition for periodic tasks to be EDF-feasible, is defined as follows:

$$\sum_{i=1}^{n} DBF(T_i, t) = \sum_{i=1}^{n} \left\lfloor \frac{t}{P_i} \right\rfloor \times C_i \tag{3.1}$$

sufficient and necessary condition for feasibility of periodic tasks is given below:

If

$$\forall \ \ t > 0 \ \ :: \ \ \sum_{i=1}^{n} DBF(T_i, t) \leq t$$

The calculations of NPZ according to approach of Baruah are carried out by the following equation.

$$NPZ_i = min\left(NPZ_{i-1}, d_i - \sum_{i=1}^{n} DBF(T_i, t)\right) \tag{3.2}$$

Tasks are sorted in an increasing order of their period lengths and NPZ_1 is defined to be equal to L_0. This approach is iterative and NPZ_i is calculated before NPZ_{i+1}. This approach is based on static parameters of tasks, that's why it does not support dynamic creation and deletion of tasks.

We propose to calculate the NPZ in such a way that it supports dynamic creation and deletion of tasks. We have two forms of our approach to calculate NPZ of each task, one is static and the other is dynamic. In static approach, we consider the worst case scenario when all high priority tasks (tasks which have smaller time periods than that of running task) arrives at same instant (which is quiet rare in real systems), while in dynamic approach we take into account only those tasks which have absolute deadlines earlier than that of running tasks.

Example 3.1. *The laxity of any task gives us a measure of the maximum time for which it can be delayed. The task having the smallest time period will never be preempted as no other task can have higher priority during its execution. The approach is demonstrated by considering a simple model comprising of three tasks T_1, T_2 and T_3 given in Table 3.1. Worst case execution time for T_1, T_2 and T_3 are 6, 4 and 9 respectively while 21, 10 and 31 are their time periods of task respectively. Deadlines of the tasks are assumed to be equal to their periods.*

The task having the smallest time period is never preempted. In this example, T_2 has the smallest time period. Task T_1 can only be preempted by T_2 as only task T_2 has time period less than that of T_1. In this case T_1 can cause T_2 to go in priority inversion for time equal to at most laxity of task T_2 i.e., $NPZ_1 = P_2 - C_2$. Task T_3 can be preempted by both tasks T_1 and T_2 as both have time period less than that of task T_3. In this case,

Task	C	P	Offset	NPZ
T_1	6	21	0	6
T_2	4	10	3	is never preempted
T_3	9	31	0	6

TABLE 3.1: Task Parameters

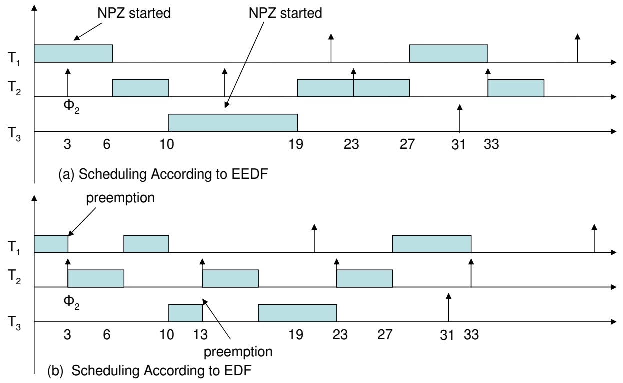


FIGURE 3.2: complete Example

NPZ for T_3 is calculated as follows:

$$L_{abs} = P_1 - \sum_{i=1}^{2} \frac{P_1}{P_i} \times C_i$$

$$L_{abs} = 21 - 6 - \frac{21}{10} \times 4$$

$$L_{abs} = 21 - 6 - 8.4$$

$$L_{abs} = 6.6$$

$$NPZ_3 = min(NPZ_1, L_{abs})$$

$$NPZ_3 = 6$$

We observe that there are two preemptions (Fig:3.2(b)) if scheduled according to EDF scheduler but preemptions are reduced to zero (Fig:3.2(a)) if preemption is delayed for their NPZ time values.

NPZ of task T_i can be calculated either statically or dynamically in order to minimize preemptions.

3.2.1.1 Static Computation

As explained earlier, laxity L_i of Task T_i can be shifted anywhere in its period such that lower priority tasks can continue with their executions. This laxity of task T_i defines the time duration (NPZ_{i+1}) for which its lower priority task T_{i+1} can continue its execution without making T_i to miss its deadline.
For Task T_2,

$$NPZ_2 = L_1 \tag{3.3}$$

Tasks T_1 and T_2 are projected in an abstract task T_{abs}, which has time period equal to task T_2 to calculate NPZ for task T_3. (The WCET of this abstract task is the sum of WCET of T_2 and proportional WCET of T_1 for period length equal to the time period of this abstract task). Then:

$$NPZ_3 = \min(NPZ_2, L_{abs})$$

This abstract task is projected into another abstract task having time period equal to the time period of task T_3 to calculate NPZ for task T_4, $NPZ_4 = \min(NPZ_3, L_{abs})$ This is done because only task T_3 have the latest deadline among all tasks (T_1, T_2, T_3) that can preempt task T_4. This is repeated until NPZ of each task is calculated. NPZ_i of a task T_i depends upon either on laxity of abstract tasks or on NPZ_j of task T_j whose time period is shorter than that of T_i. Abstract task is explained as follows:

Abstract Task T_{abs}: Task T_i can be preempted by a number of tasks i.e., $\{T_1, T_2, ..., T_{i-1}\}$. Task T_i can be preempted by all those tasks having time period less than T_i's time period. In this case task T_{i-1} has the highest period among all those tasks that can preempt T_i. We will project all those tasks that can preempt task T_i into a single task T_{abs} (abstract task).

Worst Case Scenario: To calculate the worst case execution time of this abstract task T_{abs}, the worst case scenario is identified. Worst case for task T_i is that all higher priority tasks are released at the same time during the execution of task T_i. This abstract task T_{abs} has time period P_{abs} equal to the time period of task T_{i-1} while worst case execution time C_{abs} is:

$$C_{abs} = \sum_{k=1}^{i-1} \frac{P_{i-1}}{P_k} \times C_k \tag{3.4}$$

$$L_{abs} = P_{abs} - C_{abs} \tag{3.5}$$

Task T_{abs} represents all those tasks, which have time periods equal to or less than that of task T_i. NPZ_i for task T_i is calculated considering only laxity of task T_{abs}, may cause any task to miss its deadline if laxity of this task is smaller than L_{abs} calculated for task T_i(lets say task that T_k having time period smaller than that of T_i). Therefore NPZ_i is

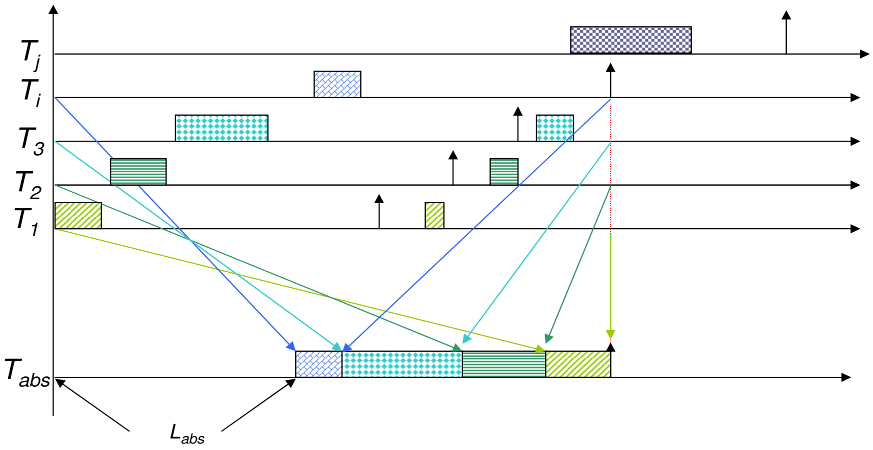


FIGURE 3.3: Calculation of NPZ_j

calculated recursively and is defined as:

$$NPZ_i = \min(NPZ_{i-1}, L_{abs}) \tag{3.6}$$

The pseudo code for calculation of NPZ statically is presented in Algorithm given below:

Algorithm 4 Static calculations of NPZ in EEDF approach

```
tasks sorted in increasing order of their period
for $i = 1$ to $n$ do
    $P_{abs} = d_{i-1}$;
    $C_{abs} = \sum_{k=1}^{i-1} \frac{P_{i-1}}{P_k} \times C_k$;
    $L_{abs} = P_{abs} - C_{abs}$;
    $NPZ_i = \min(NPZ_{i-1}, L_{abs})$;
end for
```

Dynamic Creation and Deletion of tasks: Our proposed approach allows dynamic creation and deletion of tasks. If, at time instant t, a task T_j is executing in presence of high priority task T_i but T_i is deleted at this instant and is replaced by another task T_k (with $\mu_k \leq \mu_i$ and $P_k \geq P_i$), then T_j can still keep on executing without causing other tasks to miss their deadlines.

3.2.1.2 Dynamic Computations

NPZ calculated statically minimizes number of preemptions but it can even be further enhanced with little modifications during run time. We consider the worst case scenario to calculate the NPZ of each task where we assume that all low periodic tasks preempt high periodic task T_i, which rarely happens at run time in case of EDF scheduling.

Algorithm 5 Dynamic calculations of NPZ in EEDF approach

```
tasks sorted in increasing order of their period
for $i = 1$ to $n$ do
    $P_{abs} = d_{i-1}$;
    $C_{abs} = \sum_{T_k \in \tau_r(t)} \left\lfloor \frac{d_{i-1} - \phi_k}{P_k} \right\rfloor C_k$;
    $L_{abs} = P_{abs} - C_{abs}$;
    $NPZ_i = \min(NPZ_{i-1}, L_{abs})$;
end for
```

Parameters of Preempting task: A task T_i can preempt a task T_j if and only if the $d_i < d_j$. A task T_i may not preempt a task T_j even if $P_i < P_j$, this case occurs when absolute deadline of T_i is later than that of T_j.

Best Case Scenario: The value NPZ_i of a task T_i is calculated by the following equation:

$$NPZ_i = \min(NPZ_{i-1}, L_{abs})$$

If $\min(NPZ_{i-1}, L_{abs}) = L_{abs}$, It implies that calculation of NPZ_i of task T_i depends upon the execution times of all those tasks which have time period less than time period of task T_i. If NPZ_i of a task T_i is dictated by laxity of one task, then the release of this task enforces dynamic calculations of NPZ_i to be equal to static calculations of NPZ_i. But, if NPZ_i depends on more than one task, then their release times have significant effect on calculation of NPZ_i. Moreover, we have considered in case of static computations that all low periodic tasks are ready at the same time which is very rare in any type of scheduling. Preemption may further be minimized by taking into account all these run time parameters.

Dynamic calculations of NPZ_i are carried out in a same way as in case of static calculations but taking into account runtime parameters instead of off line parameters. NPZ_i of running task is calculated by considering only those tasks which have absolute deadline earlier than that of T_i. When a task having absolute deadline before the absolute deadline of running task is released, NPZ_i is set equal to the laxity of the released task. If another task is released and its absolute deadline is also before the absolute deadline of T_i, then NPZ_i is recalculated. The recalculated value of NPZ_i can be less than or equal to the value of NPZ_i calculated earlier.

The task having the most imminent absolute deadline is considered as the first task (instead of the most frequent task) and task having latest absolute deadline but earlier than that of the T_i is considered last such task that can preempt running task. This is different from the static calculation, where most frequent task is considered as the first task and the task having the largest period but smaller than that of T_i is considered last task to calculate NPZ_i.

Dynamic Creation and Deletion of tasks: If NPZ_i of task T_i is calculated at run time, then tasks can leave or join the system without any condition imposed but the feasibility test.

3.2.1.3 Feasibility Analysis

A task set remains feasible even if a lower priority task keeps on executing for time equal to NPZ.

Theorem 3.3. *The Enhanced Earliest Deadline First (EEDF) algorithm determines a feasible schedule if* $U(\tau) \leq 1$.

Proof. Our Enhanced Earliest Deadline First (EEDF) algorithm does not invalidate the classic EDF theorem on feasibility. A low priority task is allowed to continue its execution in presence of higher priority tasks if and only if all higher priority tasks have laxity greater than zero.

We consider a set of task $T = \{T_1, T_2, ..., T_n\}$ where period of T_2 is greater than period of T_1. This task set if scheduled by EDF always respects its deadline (if $U(\tau) \leq 1$). We prove that with modification introduced in form new preemptions zone concept , all tasks still respect their deadlines. We use the demand bound function [16] analysis to prove the feasibility of scheduling algorithm. Demand bound function $DBF(T_i, t)$ defined by Baruah is an approach to establish the necessary and sufficient condition for feasibility analysis of tasks over interval $[0, t)$.

$$\sum_{i=1}^{n} DBF(T_i, t) = \sum_{i=1}^{n} \left\lfloor \frac{t}{P_i} \right\rfloor \times C_i \tag{3.7}$$

If

$$\sum_{i=1}^{n} DBF(T_i, t) \leq t$$

then task set is schedulable.
if $t = P_k$ then:

$$\left\lfloor \frac{P_k}{P_0} \right\rfloor \times C_0 + \left\lfloor \frac{P_k}{P_1} \right\rfloor \times C_1 +, ... \left\lfloor \frac{P_{k-1}}{P_0} \right\rfloor \times C_{k-1} + C_k \leq P_k$$

If the same task set is scheduled by the enhanced algorithm it respects its deadline as well. To support this statement we start with considering that introduction of NPZ_j of task T_j causes task T_k to miss its deadline. $(P_j > P_k) \Rightarrow$

$$NPZ_j + \left\lfloor \frac{P_k}{P_0} \right\rfloor \times C_0 + \left\lfloor \frac{P_k}{P_1} \right\rfloor \times C_1 +, ... \left\lfloor \frac{P_{k-1}}{P_0} \right\rfloor \times C_{k-1} + C_k > P_k \tag{3.8}$$

NPZ_{k+1} of task T_{k+1} is given below $(k < (k+1) \leq j)$

$$NPZ_{k+1} = \min\left(NPZ_k, P_k - \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i \right) \tag{3.9}$$

$$\sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i \geq \left\lfloor \frac{P_k}{P_0} \right\rfloor \times C_0 + \left\lfloor \frac{P_k}{P_1} \right\rfloor \times C_1 +, \ldots \left\lfloor \frac{P_{k-1}}{P_0} \right\rfloor \times C_{k-1} + C_k$$

Replacing right hand side with left hand side in inequality 3.8

$$NPZ_j + \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i > P_k$$

As $j \geq k+1$ it implies that

$$NPZ_j \leq NPZ_{k+1}$$

and replacing NPZ_j with NPZ_{k+1} gives:

$$NPZ_{k+1} + \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i > P_k \tag{3.10}$$

Let us assume that

$$P_k - \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i \leq NPZ_k$$

then by definition of Equation(3.6)

$$NPZ_{k+1} = P_k - \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i$$

replacing NPZ_{k+1} in inequality 3.10 $\Rightarrow$

$$P_k - \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i + \sum_{i=0}^{k} \frac{P_k}{P_i} \times C_i > P_k$$

As $P_k > P_k$ is not possible, we get:

$$NPZ_j + \left\lfloor \frac{P_k}{P_0} \right\rfloor \times C_0 + \left\lfloor \frac{P_k}{P_1} \right\rfloor \times C_1 +, \ldots \left\lfloor \frac{P_{k-1}}{P_0} \right\rfloor \times C_{k-1} + C_k \leq P_k$$

It proves that task set τ having $U(\tau) \leq 1$ is schedulable according to EEDF. □

3.2.1.4 Experimental Results of EEDF

We have compared our algorithm EEDF with classical EDF and we performed simulations on CoFluent tool, and we have measured the number of preemptions both in case of static calculations of NPZ and in case of dynamic calculations as well. We have considered different number of tasks to compare our results, and parameters of these tasks are generated randomly. Sum of utilization of all tasks in every set of tasks is 98%. We can observe in (Fig:3.4) that EEDF algorithm has significantly reduced the number of preemptions. We have illustrated that these preemptions are further reduced if we make our

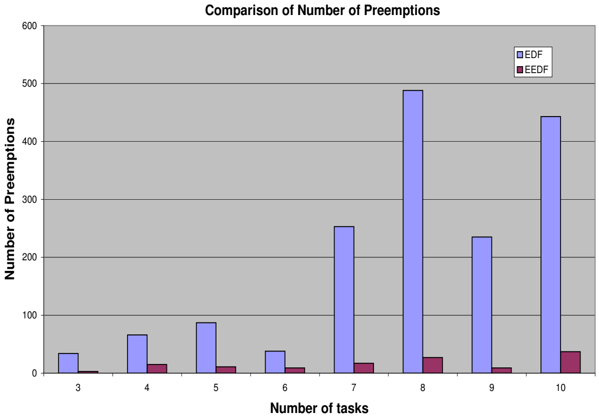


FIGURE 3.4: Comparison of Number of Preemptions

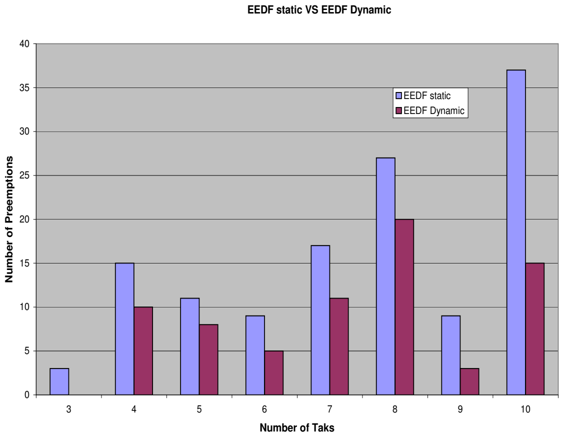


FIGURE 3.5: EEDF static Vs EEDF dynamic

calculations dynamically and in a more realistic manner Fig:3.5. We also observed that both algorithms i.e., EDF and EEDF have fewer preemptions of tasks if load of processor is decreased Fig:**??**.

3.2.2 Algorithm ERM

Rate Monotonic (RM) scheduling algorithm is a simple rule that assigns priorities to tasks according to their rates. Specifically, tasks with higher requests rates (that is, with shorter periods) have higher priorities. Since periods are constant, RM is a fixed priority assignment: priorities are assigned to tasks before execution and do not change over time. Moreover, RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with shorter period.

We define a parameter NPZ_i of task T_i, as it is defined in earlier section.

3.2.2.1 Feasibility Analysis of RM

Liu and Layland [48] proved that the worst case phasing occurs when all tasks release at same time i.e., at zero. This is called a critical instant in which all tasks are simultaneously instantiated. Using this concept, Liu and Layland also proved that task set is schedulable using the rate monotonic algorithm if the first job of each task can meet its deadline when it is initiated at critical instant.

For rate monotonic scheduling, the sufficient schedulable bound for n tasks has been shown to be the following:

$$U(\tau) = n(2^{1/n} - 1)$$

$U(\tau)$ asymptotically converges to $ln(2)$ or 69%, which is less efficient than some run-time schedulers such as earliest deadline. The utilization bound test allows schedulability analysis by comparing the calculated utilization for a set of tasks and comparing that total to the theoretical utilization for that number of tasks. If this equality is satisfied, all of the tasks will always meet their deadlines. If the total utilization is between the utilization bound and 100%, the utilization bound test is inconclusive and a more precise test must be used. Richard and Goossens [60] established a near optimal schedulable bound for static priority scheduling algorithm where release jitter of a task is considered as well, but it is not optimal. The more inclusive test to provide deadline guarantees is response time analysis. The response time (RT) test allows analysis of schedulability based upon the following lemma:

Lemma 3.4. *For a set of independent periodic tasks, if each task meets its deadline with worst case task phasing, the deadline will always be met [56].*

The RT test requires computation of the response time of each task in the system. Based on the above lemma if each response time is less than its corresponding period, the system is schedulable.

To determine if a task T_i can meet its deadline under the worst case phasing, the processor demand made by the task set is considered as a function of time. If we focus on tasks T_1, T_2,...,T_i, then the expression:

$$W_i(t) = \sum_{j=1}^{i} C_j \times \left\lceil \frac{t}{P_j} \right\rceil$$

gives the cumulative demands on the processor made by tasks over[0,t], when zero is a critical instant. Lui sha et all [45] has developed a condition if verified provides guarantees to all tasks. They introduced few following notations:

$$Z_i(t) = W_i(t)/t,$$

$$Z_i = min_{\{0<t\leq P_i\}} Z_i(t),$$

$$Z = max_{\{1\leq i\leq n\}} Z_i$$

Lui sha et al.[45] has proved the following corollary:

Corollary 3.5. *[45] Given periodic tasks* $T_1, T_2, ..., T_n$

1. *T_i can be scheduled for all task phasings using the rate monotonic algorithm if and only if $Z_i \leq 1$*

2. *The entire task set can be scheduled for all task phasings using the rate monotonic algorithm if and only if $Z \leq 1$*

We consider these schedulability conditions to compute statically and dynamically values of NPZ_i.

Static Calculations of NPZ_i: NPZ_i of task T_i defines the largest value such that τ remains feasible even if T_i is not preempted for this time at the release of high priority tasks. To calculate NPZ_i of task T_i, let us define another parameter PIV_i, which defines the largest value such that T_i remains schedulable even if task T_k, having lower priority than T_i, executes for time equal to PIV_i.

$$PIV_i = max_{\{0<t\leq P_i\}} \left[max(0, t - W_i(t))\right]$$

NPZ_{i+1} of task T_{i+1} such as $P_i \leq P_{i+1}$, is calculated by the equation given below:

$$NPZ_{i+1} = min(NPZ_i, PIV_i)$$

Algorithm 6 Static calculations of NPZ in ERM approach

```
tasks sorted in increasing order of their period
$NPZ_1 = PIV_1 = P_1 - C_1$;
$NPZ_2 = \min(NPZ_1, PIV_1)$;
for $i = 2$ to $n - 1$ do
    $W_i(t) = \sum_{j=1}^{i} C_j \times \left\lceil \frac{t}{P_j} \right\rceil$;
    $PIV_i = max_{\{0<t\leq P_i\}} [max(0, t - W_i(t))]$;
    $NPZ_{i+1} = \min(NPZ_i, PIV_i)$;
end for
```

Dynamic Calculations of NPZ_i: In the above section, NPZ_i of task T_i is calculated considering the worst case scenario which is worst case phasing. If task T_{i-1} is not released

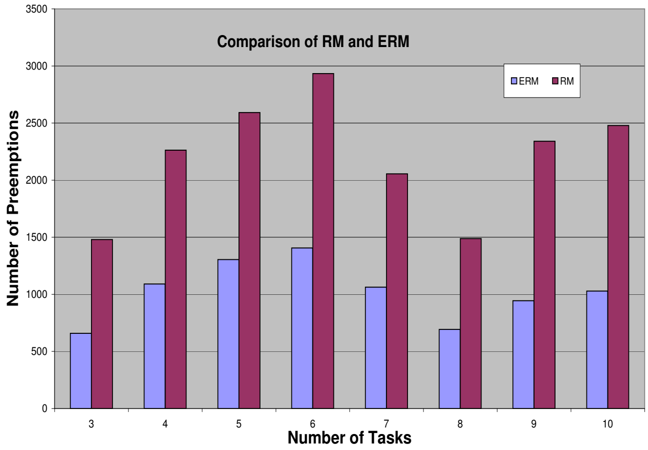


FIGURE 3.6: Comparison of Number of Preemptions

at the same time when task T_i is released and T_{i-1} has finished its execution before release of task T_i, then number of instants of T_{i-1} executing during the current instant of T_i can be one less than calculated during the worst case phasing. Dynamic calculations of NPZ_i of task T_i are carried out only if high priority task is released and run time parameters of tasks are taken into account to calculate it. Let us define that τ_r represents the set of ready tasks and $P_i \geq P_{i-1}$ then cumulative demand is calculated as follows:

$$W_{i-1}(t) = \sum_{\substack{j=1 \\ T_j \in \tau_r}}^{i-1} C_j \times \left\lceil \frac{d_{i-1} - r_j}{P_j} \right\rceil$$

NPZ_i of a task is updated if another high priority task is released during the execution of task T_i.

3.2.2.2 Experimental Results of ERM

We have compared our algorithm ERM with classical RM and we have performed simulations on STORM[1] tool, and we have measured the number of preemptions both in case of RM and ERM. We have considered different number of tasks to compare our results, and parameters of these tasks are generated randomly. Sum of utilization of all tasks in every set of tasks also varies between 70% to 100%. Calculations of NPZ for ERM based schedule task is extracted from feasibility test of task, and if task set is not schedulable then simulations are not performed. We can observe in (Fig:3.6) that number of preemp-

[1] STORM is a **S**imulation **TO**ol for **R**eal time **M**ultprocessor scheduling developed by IRCCyN in context of PHERMA project web: http://pherma.irccyn.ec-nantes.fr/

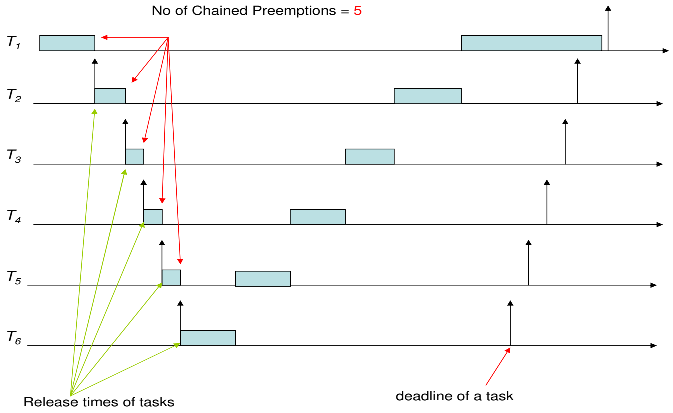


FIGURE 3.7: Phenomena of chained preemptions in EDF scheduling

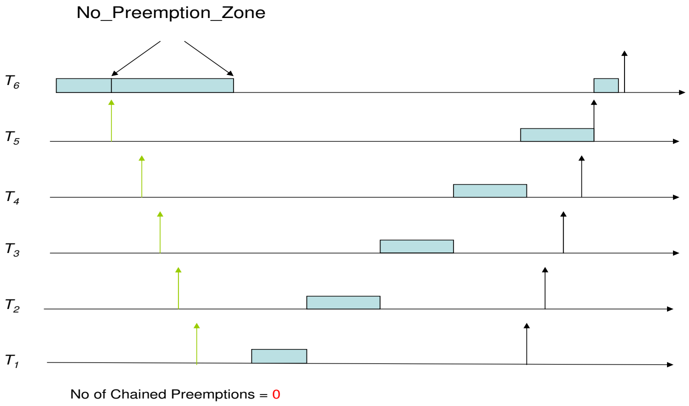


FIGURE 3.8: Phenomena of chained preemptions in EEDF scheduling

tions of task in case of RM schedule tasks are almost 2.5 times higher than those if tasks are scheduled according to ERM algorithm.

3.2.3 Chained Preemptions

Chained preemptions is a phenomenon, which increases the number of preemptions to a great extent in case of RM and EDF scheduling algorithms. This is a phenomenon, where one task is preempted by newly released higher priority task, then this new active task is also preempted by another release of a higher priority task (Fig:3.7). In the worst case, these chained preemptions can be equal to the number of tasks in a task set. NPZ_i introduced for each task can significantly minimize this phenomenon (Fig:3.8).

EEDF and ERM algorithms efficiently minimize the number of preemptions and hence energy of the system. EEDF and EDF both introduce less preemptions if processor load is low. But, DVFS algorithms reduce the operating frequency of the system leading to an increase in the processor load and hence increases preemptions.

3.3 Minimizing switching Points and Preemptions

DVFS mechanism may reduce the operating frequency and voltage when tasks use processor time less than their worst-case time allotment. When the task completes, actual processor time is compared with the worst-case execution time. Any unused computation time that was allotted to the task would normally (or eventually) be wasted, idling the processor. Instead of idling for extra processor time, DVFS algorithms are used that avoid wasting cycles by reducing the operating frequency for subsequent ready tasks. These algorithms are tightly-coupled with the operating system's task management services, since they may need to reduce frequency on task completion, and increase frequency on task release. These approaches are pessimistic as they reduce frequency of the processor right after the completion of the task (if $C_i > AET_i$), and increase the frequency of the processor back to normal when a recently finished task is released again for next instant. These algorithms assume that these extra cycles are wasted if frequency is not decreased right after the completion of a task. Switching from one frequency level to another level takes processor time and uses system energy, and hence the net effect could as increase in power consumption of the system.

We propose an algorithm EDVFS where we try to minimize the frequency switching points. We propose to accumulate the cycles $(C_i - AET_i)$ and don't decrease frequency until a point after which these cycles are wasted -idling the processor- if frequency of the processor is not decreased.

In(Fig:3.9),we illustrate our approach how the number of switching points is decreased. In Fig:3.9(a), frequency of processor for task T_2 is decreased because T_1 has used fewer processor time than C_1. When task T_2 finishes its execution, frequency for task T_3 is decreased again (as task T_2 has also used fewer processor cycles than C_2). Frequencies are restored again at instants when tasks T_1 and T_2 are released for their next jobs. In Fig:3.9(c), we illustrate that frequency for task T_2 is not decreased even if AET_1 of task T_1 is smaller than C_1. Frequency for task T_3 is decreased because there is an idle time on processor (Fig:3.9(b)), before d_1 if frequency is not decreased for task T_3.

3.3.1 Identification of switching points

Whenever a task finishes its execution, the number of ready tasks is tested in the ready queue of the scheduler, and if it is more than one then the frequency of the processor is

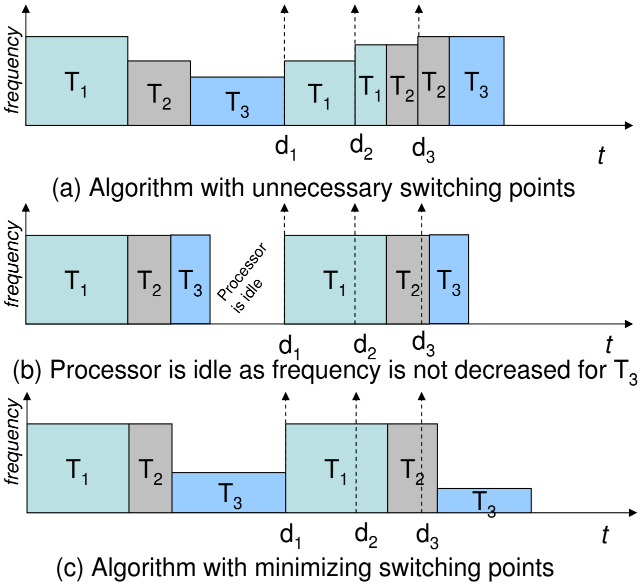

frequency
T1
T2
T3
T1
T1
T2
T2
T3
d1
d2
d3
t
(a) Algorithm with unnecessary switching points
frequency
T1
T2
T3
Processor is idle
T1
T2
T3
d1
d2
d3
t
(b) Processor is idle as frequency is not decreased for T3
frequency
T1
T2
T3
T1
T2
T3
d1
d2
d3
t
(c) Algorithm with minimizing switching points

FIGURE 3.9: Comparison of two approaches

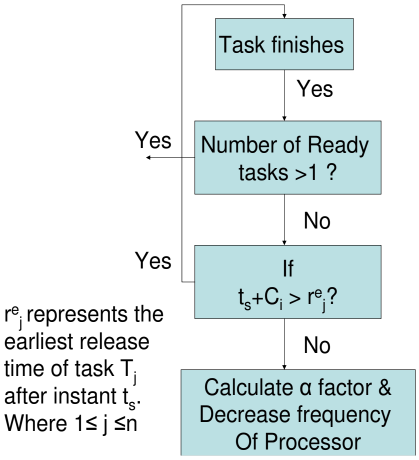

Task finishes
Yes
Yes
Number of Ready tasks >1 ?
No
Yes
If $t_s+C_i > r^e_j$?
r^e_j represents the earliest release time of task T_j after instant t_s. Where 1≤ j ≤n
No
Calculate α factor & Decrease frequency Of Processor

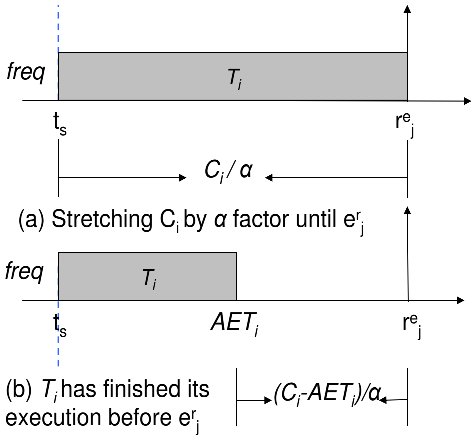
FIGURE 3.10: Approach Description (Flow Diagram)

not decreased. If there is single task in the ready queue, then subsequent calculations for identification of switching points are performed (Fig:3.10).

The principle is to change the frequency of the processor only at those instants after which processor goes idle if frequency is not decreased. According to our approach, frequency of the system is decreased at time t_s when there is only one ready task T_i and:

$$t_s + C_i^{rem}(t_s) < r_j^e$$

Value r_j^e denotes the earliest release time of task T_j, $1 \leq j \leq n$, after time instant t_s. At this time instant t_s, the frequency of the processor is decreased to extend the execution of task T_i until the r_j^e.



(a) Stretching C_i by α factor until e^r_j

(b) T_i has finished its execution before e^r_j

Figure 3.11: Idle time on the processor

3.3.2 Calculation of the frequency slow down (α) factor

Once appropriate switching point is identified, frequency of the processor is decreased by a factor of α ($\alpha < 1$) which is calculated in the following way:

$$\alpha = \frac{C_i^{rem}}{r_j^e - t_s}$$

$$f_{new} = \alpha \times f$$

3.3.3 Processor Idling

Actual execution time of task T_i may vary from B_i processor cycles to C_i processor cycles.

If frequency of the processor is decreased, considering that task T_i will take C_i time units, then a lot of processor time will be unused if its AET_i appears to be much smaller than its C_i (Fig:3.11 b). In worst case, processor may go idle for processor time equal to $(C_i - B_i)/\alpha$.

3.3.3.1 Minimizing Processor Idle Time

There is a need to define an approach such that wasted time ($(C_i - B_i)/\alpha$) on processor is minimized. We propose to calculate the slow down factor by considering B_i instead of C_i.

In this case slow down factor α is calculated as:

$$B_i^{rem} = B_i - C_i^{completed}$$

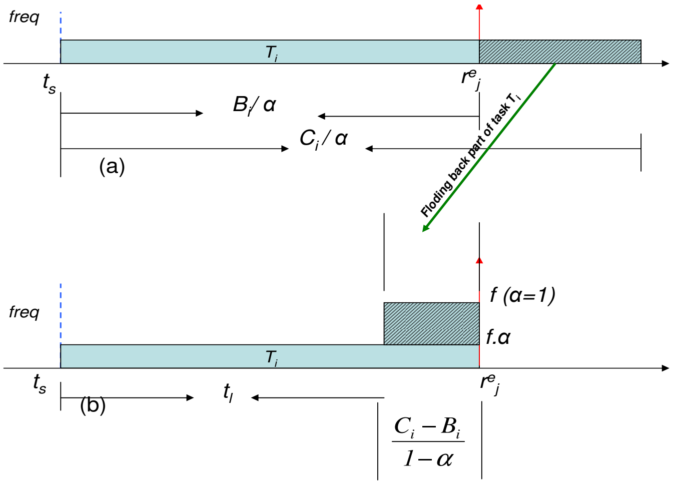


FIGURE 3.12: Folding execution time

B_i^{rem} represents the remaining execution time of task T_i while considering that T_i will take B_i time units to execute.

$$\alpha = \frac{B_i^{rem}}{r_j^e - t_s}$$

$$f_{new} = \alpha \times f$$

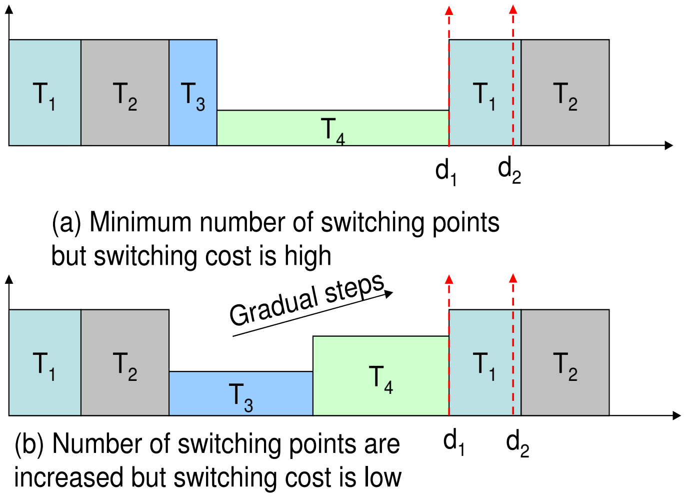
This may cause task T_i to miss its deadline if its AET_i happens to be more than B_i. To ensure deadline guarantees for task T_i, there is a need to increase the frequency back to normal value at time t_l before the earliest release time of task T_j, $1 \leq j \leq n$. This time is calculated by folding back a part of task T_i that was crossing the time r_j^e(Fig:3.12b).

$$t_l = r_j^e - t_s - \frac{C_i - B_i}{1 - \alpha}$$

Value t_l represents a time after which frequency of processors is restored to normal value i.e., $\alpha = 1$. This approach has one possible drawback which is the cost of switching frequency of processor from very low value to normal value (i.e., $\alpha = 1$).

3.3.3.2 Gradual Increase in Frequency

The main reason to change the frequency (increasing) in gradual steps is the DVFS switching cost, which includes both time and energy cost. Switching cost is proportional to the magnitude of the switch.



(a) Minimum number of switching points but switching cost is high

(b) Number of switching points are increased but switching cost is low

FIGURE 3.13: Gradual increase in frequency of the processor

We propose to change the frequency in such a way that switching cost is reduced. Switching cost is low when frequency of the processor is changed from a higher value to low value but cost is higher in case of transition from low to high frequency. Moreover switching cost also depends upon the size of step (difference between the current value of frequency and next value of frequency). That's why we propose to increase the frequency in gradual steps until frequency is restored to normal value (i.e., until slow down factor equals to 1). This approach is similar to the approach explained in the above section with only difference that frequency of the processor is decreased when there are two or more (auto adaptive) tasks in the ready queue of the scheduler. Slow down factor for first task (higher priority first task T_f) is selected to be higher than that for second task T_s. To achieve this, more accumulated cycles are allocated to first task than to second task.

$$
\begin{aligned}
CY_f = & \frac{C_f^{rem}}{C_f^{rem} + C_s^{rem}} \times (r_j^e - t_s - C_f^{rem} - C_s^{rem}) \\
& + 0.5 \times \frac{C_s^{rem}}{C_f^{rem} + C_s^{rem}} \times (r_j^e - t_s - C_f^{rem} - C_s^{rem}) \\
CY_s = & \, 0.5 \times \frac{C_s^{rem}}{C_f^{rem} + C_s^{rem}} \times (r_j^e - t_s - C_f^{rem} - C_s^{rem})
\end{aligned}
$$

CY_f represents the cycles allotted to higher priority task (T_f) and CY_s represents cycles allocated to task T_s.

$$\alpha_f = \frac{C_f^{rem}}{CY_f}$$
$$\alpha_s = \frac{C_s^{rem}}{CY_s}$$

Higher priority task T_3 (Fig:3.13) is allocated more cycles to keep process frequency lower during execution of task T_3 than that for task T_4. Frequency for task T_4 will be higher than that of T_3 and it will be restored to normal value ($\alpha = 1$) at r_j^e.

3.3.3.3 Self adaptive approach

If frequency of a processor can not be decreased more than its threshold value then a self adaptive approach can be used to decrease the frequency of the processor. In this case, the number of tasks which execute at low processor frequency is not fixed. Let us assume that f_{th} represents the threshold frequency and α_{th} is defined as follows:

$$f_{th} = \alpha_{th} * f$$

When a task finishes its execution at time t, and if:

$$\frac{\sum_{i=1}^{n} C_i^{rem}}{r_j^e - t} \leq \alpha_{th}$$

then the frequency of the processor for remaining tasks is decreased to f_{th} until r_j^e.

3.3.3.4 Experimental Results of EDVFS

We have illustrated (Fig:3.14) that numbers of switching points are dramatically reduced as compared to existing DVFS algorithms. We have demonstrated that if number of tasks in the system increases then the number of switching points increases as well. In this case, the factor by which number of switching points is reduced by our algorithm increases exponentially (almost). We have compared EDVFS algorithm with already proposed scheduling algorithms DVFS i.e., ccEDF (cycle conserving EDF) to evaluate the performance of this algorithm.

3.4 Conclusion

In this chapter, we have presented an approach for minimizing preemptions of tasks, as high number of preemptions in a schedule reduces the practically achievable schedulable

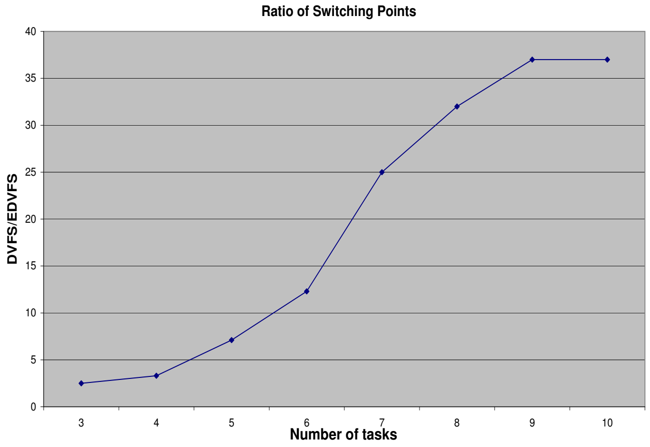


Figure 3.14: EDVFS Vs DVFS

bounds. We have proposed modifications both in EDF and RM scheduling algorithms which take care of implicit parameters of a task to efficiently cut down the number of preemptions. Moreover, we have also proposed to consider not only static implicit parameters of tasks but also runtime parameters of task (release times of task etc.) which helps reducing the preemptions of tasks to a greater extent. We have compared the performance of our algorithm in terms of reducing number of preemptions with EDF algorithm for hard real-time periodic tasks. Our comparative study (using software simulation) shows that numbers of preemptions are reduced exceptionally. We have also extended the approach of dynamic voltage and frequency scaling scheme, as this approach has it adverse effect on number of preemptions. Reduced number of preemptions not only increase the practical achievable schedulable bound but also decrease the energy consumption of the system. We have proposed to embed our approach with DVFS technique in such a way that number of preemptions are not increase, moreover, the frequency switching points are decreased to a great extent. This extension not only reduces the preemptions of the tasks but it also decreases the switching points.

Chapter 4

Efficient and Optimal Multiprocessor Scheduling for Real Time Tasks

4.1 Introduction

Multiprocessor scheduling techniques in real-time systems fall into two general categories: partitioning and global scheduling. Under partitioning, each processor schedules tasks independently from a local ready queue. Each task is assigned to a particular processor and is only scheduled on that processor for each instance of the task. In contrast, all ready tasks are stored in a single queue under global scheduling. A single system-wide priority space is assumed; the highest-priority task is selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled.

Presently, partitioning is the favored approach in embedded system theory. This is largely because partitioning has proved to be both efficient and reasonably effective when using popular uniprocessor scheduling algorithms, such as the earliest-deadline-first (EDF) and rate-monotonic (RM) algorithms. Producing a competitive global scheduler, based on such well-understood uniprocessor algorithms, has proved to be a daunting task. In fact, Dhall and Liu [24] have shown that global scheduling using either EDF (g-EDF) or RM can result in arbitrarily-low processor utilization in multiprocessor systems. Goossens et al [13] have detailed feasibility bounds for RM on multiprocessor architecture which are not optimal schedulable bounds. Partitioning, regardless of the scheduling algorithm used, has two primary flaws. First, it is inherently suboptimal when scheduling periodic tasks. Second, the assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense.

In contrast, the non-partitioning method has received much less attention [44, 49, 51, 52], mainly because it is believed to suffer from scheduling and implementation related shortcomings, also because it lacks support for more advanced system models, such as the management of shared resources. An other important factor that makes this approach of less interest is the use of a shared memory, which introduces the bottleneck for system scalability. Cost of preemption and migration is high if distributed shared memory is used. On the better side, schedulability bounds are much better than its counterpart.

The Pfair [2, 14, 34] class of algorithms that allows full migration and fully dynamic priorities has been shown to be theoretically optimal i.e., they achieve a schedulable utilization bound (below which all tasks meet their deadlines) that equals the total capacity of all processors. However, Pfair algorithms incur significant run-time overhead due to their quantum-based scheduling approach.

LLREF [22] is also based on the fluid scheduling model and the fairness notion. It is not based on time quanta but number of preemptions and migrations in this schedule are also high.

We propose an algorithm called Anticipating Slack Earliest Deadline First until zero laxity(ASEDZL), which is an extension to already proposed algorithm called EDZL [75]. Tasks are selected to execute between two consecutive task's release instants. At each task's release instant, laxity of M high priority tasks until next release of any task (high priority tasks according to EDF) is calculated and is anticipated before they appear by giving higher priority to sufficient number of subsequent tasks. Sufficient number of tasks -which are not first M (M represent the number of identical processors) high priority tasks according to EDF- are given higher priority.

In this chapter, we discuss multiprocessor scheduling where $\pi = \{\pi_1, \pi_2, ..., \pi_M\}$ represents the set of M identical processors. We introduce a parameter of a task T_i named as current weight denoted as u_i^c and is defines as follows:

$$u_i^c = \frac{C_i^{rem}(t)}{d_i^{rem}(t)}$$

When a task T_i is selected to execute at time t, its current weight u_i^c decreases, and current weight of a task increases if it is waiting (blocked) for processor.

We consider the following assumptions to design a multiprocessor scheduling algorithm:

- ***Job preemption is permitted:***

  A job executing on a processor may be preempted prior its completion, and its execution may be resumed later. Initially, we assume that there is no penalty associated with such preemption to prove optimality theoretically but this penalty will be taken into account later to compare practical results.

- ***Job migration is permitted:***

A job that has been preempted on a particular processor may resume its execution on a different processor. Once again, we assume that there is no penalty associated with such migration.(will considered for practical implementation)

- ***Job parallelism is forbidden:***

  The jobs of each task are required to be executed sequentially. That is, a job cannot start its execution before the completion time of the preceding job of the same task. If a job has been released but is not able to start executing because the preceding job of the same task has not yet completed, we say that the job is *precedence-blocked.* If a job has been released and its predecessor in the same task has completed, the job is ready. Moreover, each job may execute on at most one processor at any given instant in time. If a job is ready, but M jobs of other tasks are scheduled to execute, we say that the job is *priority-blocked.*

4.2 EDF based Scheduling Algorithms and their Utilization Bounds

Some researchers are addressing the problem of schedulability analysis of classical uniprocessor scheduling algorithms, like RM and EDF, on SMP systems (Symmetric Multiprocessing). Regarding schedulability analysis of periodic real-time tasks with EDF, Goossens Funk and Baruah [27] have recently proposed a schedulability test based on an utilization bound, assuming that tasks have relative deadlines equal to their periods. Baker [12] proposed a different test extending the model to tasks with deadline less than or equal to their periods. Anderson et al. [5] provided bounds to the utilization of feasible tasks sets scheduled with fixed priority.

The advantage of these schemes is the relatively simple implementation and the minor overhead in terms of number of context switches. However, many negative results are known for such schedulers. For example, EDF loses its optimality on multiprocessor platforms. We analyzed the uniprocessor EDF scheduling algorithm differently. EDF based scheduler selects the highest priority task to execute, and time equal to laxity of this task is used to execute next higher priority tasks. Let us assume that at time $t = 0$, EDF selects T_i to execute, then $L_i/P_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} C_j/P_j$ (if $U(\tau) = 1$), but EDF scheduler does not allocate processor time to all tasks in proportion to their weights until d_i. It allocates maximum time to next higher priority tasks, and so on. We propose to use the same idea of exploiting the laxity of the highest priority task, and to allocate maximum time to the next higher priority tasks instead of fluid scheduling model, as in Pfair base scheduling algorithms. It is observed that in case of mono-processor system, next higher priority tasks ensure their execution until next release instant, but this is not possible in case of multiprocessor system. Moreover, (in case of mono-processor system) when a task

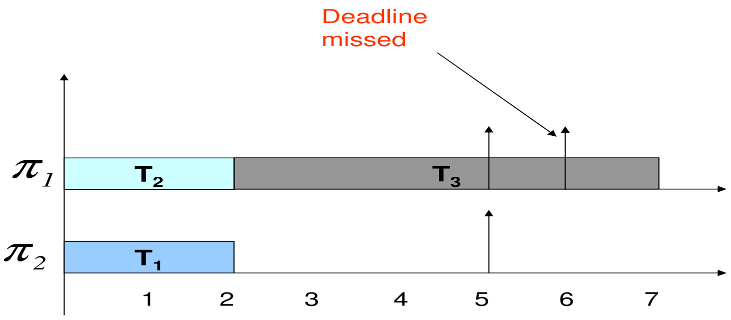


FIGURE 4.1: Dhall's Effect

claims zero laxity, it is the only ready task until its deadline. The same can be true in case of multiprocessor system as well, if we could ensure execution of next higher priority tasks until next release of a task. Then there will be only M tasks ready until the next release, if all M tasks have zero laxity. We further elaborate the reason for which g-EDF is not optimal scheduling algorithm.

1. If worst case execution requirement of high priority tasks is greater than laxity of other remaining task T_i, then T_i misses its deadline (Dhall's Effect).
2. Offloading factor of M high priority running tasks have accumulative effect on remaining tasks.

4.2.1 Dhall's Effect

Dhall and Liu [24] showed that global scheduling with optimal uniprocessor scheduling algorithms, such as EDF and RM, may result in arbitrarily low processor utilization.

Example 4.1. *To see why, consider the following synchronous periodic task system to be scheduled on M processors: M tasks with period p and execution requirement 2, and one task with period $p + 1$ and execution requirement p. At time 0, EDF favors the M tasks with period p. The task with period $p + 1$ does not get scheduled until time 2, by when its deadline cannot be guaranteed. (Fig:4.1 illustrates this problem for $M = 2$ and $p = 5$.) Note that the total utilization of this task system is $\frac{2M}{p} + \frac{p}{p+1}$; as p tends to ∞, this value tends to 1 from above. This effect can only be removed by taking another parameter into account (other than absolute deadline) to define the priority of tasks.*

- EDZL scheduling Algorithm

EDZL is a hybrid preemptive priority scheduling scheme in which jobs with zero laxity are given highest priority, and other jobs are ranked by deadline. The EDZL algorithm integrates EDF and LLF (Least Laxity First). EDZL uses EDF as long as there is no

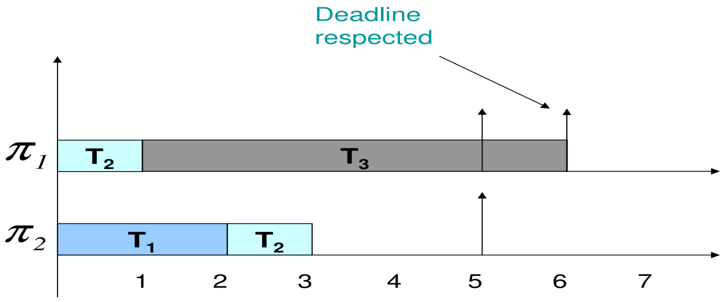


FIGURE 4.2: EDZL removes dhall's effect

urgent job (having zero laxity) in the system. When the laxity of a job becomes zero, a current job with a positive laxity that has the latest deadline among all current jobs is preempted by this job. The main idea of EDZL is shown in Fig:4.2. We take the same task set used to explain the dhall's effect, and we illustrate that how EDZL ensures scheduling of tasks. EDZL scheduling algorithm removes the dhall's effect by preempting a task if laxity of remaining task reaches a value of zero. As we illustrate in (Fig:4.1) that task set is not schedulable on two processors using g-EDF, but it is schedulable using EDZL as shown in Fig:4.2. At $t = 1$, T_3 becomes a zero-laxity job and preempts T_2. T_2 resumes execution at $t = 2$. All three jobs meet their deadlines, since at $t = 1$, the priority of T_3 is promoted to a higher level than T_1 and T_2. EDZL scheduling algorithm removes dhall's effect but it does not ensure the schedulability of all tasks. It happens when M high priority running tasks (priority tasks according to EDZL) have small periods as compared to remaining tasks. EDZL scheduling algorithm is an optimal scheduling algorithm if all tasks have same deadlines.

4.2.2 Accumulative Effect

EDZL scheduling algorithm is not optimal if all tasks do not have same time periods. In this case, weight of a ready task (a task waiting for its selection to execute) comes out to be greater than accumulating offloading of M high priority running tasks.

4.2.2.1 Single Processor Case

In case of single processor, offloading factor of a task T_i represents percentage of processor that is used to execute tasks other than T_i, and there exists no such task that has utilization greater than offloading factor of task T_i (if $U(\tau) \leq 1$).

$$\frac{L_i}{P_i} \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{C_j}{P_j}$$

4.2.2.2 Multiprocessor Case

In case of multiprocessor system, there may exists such task T_j (waiting/blocked for execution) that has utilization equal to accumulative offloading of M high priority running tasks, and P_j is greater than periods of these running tasks.

$$\sum_{\substack{i=1\\ i\neq j}}^{M} \frac{L_i}{P_i} = \frac{C_j}{P_j} \qquad P_j > P_i$$

In this case, task T_j is selected to execute (or during execution of T_j) with more than one processors idle. It leaves at least one processor idle for some time, thus causing tasks to miss their deadlines for their next jobs (if $U(\tau) = M$).

Here is an example (Fig:4.7(a)) to illustrate, that how accumulation of utilization of $M = 2$ high priority running tasks cause a processor to go idle. Given the set of periodic tasks (Task T =C/P) T_1= (2/3), T_2 = (2/3), T_3 = (4/6). The task set is not schedulable on two processors either using EDF or EDZL.

$$\frac{L_1}{P_1} + \frac{L_2}{P_2} = \frac{C_3}{P_3}$$
$$\frac{1}{3} + \frac{1}{3} = \frac{4}{6} \qquad P_3 > P_1 \; and \; P_2$$

Global EDF (g-EDF) scheduling algorithm is not an optimal scheduling algorithm. We explain in this section all those reasons/problems for which g-EDF is not an optimal scheduling algorithm. One problem (dhall's effect) is solved by EDZL scheduling algorithm. We work to explore those reasons and factors that make an optimal mono processor scheduling algorithm lost its optimality when applied to schedule tasks on multiprocessor architecture. We propose some other modifications which encompass all problems of g-EDF.

4.3 Multiprocessor ASEDZL Scheduling

In ASEDZL scheduling algorithm, it is ensured that there is no idle time on any processor until next release of a tasks, and M tasks ,which are considered M high priority tasks according to EDF (called T^{EDF} tasks), are selected to execute until next release of a task. At any release instant, if execution requirements of M high priority tasks are not sufficient to occupy all M processors until next release of a task, then few subsequent tasks (next higher priority tasks) are promoted in priority to keep all processors busy until next release of a task.

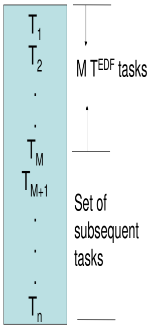


FIGURE 4.3: Tasks sorted according to EDF

To ensure that slacks of these M high priority tasks are filled, ASEZDL scheduler selects tasks from $TaskQueue$ (explained in next section) and not from ready queue.

4.3.1 $TaskQueue$

The tasks are sorted and placed in a queue called $TaskQueue$ (different from ready queue) according to closeness of their deadlines. The number of tasks in $TaskQueue$ remains unchanged, but positions of tasks in it may change depending upon absolute deadlines of tasks.

The number of tasks in $TaskQueue$ does not change and it is equal to n. This $TaskQueue$ is updated only when a task is released for its next instant, and is not updated when a task completes its execution. Scheduler selects tasks from $TaskQueue$ to execute and not from ready queue. A task T_i keeps its position in $TaskQueue$ even if it has finished its execution. It is different from the ready queue where task leaves queue when it has finished its execution. This $TaskQueue$ is divided into two groups.

- T^{EDF} ***Task Set***

- T^{S} ***Task Set***

1. T^{EDF} Task Set: A set of first M (Fig:4.3) tasks of this $TaskQueue$ are denoted as T^{EDF} tasks as these tasks are considered as M high priority tasks according to EDF algorithm. A task T_i is considered T^{EDF} tasks if its absolute deadline is earlier than absolute deadline of M^{th} T^{EDF} task, even if it has finished its execution.

2. T^{S} Task Set A subsequent task T_j ($T_j \in T^{S}$) is a task, which has deadline later than or equal to deadline of M^{th} T^{EDF} task.

There is another subgroup of these two task sets called T^{Z} (urgent) task set.

- T^{Z} Task Set:

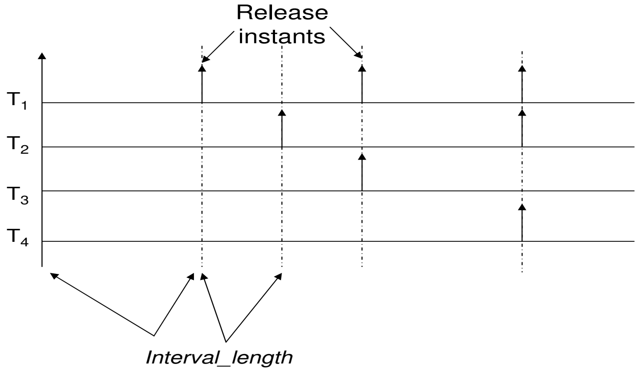


FIGURE 4.4: Release instants R_i associated with tasks and *interval_length*

If a task T_i is selected to execute when value of its laxity becomes zero, then it is called urgent task and is included into T^Z ($T_i \in T^Z$). Any task from both groups can become urgent task if its laxity reaches a value of zero.

4.3.2 Release instant R_i and *interval_length*

Laxities of M T^{EDF} tasks are filled between two consecutive release instants, which are release times of tasks (Fig:4.4). Time difference between two release instants is called *interval_length*. If M T^{EDF} tasks do not have total execution requirement equal to or more than $M * interval\_length$, then one or more than one subsequent task(s) T^S are selected to fill this laxity because scheduler needs to execute tasks between two release instants for time units equal to

$$TU = M \times (interval\_length) \tag{4.1}$$

ASEDZL algorithm tries to maximize the execution of T^{EDF} tasks to execute during this interval, but if M T^{EDF} tasks do not have sufficient remaining executing time, then sufficient subsequent tasks T^S are also executed to occupy processor until next release instant. Thus, scheduling of tasks on M processors becomes similar to filling a $2D$ bin at any time, where length of this $2D$ bin is equal to *interval_length*, and height is equal to M. To maximize the execution of T^{EDF} tasks and to ensure filling of this $2D$ bin, M T^{EDF} tasks and few number of subsequent tasks are assigned virtual deadlines and local execution time. Local execution time of tasks is equal to capacity of the $2D$ bin, and assignment of virtual deadline ensures filling of this $2D$ bin.

4.3.3 Virtual Deadline and Local Execution Time

At release instant R_k, M T^{EDF} tasks are assigned a virtual deadline equal to next release time of a task which is R_{k+1}(next release instant). By default virtual deadline of a task is equal to its absolute deadline. Local execution time of a task represents execution time of task that can be executed between two release instants. It can not be more than $interval\_length$, and it is represented by C_i^v. By default value of local execution time is zero, and it is calculated as follows (for T^{EDF} tasks):

If $\sum_{T_i \in T^{EDF}} C_i^v < TU$, then few subsequent tasks are also assigned virtual deadline, and local execution time such that total local execution time of T^{EDF} tasks and those of subsequent tasks become equal to TU. Local execution time of subsequent tasks is calculated in an iterative way:

Algorithm 7 Calculations of Local execution times of tasks

```
tasks sorted in increasing order of their deadlines
(even with zero remaining execution time)
$C^v = 0$;
for $i = 1$ to $n$ do
    $C_i^v = 0$;
    $d_i^v = d_i$;
end for
for $i = 1$ to $n$ do
    $C_i^v = min(C_i^{rem}(R_k), interval\_length, TU - C^v)$;
    $d_i^v = R_{k+1}$;
    $C^v = C^v + C_i^v$;
    if $(TU - C^v == 0)$ then
        Break;
    end if
end for
```

4.3.4 Priority Order of Tasks

Tasks are selected to execute based on their virtual deadlines rather than their actual deadlines. If virtual deadline of two tasks is same then task with earlier real deadlines is given higher priority. Virtual laxities of tasks are calculated as well, which are the differences between their virtual deadlines and local execution times.

Priority order is defined as follows:

1. Tasks with virtual zero laxity (urgent tasks T^Z) are given higher priority over tasks with earlier deadlines.

2. Tasks with actual zero laxity (difference between real absolute deadline and real execution time) are given higher priority over task with virtual zero laxity.

3. Tasks with earlier deadlines are given higher priority over tasks with later deadlines.

Let $L_i^r(t)$ represents the real laxity of task T_i at time t and $L_i^v(t)$ represents the virtual laxity of task T_i at time instant t.

Example 4.2. *Here is an example (Fig:4.5) to illustrate the principle of ASEDZL. Given a set of periodic tasks T_1 = (2/3), T_2 = (2/3), T_3 = (4/6). The task set is not schedulable on two processors using EDF because slacks of tasks T_1 and that of task T_2 could not be filled by task T_3 (a single task), as filling of these slacks is delayed until completion of tasks T_2 and T_1. Two consecutive release instants are at $t = 0$ and $t = 3$. Tasks T_1 and*

Algorithm 8 Selection of tasks according to priority

```
tasks sorted in increasing order of their absolute deadlines(even with zero remaining execution time)
int $NumberOfTasksSelected = 0$;
for $i = 1$ to $n$ do
    if $(L_i^r(t) == 0)$ then
        Select $T_i$ to execute;
        $NumberOfTasksSelected = NumberOfTasksSelected + 1$
    end if
end for
for $i = 1$ to $n$ do
    if $(NumberOfTasksSelected < M)$ then
        if $(L_i^v(t) == 0 \wedge L_i^r(t) > 0)$ then
            Select $T_i$ to execute;
            $NumberOfTasksSelected = NumberOfTasksSelected + 1$
        end if
    end if
end for
for $i = 1$ to $n$ do
    if $(NumberOfTasksSelected < M)$ then
        if $(L_i^v(t) > 0 \wedge L_i^r(t) > 0)$ then
            if $(C_i^{rem}(t) > 0)$ then
                Select $T_i$ to execute;
                $NumberOfTasksSelected = NumberOfTasksSelected + 1$
            end if
        end if
    end if
end for
```

task T_2 are T^{EDF} tasks. So we define virtual deadlines of these two tasks equal to next release instant, which is 3. We define the local execution times of T^{EDF} tasks as follows:

$$C_1^v = min(C_1, interval\_length)$$
$$C_1^v = min(2, 3) = 2$$
$$C_2^v = min(C_2, interval\_length)$$
$$C_2^v = min(2, 3) = 2$$

*Total time units TU to be filled until next release instant are 2*3=6. Time units which are filled by T^{EDF} tasks are $C_1^v + C_2^v = 4$. Remaining time units $(6-4=2)$ are filled by subsequent tasks. The virtual deadline of first subsequent task is set to 3 as well and its local execution time is calculated as follows:*

$$C_3^v = min(C_3, interval\_length, (TU - C_1^v - C_2^v))$$
$$C_3^v = min(4, 3, 2) = 2$$

Laxities of M T^{EDF} tasks are filled during each interval_length. It ensures that there will be no idle time on any processor as depicted in Fig:4.5.

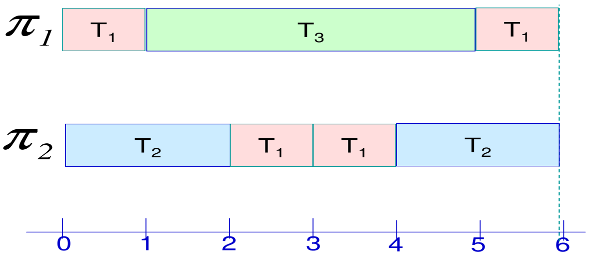
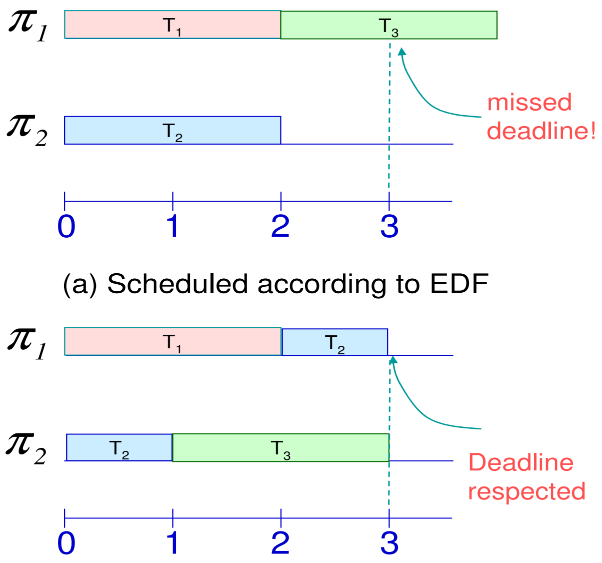


FIGURE 4.5: ASEDZL scheduling Algorithm Description



(a) Scheduled according to EDF

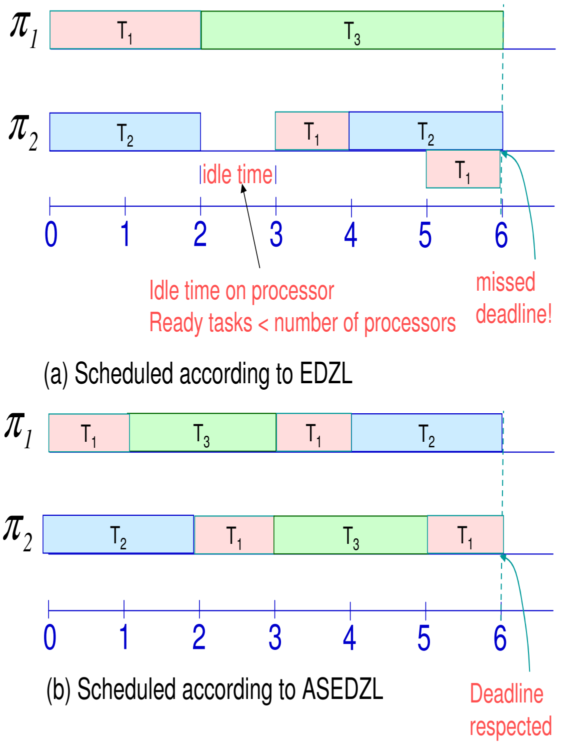
(b) Scheduled according to EDZL

FIGURE 4.6: Comparison of EDF with EDZL

4.4 Comparison of EDZL With ASEDZL

To illustrate the difference between EDF, EDZL and ASEDZL, we take the same example where $T_1 = (2/3)$, $T_2 = (2/3)$, $T_3 = (2/3)$. EDF is not able to schedule these tasks as shown in Fig:4.6(a) but EDZL schedules these tasks and all tasks respect their deadlines. However, EDZL algorithm is not an optimal scheduling algorithm due to possibility of leaving processor idle in the schedule. Now, we take the same example again, but we change the parameters of task T_3. Parameters of task T_3 are now 4/6 instead of 2/3.

We can observe that when tasks are scheduled according to EDZL algorithm there is an idle time on processor π_2 as there was only one task ready at instant $t = 2$ (Fig:4.7(a)). It happens because slack of T^{EDF} tasks (i.e., T_1 and T_2) is not anticipated in advance. Hence, task T_3 cannot fill this slack between time 2 and 3. According to ASEDZL, slack


(a) Scheduled according to EDZL

(b) Scheduled according to ASEDZL

Figure 4.7: Comparison of EDZL with ASEDZL

is anticipated in advance, and subsequent tasks T_3 is also assigned virtual deadline equal to 3 and is also assigned local execution time which is equal to 2 (Fig:4.7. Tasks T_2 and T_1 are selected to execute. At $t = 1$, laxity of T_3 becomes zero, hence its priority is promoted and it preempts lower priority task which is T_1. At $t = 2$, laxity of T_1 becomes zero as well, hence it replaces the T_2 which has finished its execution.

We can observe that laxities of T^{EDF} tasks (i.e., T_1 and T_2) are filled by T_3 until $t = 3$, and there is no idle time on processors.

4.5 Properties of ASEDZL Algorithm

A fundamental property of our propose ASEDZL algorithm is its controlled run time complexity and minimum preemptions of tasks. It ensures that all tasks meet their deadlines. The optimality of scheduling algorithm is provided, while minimizing task preemptions and scheduling overheads. In this sub section, we detail these properties.

4.5.1 Not Based on Basic Time Quanta

ASEDZL scheduling algorithm is not based on time quanta. Worst case execution and time period of a task can have any arbitrary value. Moreover, there is implication on execution requirements of task, as it is in case of LLFEF where execution requirements are assumed to be much higher than basic time quanta.

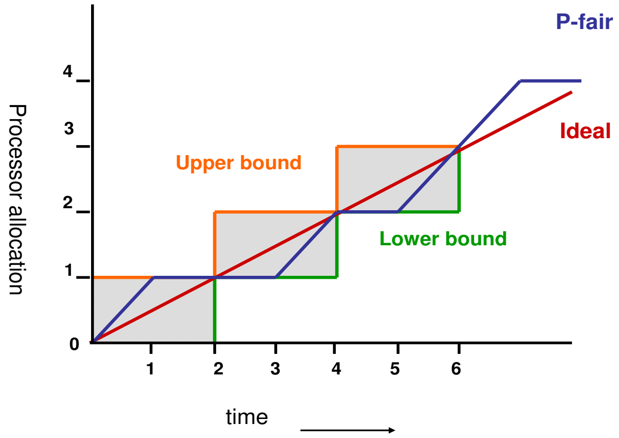


Figure 4.8: Fairness ensured at each instant

4.5.2 Notion of Fairness

For designing an optimal scheduling algorithm, the fluid scheduling model and the fairness notion are considered. In the fluid scheduling model, each task executes at a constant rate at all times. For a task set to be schedulable, fairness must be ensured for each task up to its deadline boundary and not necessarily through its execution. In the following subsection, we present fairness boundaries for three optimal global scheduling algorithms.

4.5.2.1 Pfair Fairness

The quantum-based Pfair scheduling algorithm is based on the fluid scheduling model, as the algorithm constantly tracks the allocated task execution rate through task utilization. The Pfair algorithms success in constructing optimal multiprocessor schedules can be attributed to fairness informally, all tasks receive a share of the processor time, and thus are able to simultaneously make progress (Fig:4.8). P-fairness is a strong notion of fairness, which ensures that at any instant, no application is one or more quanta away from its due share (or fluid schedule). Fairness is provided at each instant (integer multiple of basic time quanta) but the counter part is that Pfair introduces a lot of scheduling overheads in terms of increase in preemptions and migrations.

4.5.2.2 LLREF Fairness

LLREF is also based on the fluid scheduling model and the fairness notion. To avoid Pfairs quantum-based approach, LLREF introduces an abstraction called the Time and Local Execution Time Domain Plane (or abbreviated as the T-L plane), where tokens representing tasks move over time. LLREF scheduler splits tasks to construct optimal schedules, not at time quantum expiration, but at other scheduling events which are release times of tasks (release instants), and consequently avoid Pfairs frequent scheduling and migration

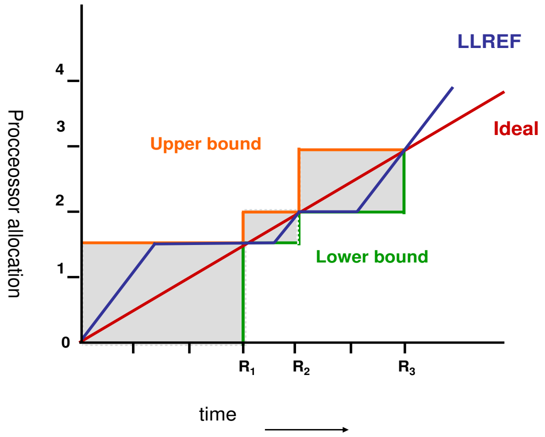


FIGURE 4.9: Fairness ensured at each release instant

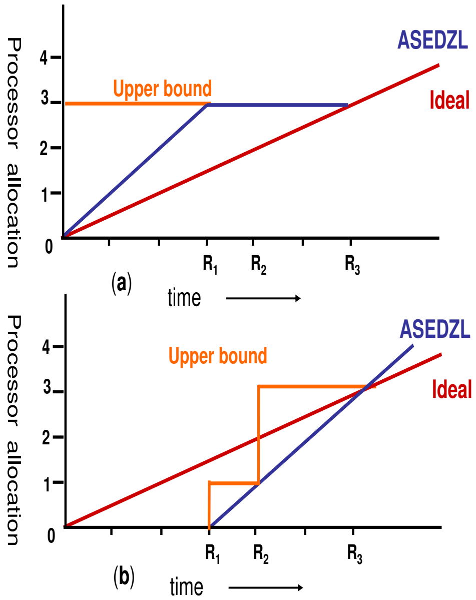


FIGURE 4.10: Fairness ensured at deadline boundaries

overheads. LLREF ensures fairness at all release instant boundaries (Fig:4.9)(task release times).

4.5.2.3 ASEDZL Fairness

ASEDZL algorithm is not based on the notion of fairness. It ensures fairness for a task only at its deadline boundary (Fig:4.10), which is mandatory for a task to be schedulable. Unlike LLREF, allocation of processor time to different tasks between any two release instants is not proportional to their weights.

4.5.3 Minimum Active Tasks Between two Release Instants

The minimum number of tasks that must be executing cannot be less than M if processor utilization induced by the task set is M. LLREF scheduling algorithm executes all n tasks to execute between any two release instants, even if the difference between two consecutive release instants is very small. In this case, each task is executed for a very small fraction of time leading to increased scheduling overheads, and sometimes make it impractical. Tasks selected to execute between any two release instants by ASEDZL algorithm can be as minimum as M if $U(\tau) = M$. If $U(\tau) < M$, the minimum number of tasks between any two release instants can be 1 if scheduled by ASEDZL while LLREF schedules all n tasks between any two release instants.

4.5.4 Anticipation of Laxities of M T^{EDF} Tasks

ASEDZL algorithm is mainly based on the idea of anticipating laxities of M T^{EDF} tasks before those laxities appear. The objective of ASEDZL is that these laxities can be filled by execution requirement of subsequent tasks. Subsequent tasks have sufficient execution requirements to fill the laxities of the M T^{EDF} tasks.

Let utilization of the task set equals M then we find that:

$$\sum_{T_i \in T^{EDF}} \frac{C_i}{P_i} + \sum_{T_j \in T^{S}} \frac{C_j}{P_j} = M$$

$$\sum_{T_i \in T^{EDF}} \frac{C_i}{P_i} + \sum_{T_i \in T^{EDF}} \frac{L_i}{P_i} = M$$

$$\sum_{T_j \in T^{S}} \frac{C_j}{P_j} = \sum_{T_i \in T^{EDF}} \frac{L_i}{P_i} \tag{4.2}$$

if $U(\tau) < M$ then:

$$\sum_{T_j \in T^{S}} \frac{C_j}{P_j} < \sum_{T_i \in T^{EDF}} \frac{L_i}{P_i} \tag{4.3}$$

The objective is now to demonstrate that there are at least M ready tasks at each release instant (if $U(\tau) = M$) and each task meets its deadline.

4.6 Optimality of ASEDZL

Due to dhall's effect g-EDF scheduling algorithm can cause a task to miss its deadlines even at very low utilization of task set.

Lemma 4.1. *A task T_i can miss its deadline if scheduled by g-EDF scheduler even if it is the only task executing on one processor and all other $(M-1)$ processors are free.*

Proof. According to g-EDF scheduler, tasks are sorted in queue according to closeness of their deadlines. Scheduler selects first M tasks to execute on M processors. When a task finishes its execution, scheduler selects next task to execute. If we consider the example.4.1 explained earlier where task set is composed of $M+1$ tasks. We can find that $(M+1)^{th}$ task with execution requirement of p and period $p+1$ misses its deadline at $p+1$ while all other processors are free during the time interval [2,p]. □

Lemma 4.2. *A task T_i can miss its deadline if scheduled by EDZL scheduler, when all M processors are executing urgent tasks when laxity of T_i becomes zero.*

Proof. EDZL scheduler preempts a task with latest deadline, if task T_i becomes an urgent task. If task T_i has missed its deadline, then it implies that EDZL scheduler has not preempted a task to execute T_i. It is against EDZL scheduling policy. There is only one reason for not preempting a task which is that all tasks executing on M processors are urgent tasks. Hence, task T_i misses its deadlines only when all processors are executing M urgent tasks. □

We now establish optimality of ASEDZL scheduling algorithm by proving that all tasks meet their deadlines if utilization of task set does not exceed processing capacity of the system.

4.6.1 $U(\tau)$ at Release Instants R_i

Let us consider mono-processor case and a task set composed of two tasks T_i and T_j such that $u_i + u_j = 1$ and $P_i < P_j$. Both tasks are released at $t = t_1$. let use assume that task T_i is selected to execute by scheduler (single processor). At time $t = t_1 + IL$, current weight u_i^c decreases. The decrease in its weight of task T_i at time t, denoted by $DW_i(t)$, is calculated as follows:(task T_i has not finished its execution yet)

$$\begin{aligned} DW_i(t) &= \frac{C_i}{P_i} - \frac{C_i - IL}{P_i - IL} \\ &= \frac{IL(P_i - C_i)}{P_i(P_i - IL)} \end{aligned}$$

$$DW_i(t) = \frac{IL(L_i)}{P_i(P_i - IL)} \tag{4.4}$$

At the same time current weight of task T_j increases, and increase in its weight, represented by $IW_j(t)$ is calculated as follows:

$$IW_j(t) = \frac{C_j}{P_j - IL} - \frac{C_j}{P_j}$$

$$IW_j(t) = \frac{IL(C_j)}{P_j(P_j - IL)} \tag{4.5}$$

As

$$\frac{C_i}{P_i} + \frac{C_j}{P_j} = 1$$

$$\Rightarrow \frac{L_i}{P_i} = \frac{C_j}{P_j}$$

And

$$P_j > P_i$$

Then Equation(4.4) and Equation(4.5) implies that decrease in weight of task T_i is greater than increase in weight of task T_j until time t=$t_1 + IL$.

$$\frac{IL(C_j)}{P_j(P_i - IL)} - \frac{IL(C_j)}{P_j(P_j - IL)} = \frac{IL.C_j(P_j - P_i)}{(P_i - IL)(P_j - IL)}$$

As $P_j - P_i > 0$, it implies that decrease in weight of running task is greater than increase in weight of waiting task.

if $IL = P_i$, then task T_i has finished its execution and task T_j has executed for time units equal to L_i until $t_1 + P_i$. It implies that total decrease/increase in weight of task T_j is zero if $\mu_i + \mu_j = 1$.

$$\begin{aligned}\frac{C_j}{P_j} - \frac{C_j - L_i}{P_j - P_i} &= \frac{P_j.L_i - P_i.C_j}{P_j.(P_j - P_i)} \\ &= \frac{P_j.P_i\left(\frac{L_i}{P_i} - \frac{C_j}{P_j}\right)}{P_j.(P_j - P_i)}\end{aligned}$$

$$\frac{C_j}{P_j} - \frac{C_j - L_i}{P_j - P_i} = \frac{P_i\left(\frac{L_i}{P_i} - \frac{C_j}{P_j}\right)}{(P_j - P_i)} \tag{4.6}$$

But if there are more than two tasks in the system (where $P_i \neq P_j$), then total decrease in weights of all tasks is greater than zero. Moreover current weight of task T_i is also zero before $t = t_1 + P_i$. It implies that sum of current utilization of all tasks ready at $t = t_1 + P_i$ is less than or equal to 1. But if task T_j does not execute for L_i time units until $t = t_1 + P_i$, then sum of current weights is higher than 1 at release of task T_i.

It connotes that if tasks with closer absolute deadlines are given higher priority than tasks with later absolute deadlines, then sum of current weights of all tasks does not exceed than capacity of architecture until release of a task (if processor has been busy until release of task$U(\tau) = 1$). If in Equation(4.6), $\frac{L_i}{P_i} < \frac{C_j}{P_j}$ then increase in weight of block task is higher than decrease in weight of running task.

In the same way on multiprocessor, if tasks with closer absolute deadlines are given higher priority over tasks with later deadlines, then sum of decrease in current weights of higher priority tasks until next release of a task is higher than sum of increase in current weights of tasks waiting for execution, provided all processor are busy until next release instant. As explained in section 4.2.2, $\frac{L_i}{P_i} < \frac{C_j}{P_j}$ can be true where T_i is running task while T_j represent the blocked task. In this case, increase in weight of T_j is higher than decrease in weight of T_i, but we know from Equation(4.3) that:

$$\sum_{T_j \in T^S} \frac{C_j}{P_j} < \sum_{T_i \in T^{EDF}} \frac{L_i}{P_i}$$

It implies that sum of decrease in weights of running tasks is greater than or equal to sum of increase in weights of blocked tasks, even if increase in weights of k blocked tasks ($k < M$) is higher than decrease in weights of k running tasks. In this case, rate of decrease in weights of remaining $M - k$ blocked tasks is much higher to encounter increase in weights of k blocked tasks.

⇒ that at any release instant R_k.

$$\sum_{i=1}^{n} \mu_i^c \leq M \tag{4.7}$$

Let $\tau_r^{R_k}$ defines the set of tasks which has non zero remaining execution time at $t = R_k$ then we have:

$$\sum_{T_i \in \tau_r^{R_k}} \frac{C_i^{rem}(R_k)}{d_i^{rem}(R_k)} \leq M - \sum_{\substack{j=1 \\ T_j \notin \tau_r^{R_k}}}^{n} \frac{C_j}{P_j} \tag{4.8}$$

As ASEDZL assigns higher priority to tasks with earlier deadlines, it demonstrate

$$\sum_{T_i \in \tau_r^{R_k}} \frac{C_i^{rem}(R_k)}{d_i^{rem}(R_k)} \geq \sum_{T_i \in \tau_r^{R_k}} \frac{C_i^{rem}(R_{k+1})}{d_i^{rem}(R_{k+1})} \tag{4.9}$$

Equation(4.8) and Equation(4.9) imply

$$\sum_{T_i \in \tau_r^{R_{k+1}}} \frac{C_i^{rem}(R_{k+1})}{d_i^{rem}(R_{k+1})} \leq M \tag{4.10}$$

4.6.2 No Idle Time on Processor

In this section, we demonstrate that there is no idle time on any processor, if scheduled using ASEDZL (provided $U(\tau) = M$).

4.6.2.1 2D Bin Packing

At release instant R_k, ASEDZL assigns virtual deadlines to tasks until next release instant R_{k+1}, and tries to schedule tasks on M processors. Thus scheduling of tasks can be described as filling of a 2D bin, where height of this bin is fixed to M and length of this

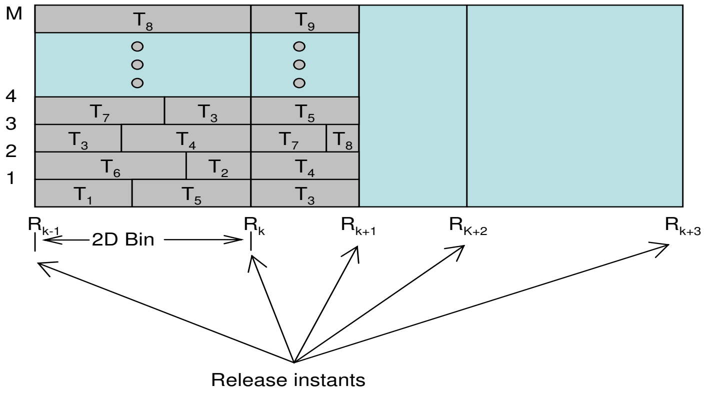


FIGURE 4.11: 2D Bin Packing

bin is defined as difference between two consecutive release instants. As this difference between two release instants is different, so we have bins of different lengths and fixed height at different release instants as shown in Fig:4.11.

There is no idle time on any processor if scheduled using ASEDZL algorithm (utilization of task set = M). Global scheduler works as packing of these 2D bins of different lengths. At any time t, there is only one $2D$ bin to be filled. Length of a bin at time t_1 can be different from length of other $2D$ bin at time t_2, where t_1 and t_2 don't lie between same release instants. All $2D$ bins have same height which is equal to M.

EDZL [75] is an optimal scheduling algorithm if all tasks have the same deadlines. It implies that if EDZL is used to fill this 2D bin, such that local execution requirements of tasks is equal to capacity of this 2D bin and all tasks have same virtual deadline, then there will be no idle slot (space) in this 2D bin.

4.6.2.2 Minimum Number of Ready Tasks:

Let us assume that there is an idle time on a processor between two release instants R_k and R_{k-1}, it implies that($U(\tau) = M$):

1. Total remaining execution requirements of all tasks is less than $M \times (R_k - R_{k-1})$and/or
2. Number of ready tasks at Release Instant (R_{k-1}) is less than M

As $U(\tau) = M$, it implies that:

$$\sum_{i=1}^{n} \frac{HP}{P_i} \times C_i = M \times HP$$

which implies that at any release instants t=R_{k-1}:

$$\sum_{i=1}^{n} C_i^{rem}(t) \geq M \times (R_k - R_{k-1})$$

For any schedule (feasible or infeasible), at any release instant $t = R_{k-1}$:

$$\sum_{i=1}^{n} min\left[C_i^{rem}(t), (R_k - R_{k-1})\right] \geq M \times (R_k - R_{k-1})$$

If

$$\sum_{i=1}^{n} min\left[C_i^{rem}(t), (R_k - R_{k-1})\right] < M \times (R_k - R_{k-1})$$

It implies that at least one task has not executed sequentially before $t = R_{k-1}$ i.e., it has executed on more than one processors simultaneously, which is against the basic assumption. It demonstrates that:

$$\sum_{i=1}^{n} min\left[C_i^{rem}(t), (R_k - R_{k-1})\right] \geq M \times (R_k - R_{k-1}) \tag{4.11}$$

Equation(4.11) implies that there are, at least, M ready tasks during this interval.

4.6.3 No More than M Urgent tasks

If tasks are scheduled according to ASEDZL, then no more than M tasks can claim zero laxity at the same time. If tasks are scheduled according to ASEDZL, then no more than M tasks can have zero laxity at time t.

Theorem 4.3. *No more than M tasks can have zero laxity at time t when scheduled using ASEDZL where $n \leq M$.*

Proof. Laxity of task reaches a value of zero if it is not selected to execute. As we have M processors, ASEDZL always selects $\leq M$ tasks (if there are M ready tasks) to execute. Hence, at any time t, there is no task waiting (blocked task) for processor. It implies that there will be no task with zero laxity at time t. □

Theorem 4.4. *No more than M tasks can have zero laxity at time t when scheduled using ASEDZL where $n > M$.*

Proof. Let us say that a task T_j claims a zero laxity at instant t_1 $(t_1 < t \leq d_j)$. It implies all processors were executing M T^{EDF} high priority tasks until t_1. It implies that:

$$\sum_{T_i \in T^{EDF}} \frac{C_i^{rem}(t_1)}{d_i^{rem}(t_1)} + \frac{C_j^{rem}(t_1)}{d_j^{rem}(t_1)} \geq 1$$

At least one processor which executes task T_j and a T^{EDF} task is fully busy until t. Let us assume again that task T_k also claims zero laxity at instant t_2 $(t_1 \leq t_2 < t)$. It implies all processors were executing $(M-1)$ high priority tasks and one urgent task until t_2 It

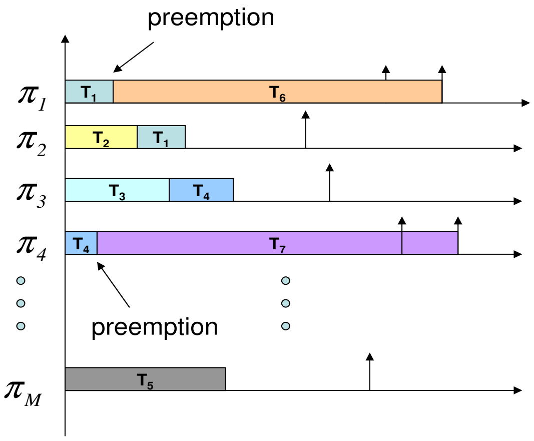


Figure 4.12: Optimality proof

implies that until t:

$$\sum_{T_i \in T^{EDF}} \frac{C_i^{rem}(t_2)}{d_i^{rem}(t_2)} + \frac{C_j^{rem}(t_2)}{d_j^{rem}(t_2)} + \frac{C_k^{rem}(t_2)}{d_k^{rem}(t_2)} \geq 2$$

In the same way if $t_m < t$ is such an instant where there were $(M-1)$ urgent tasks executing on processors and task T_v also claims zero laxity then it implies that:

$$\sum_{T_i \in T^{EDF}} \frac{C_i^{rem}(t_m)}{d_i^{rem}(t_m)} + \sum_{\substack{T_j \in T^Z \\ T_j \neq T_i}} \frac{C_j^{rem}(t_m)}{P_j^{rem}(t_m)} \geq M$$

If at instant t, a $(M+1)^{th}$ task also claims zero laxity then it implies that:

$$\sum_{T_i \in T^{EDF}} \frac{C_i^{rem}(t)}{d_i^{rem}(t)} + \sum_{\substack{T_j \in T^Z \\ T_j \neq T_i}} \frac{C_j^{rem}(t)}{d_j^{rem}(t)} \geq M+1 \tag{4.12}$$

But this contradicts Equation(4.7). □

It means that if there is no idle time on any processor before instant t, then no more than M tasks can claim zero laxity simultaneously, if scheduled according to ASEDZL. From this, we can deduce that if there is no idle time on any processor, all tasks respect their deadlines. This property is illustrated in Fig:4.12 where tasks T_1 and T_4 are preempted by urgent tasks T_6 and T_7 respectively. It imply that $\sum_{i=1}^{n} C_i^{rem}/d_i^{rem} \geq 2$.

4.7 Complete Example

We illustrate this principle by an example. Task set is given in Table 4.1. Utilization of this task set is 3. This task set is scheduled on 3 processors. At $t=0$,Tasks T_4, T_5, T_2 are

Task	C	P
T_1	2	5
T_2	2	5
T_3	7	10
T_4	3	4
T_5	3	4

Table 4.1: Task Parameters

three T^{EDF} tasks according to closeness of their deadlines. Virtual deadline is assigned to all these three T^{EDF} tasks which is 4. Local execution time of these three tasks is calculated as follows:

$$C_4^v = min(3,4) = 3$$
$$C_5^v = min(3,4) = 3$$
$$C_2^v = min(2,4) = 2$$

Total local execution time of three T^{EDF} tasks is 8, and TU required is (3*4) 12. So we need to add few subsequent tasks that have execution requirement of 4 time units. First subsequent task is T_1 (Fig:4.14)and its local execution time is:

$$C_1^v = min(2,4,12-8) = 2$$

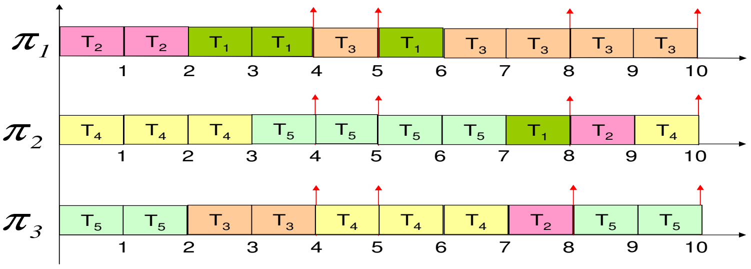
So we need to select second subsequent task which is T_3 and its local execution time is calculated as follows

$$C_3^v = min(7,4,12-10) = 2$$

Virtual deadlines of the two subsequent tasks are same as virtual deadlines of T^{EDF} tasks.

T_4, T_5 and T_2 are selected for execution due to their higher priorities. At $t = 2$, local laxity of tasks T_1, T_3 reaches a value of zero, so tasks T_5 (T^{EDF}) is replaced with urgent task T_3, and T_1 is selected to execute as T_2 has finished its execution. At $t = 2$, T_3 is given higher priority over T_5, as local laxity of T_3 has reached as value of zero (Fig:4.13).

At $t = 4$, T_2, T_1 and T_4 are three T^{EDF} tasks and they are assigned virtual deadline equal to 5. Their calculated local execution times are 0,0,1 respectively. TU required is equal 3, so there is a need to add subsequent tasks. First subsequent task is T_5 and its local execution is calculated to be 1, second subsequent task is T_3 and its local execution time is 1. At $t = 6$, actual laxity of task T_3 becomes zero so it has been given the highest priority and it has preempted the lowest priority task T_1 (one which has the latest deadline) and so on.



π_1
T2 T2 T1 T1 T3 T1 T3 T3 T3 T3
π_2
T4 T4 T4 T5 T5 T5 T5 T1 T2 T4
π_3
T5 T5 T3 T3 T4 T4 T4 T2 T5 T5
1 2 3 4 5 6 7 8 9 10

FIGURE 4.13: : Illustration of Algorithm by Complete Example

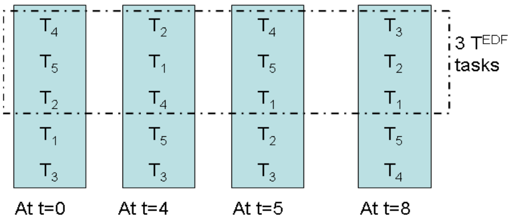


FIGURE 4.14: *TaskQueue* at different Release Instants

4.8 Performance Evaluation

In this section, we shall provide a comparison of our proposed approach with already existing optimal global scheduling algorithms i.e., Pfair and LLREF. ASEDZL algorithm is not based on time quanta unlike Pfair. Execution requirement and time periods of tasks can have any arbitrary value. Pfair scheduler selects tasks to execute at each instant, thus introducing a lot of overheads in terms of increased released instants R_i. Unlike Pfair, LLREF is not based on time quanta but it increases preemptions of tasks to a great extent. LLREF schedules all n tasks between any two release instants while our proposed algorithm schedules minimum number of tasks between any two release instants. Minimum number of tasks selected to execute can be as minimum as M, if tasks are scheduled according to ASEDZL. We can observe it in Fig:4.13, between release instant at 4 and 5, ASEDZL has selected only 3 (M) tasks to execute while LLREF will schedule all 5 (n) tasks between these two release instants. We have performed simulations on Cofluent Studio with 2 processors and different number of tasks such that overall utilization of task set is 1.9. Parameters of tasks i.e., period and worst case execution time, are generated randomly. We have illustrated that ASEDZL outperforms LLREF scheduling algorithm. Number of scheduling events in ASEDZL schedule are much lower than appearing in LLREF based schedule of task, and the difference between number of preemptions of two tasks is exceptional, as shown in Fig:4.15. We have also compared the performance of ASEDZL with EDZL. We can observe than number of preemptions are higher in case

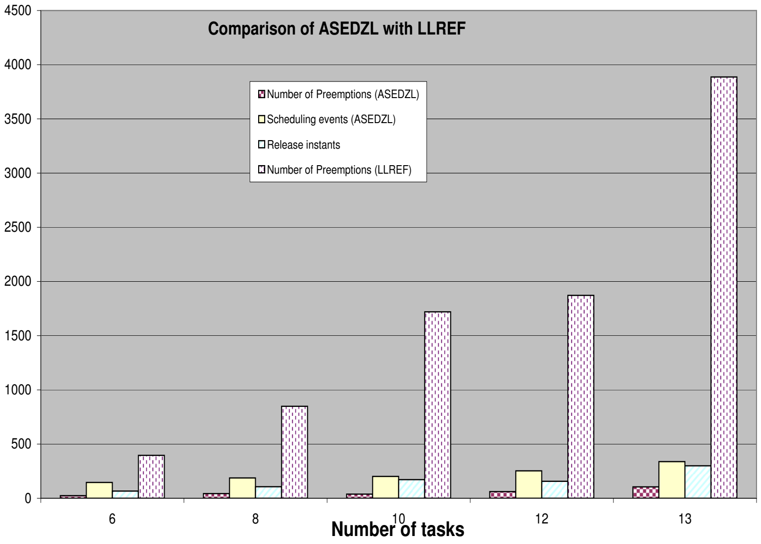


FIGURE 4.15: Comparison of ASEDZL algorithm with LLREF

of ASEDZL that those of EDZL (Fig:4.16) but there are deadline misses if tasks are scheduled according to EDZL scheduling algorithm.

4.9 Algorithm Overhead

One of the main concerns against global scheduling algorithms (e.g., Pfair, LLREF, g-EDF) is their overhead caused by frequent scheduler invocations. Srinivasan et al. identify three specific overheads:

1. ***Scheduling overhead:*** this accounts for the time spent by scheduling algorithm including that for constructing schedules and ready-queue operations.

2. ***Context-switching overhead:*** which accounts for the time spent in storing the preempted task's context and loading the selected task's context.

3. ***Cache-related preemption delay:*** which accounts for time incurred in recovering from cache misses that a task may suffer when it resumes after a preemption. Note that when a scheduler is invoked, the context switching overhead and cache-related preemption delay may not happen.

In contrast to Pfair, LLREF is free from time quanta but it yields more frequent scheduler invocations than g-EDF. EDZL algorithm adds one more scheduling event to g-EDF which

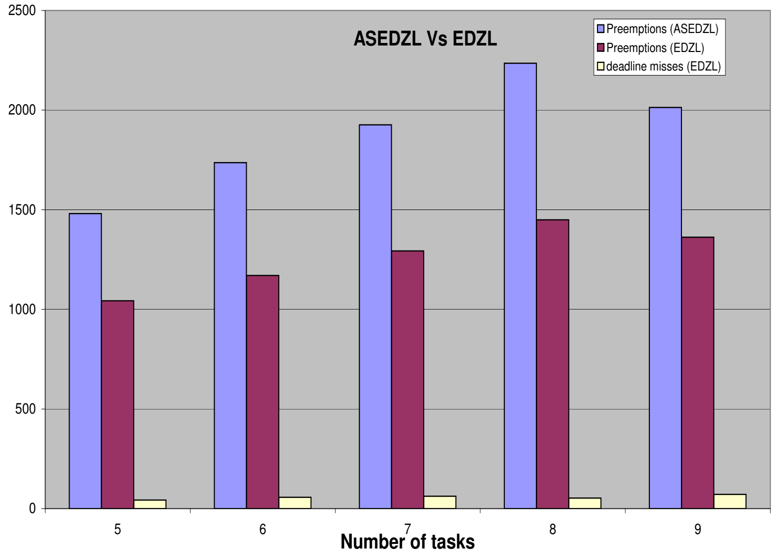


FIGURE 4.16: Comparison of ASEDZL algorithm with EDZL algorithm

is zero laxity of a task. However, ASEDZL does not add any more scheduling event than appearing in EDZL.

4.10 Conclusions

We have presented an optimal real-time scheduling algorithm for multiprocessors, which is not based on time quanta. This algorithm called ASEDZL, is designed based on idea of anticipating the filling of laxities of M T^{EDF} tasks. This proposed algorithm is a modified version of g-EDF scheduling algorithm. g-EDF scheduling algorithm has the least runtime complexity for scheduling tasks on multiprocessor architecture, but it is not optimal. We have compared the characteristics of g-EDF with basic intrinsic properties of optimal mono-processor EDF scheduling algorithm, and ensured these properties to appear in g-EDF scheduling algorithm by introducing few modifications which brought optimality in g-EDF as well. We proved that EDZL scheduling algorithm is optimal if there is no idle time on any of processor during execution. This condition is accomplished by our algorithm ASEDZL by anticipating the laxities in advance to assure that there is no idle time on processors.

We demonstrated that our algorithm does not increase the scheduling events. It is much better than LLREF scheduling algorithm where fraction of all n tasks are executed between any two release instants, while in our proposed approach the number of tasks which are selected to execute between any two release instants can be as minimum as one. We have performed simulations on Cofluent tool and have demonstrated different results in terms of number of scheduling instants and number of preemptions. We are also developing with our partners a simulation tool in context of PHERMA[1] project which will help us to simulate these algorithms. This simulator is designed to take care of all real parameters of an architecture like, effect of shared or distributed memory, cache penalties, scheduling overheads, cost of preemption and migration of a task etc.

[1] PHERMA is a National project funded by ANR ((Agence Nationale de la Recherche), France)under the project call 2006 of *Architecture of Future*. There are four partener of this project 1)CEA List - LCEI 2)LEAT - Université de Nice Sophia Antipolis, CNRS UMR 6071 3)Ecole Centrale de Nantes IRCCyN : équipe Systèmes Temps Réel 4)THALES Communications
web: http://pherma.irccyn.ec-nantes.fr/

Chapter 5

Hierarchical Scheduling Algorithm

5.1 Introduction

Some tasks have high affinity with specific processors and they can be executed only on those processor. That's why, these kinds of tasks are partitioned on processors and tasks on one processor are grouped to form supertask. Scheduling of global tasks and supertasks is called group-based or hierarchical scheduling technique [32–34, 53] (supertasking approach). Under this approach, sets of tasks are scheduled instead of individual tasks. This approach of grouping tasks (forming a supertask) was pioneered by Moir et al. [53]. Their work was motivated by the fact that many applications have tasks that cannot migrate because they interact with sensors and actuators at fixed locations. In this approach of supertasking, supertasks (one supertask from each group) are scheduled by a global scheduler among other global tasks. When a supertask is selected to execute, an internal scheduler is invoked to distribute the processor time among the individual tasks (local tasks). Use of optimal global scheduling algorithm like Pfair, ensures deadline guarantees for supertasks and global tasks but it does not guarantees if local tasks will respect their deadlines, as scheduling of local tasks (partitioned tasks) is dependent on decisions of global scheduler and its policy. Works of Moir et al.[53] and Holman et al.[32] demonstrated that local tasks miss their deadlines when supertasking is used in conjunction with all known Pfair scheduling algrithms.

In this chapter, we present two contributions in the field of hierarchical scheduling algorithms where we develop/establish sufficient conditions to ensure deadline guarantees to all local tasks. Moreover, we also propose to used ASEDZL as global scheduler to schedule supertasks and global tasks which does not impose any further conditions for a feasible schedule. The proposed approaches are as follows:

1. Establishing weight bound relation between supertask and component tasks for guaranteed real time constraints, where a Pfair Scheduler is used to schedule tasks at global level.

2. Hierarchical scheduler based on scheduling global tasks using ASEDZL at global level.

We define few more symbols before diving into details of the algorithms. In these algorithms a set of task is partitioned on M processors and is represented by τ_{par}. The nj number of tasks partitioned on processor π_j are represented by τ^j. Tasks that are not grouped to form a supertask are called migrating (global) tasks, and the total number of global tasks are represented by ng.

$$(n1 + n2 + n3 + ... + nM) + ng = n$$

5.2 Weight Bound Relation between supertask and component tasks for guaranteed real time constraints

A given periodic task set is subdivided into two groups -partitioned and global tasks. In partitioned subset of tasks, each task is assigned to a particular processor statically (offline), and it always executes on the same processor during runtime.

A set of tasks partitioned on processor $(\pi)_j$ is grouped to form a supertask T^j_x, and tasks comprising this supertask are called component tasks. These supertasks among other migrating tasks are scheduled globally by a Pfair scheduling algorithm. Whenever a supertask is scheduled, its processor time is allocated to one of its component task according to an internal scheduling algorithm. Supertasking effectively relaxes the strictness of Pfair scheduling: the group is required to make progress at a steady rate rather than individual tasks. Unfortunately, Moir and Ramamurthy [53] demonstrated that using an ideal weight assignment with a supertask cannot, in general, guarantee the deadliness of its component tasks. Holman [32] has developed criteria where supertasks are assigned weights more than their actual weight (u^j_x) to provide deadline guarantees to component tasks. Reweighting of supertasks proposed by Holman in [32] provides feasible schedule. Let us consider that $\hat{u}^j_x$ represents the weight of a supertask that ensure deadline guarantees for local tasks ($\hat{u}^j_x \geq u^j_x$). The difference $\hat{u}^j_x - u^j_x$ is called the inflation factor. Holman et al. proposed to increase the weight of a supertask to ensure a feasible schedule except when weight of a supertask is equal to one. In this section, we establish a sufficient condition if respected, guarantees feasible schedule of local tasks. In this case, increase in weight of supertask is not required. If this condition is not respected then there is a need to increase the weight of supertask such that this condition is respected. Thus this condition also provides the more accurate value of weight of supertask which ensure deadline guarantees to local tasks. Inflation factor introduced in this case is smaller than proposed by Holman et al.

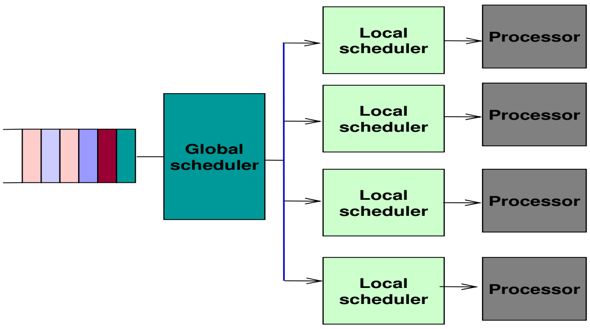


FIGURE 5.1: Hierarchical scheduler

In this section, we address issue of establishing a relationship between weight of a supertask and those of component tasks to provide deadline guarantees. We establish a condition, which must be taken into account while constructing a supertask from component tasks. We prove that all component tasks respect their deadline constraints, if supertask and its component tasks have the established relation/condition in them. Moreover the schedulable bound of algorithm remains intact i.e., M.

5.2.1 Background

We extend the work of Moir et al. [53] and Holman [32, 33] on the approach of hierarchal scheduling (Fig:5.1) of tasks. According to this approach, tasks are grouped to form M sets of tasks and each set of tasks represents a supertask. These supertasks among other tasks, which were not grouped to form a supertask, are scheduled using Pfair scheduling algorithm.

We describe briefly the Pfair global scheduling approach as it is related to results presented in this section.

5.2.1.1 Pfair Scheduling Algorithm

We now formally describe the Pfair scheduling concepts on which our work is based. In Pfair scheduling, processor time is allocated in discrete time units, or quanta. We refer to the time interval $[t; t+1)$, where t is a nonnegative integer, as slot t (hence, time t refers to the beginning of slot t). Following Baruah[14], we assume that all task parameters are expressed as integer multiples of the quantum size.

A task's weight defines the rate at which it is to be scheduled. In a perfectly fair (ideal) schedule, every task T_i should receive a share of $u_i.t$ time units over the interval $[0; t)$ (which implies that each job meets its deadline). In practice, this degree of fairness is impractical, as it requires the ability to preempt and switch tasks at arbitrarily small time scales. Notice that such idealized sharing is clearly not possible in a quantum-based schedule. Instead, Pfair scheduling algorithms strive to ”closely track” the allocation of processor time in the ideal schedule. This tracking is formalized in the notion of per-task lag, which is the difference between a task's allocation in the Pfair schedule and the allocation it would receive in an ideal schedule. Formally, the *lag* of task T_i at time t, denoted $lag(T_i, t)$, is defined as follows:

$$Lag(T_i, t) = (u_i).t - allocated(T_i, t) \tag{5.1}$$

Task T_i is said to be over-allocated or ahead at time t, if $lag(T_i, t) < 0$, i.e., the actual processor time received by T_i over [0; t) is more than its ideal share over [0, t). Analogously, task T_i is said to be under-allocated or behind at time t if $lag(T_i, t) > 0$. If $lag(T_i, t) = 0$, then T_i is punctual, i.e., it is neither ahead nor behind. A schedule is defined to be proportionate fair (Pfair) if and only if:

$$\forall\ T_i, t :: -1 < Lag(T_i, t) < 1 \tag{5.2}$$

Informally, allocation error associated with each task must always be less than one quantum. The lag bounds given in Equation(5.2) have the effect of breaking a task into smaller executable units that are subject to intermediate deadlines.

Window tasks: The lag bounds given in Equation(5.2) have the effect of breaking each task into a finite sequence of unit-time subtasks (we call them window tasks) Fig:5.2. We denote the v^{th} subtask of task T_i as T_i^v, where $v \geq 1$. Pfair associates with each window task a pseudo release (Equation(5.3)) and pseudo deadline (Equation(5.4)).

$$r(T_i^v) = \left\lfloor \frac{(v-1) \times P_i}{C_i} \right\rfloor \tag{5.3}$$

$$d(T_i^v) = \left\lceil \frac{v \times P_i}{C_i} \right\rceil \tag{5.4}$$

The interval $[r(T_i^v); d(T_i^v))$ is called the window-length of subtask T_i^v and is denoted by $w(T_i^v)$. The release time $r(T_i^v)$ is the first slot into which T_i could potentially be scheduled and $d(T_i^v)$ is the last such slot. We refer to a window (or sum of k window tasks) of length n slots as an $k - window$ and length of such k windows is defined as L^k.

$$L_i^k = \left\lceil \frac{k \times P_i}{C_i} \right\rceil \tag{5.5}$$

At present, three Pfair scheduling algorithms are known to be optimal on an arbitrary number of processors: PF [14], PD [17], and PD2 [3]. These algorithms prioritize subtasks on an EPDF (Earliest Pseudo Deadline First) basis, but differ in the choice of tie-breaking rules.

The PF Pfair Algorithm: PF was the first Pfair scheduling algorithm that was shown

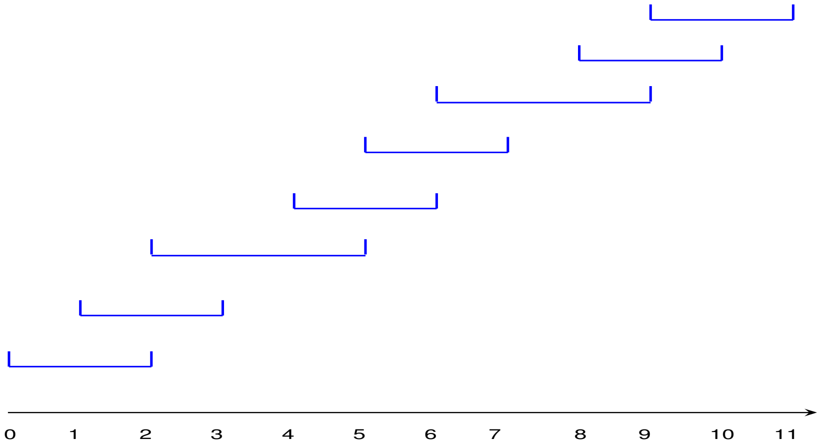


FIGURE 5.2: The Pfair windows of the first job (or 8 window tasks) of a task T_i with weight 8/11 in a Pfair-scheduled system.

to be optimal on multiprocessors. PF prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis and uses several tie-breaking rules when multiple subtasks have the same deadline. If subtasks T_i^a and T_j^b are both eligible at time t, then PF prioritizes T_i^a over T_j^b at t if:

1. $d\left(T_i^a\right) < d\left(T_j^b\right)$
2. a vector of pseudo deadlines is considered to break the tie between two subtasks with same pseudo deadlines.

The PD Pfair Algorithm: PD classify tasks into two categories depending on their weight: a task T_i is light if $u_i < 1/2$, and heavy otherwise. PD scheduling algorithm uses the group deadline to break the tie between subtasks. Consider a sequence $T_i^j, ..., T_i^v$ of subtasks of such a task T_i such that $b(T_i^c) = 1$ and length of window tasks is equal to 2 for all $j \leq c < v$. Scheduling T_i^j in its last slot forces the other subtasks (window tasks) in this sequence to be scheduled in their last slots. The group deadline of a subtask T_i^j, denoted $D(T_i^j)$, is the earliest time by which such a "cascade" must end. Formally, it is the earliest time t, where $t \geq d(T_i^j)$, such that either $(t = d(T_i^c) \wedge b(T_i^j) = 0)$ or $(t+1 = d(T_i^c) \wedge$ length of window =3) for some subtask T_i^c. PD favors subtasks with later group deadlines because scheduling them later can lead to longer cascades, which places more constraints on the future schedule. PD prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis and uses several tie-breaking rules when multiple subtasks have the same deadline. The first tie-breaking rule involves a parameter called the successor bit ,which is defined as follows.

$$b(T_i^v) = \left\lceil \frac{v}{u_i} \right\rceil - \left\lfloor \frac{v}{u_i} \right\rfloor$$

We can now describe the PD priority definition. If subtasks T_i^a and T_j^b are both eligible at time t, then PD prioritizes T_i^a over T_j^b at t if:

Task	C	P
T_1	1	2
T_2	1	3
T_3	1	3
T_4	2	9
T_5	2	9

Table 5.1: Task Parameters

1. $d\left(T_i^a\right) < d\left(T_j^b\right)$ or
2. $d\left(T_i^a\right) = d\left(T_j^b\right) \wedge b\left(T_i^a\right) = 1 \wedge b\left(T_j^b\right) = 0$ or
3. $d\left(T_i^a\right) = d\left(T_j^b\right) \wedge b\left(T_i^a\right) = b\left(T_j^b\right) = 1 \wedge D\left(T_i^a\right) > D\left(T_j^b\right)$

If the priority is not resolved then two more parameters of tasks are considered to break the tie. The first of these is a task's weight. The second is a bit associated with each group deadline which is defined as follows:

1. Bit corresponds to $D(T_i^j) = 1$, if it corresponds to 3-window.
2. Bit correspons to $D(T_i^j) = 0$, if it corresponds to 2-window.

The PD^2 Pfair Algorithm: PD^2 scheduling algorithm has been proved better than PF and PD scheduling algorithms. In PD^2 scheduled system last two tie breaking rules of PD are not considered and tie can be broken arbitrarily. So priorities are defined as follows: If subtasks T_i^a and T_j^b are both eligible at time t, then PD^2 prioritizes T_i^a over T_j^b at t if:

1. $d\left(T_i^a\right) < d\left(T_j^b\right)$ or
2. $d\left(T_i^a\right) = d\left(T_j^b\right) \wedge b\left(T_i^a\right) = 1 \wedge b\left(T_j^b\right) = 0$ or
3. $d\left(T_i^a\right) = d\left(T_j^b\right) \wedge b\left(T_i^a\right) = b\left(T_j^b\right) = 1 \wedge D\left(T_i^a\right) > D\left(T_j^b\right)$

We illustrate the major principle of Pfair scheduling algorithm by an example where we have 5 tasks given in Table 5.1. These tasks are scheduled on two processors as $U(\tau) < 2$. Complete schedule of tasks according to Pfair is presented in Fig:5.3.

5.2.1.2 Local Scheduler and parameters of supertask

A local scheduler schedules the component tasks of a supertask within the slots assigned to T_x^j. We use a Pfair scheduler to schedule the migrating tasks and supertasks, and local scheduling algorithm (EDF) to schedule the component tasks of supertask T_x^j, each time T_x^j is assigned a slot.

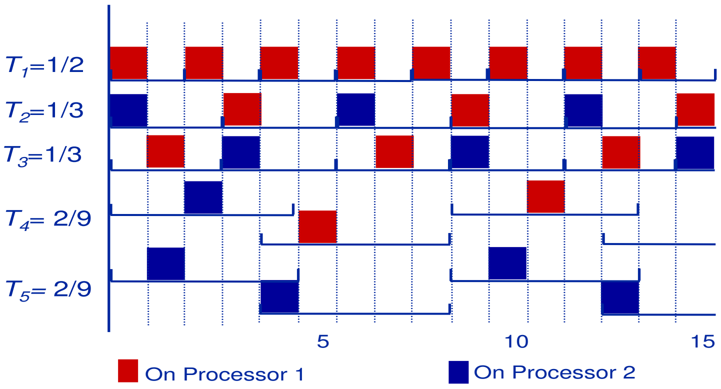


FIGURE 5.3: Scheduling of tasks on two processors by a Pfair scheduler

Note that if there are nj component tasks in supertask T_x^j, the period P_x^j and execution time C_x^j of supertask are defined as follows:

$$P_x^j = lcm_{i=1}^{nj}(P_i) \tag{5.6}$$

$$C_x^j = \sum_{i=1}^{nj} \frac{P_x^j}{P_i} \times C_i \tag{5.7}$$

Let us define two other parameters of supertask T_x^j which are the prime period $\acute{P}_x^j$ and prime execution time $\acute{C}_x^j$ such that:

$$\frac{C_x^j}{P_x^j} = \frac{\acute{C}_x^j}{\acute{P}_x^j}$$

$\acute{P}_x^j$ and $\acute{C}_x^j$ are relative prime numbers.

5.2.2 Schedulability Conditions

Baruah et al. [14], showed that a periodic task set τ is schedulable on M processors iff:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq M \tag{5.8}$$

In our model, we have M supertasks and a set of ng migrating tasks. Total tasks at global level are $ng + M$. Then above condition can be described as:

$$\sum_{j=1}^{M} \frac{C_x^j}{P_x^j} + \sum_{i=1}^{ng} \frac{C_i}{P_i} \leq M \tag{5.9}$$

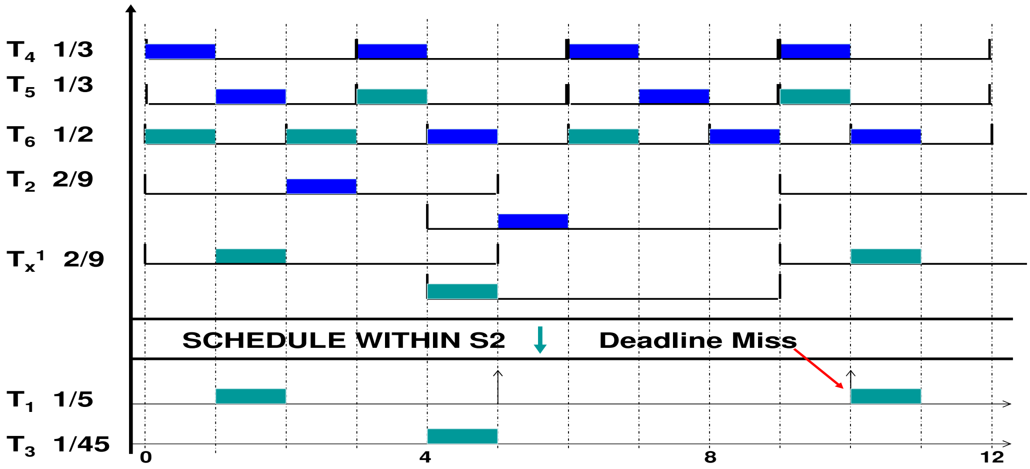


FIGURE 5.4: Scheduling of supertask and component tasks.

All global tasks are guaranteed to meet their deadlines if they are scheduled using the Pfair scheduling algorithm. It means that all ng migrating tasks and M supertasks are guaranteed to meet their deadlines.

5.2.3 Schedulability Conditions for Component tasks

No guarantees are provided for intermediate deadlines of component tasks by Pfair scheduler [53]. Then there is a need to find a way such that component tasks on each processor are guaranteed to meet their deadlines. Holman [33] has demonstrated with an example (Fig:5.4) that component tasks do not respect their deadlines, in general, if component tasks are scheduled in time slots selected by Pfair scheduler.

In this example, the task set is composed of six tasks $T_6 = 1/2, T_4 = 1/3, T_5 = 1/3, T_2 = 2/9, T_1 = 1/5$ and $T_3 = 1/45$. Tasks T_1 and T_3 are grouped to form a supertask $T_x^1 = 1/5 + 1/45 = 2/9$, other tasks are considered as migrating/global tasks. This task set is scheduled on two processors as sum of utilization of all tasks is less than 2. Pfair scheduler schedules four migrating tasks (T_2, T_4, T_5, T_6) and one supertask $T_x^1 = (T_1, T_3)$.

As depicted in Fig:5.4, task T_1 misses its deadline for second job. It implies that component tasks miss their deadlines when they are scheduled in only those time slots where supertask is selected by global Pfair scheduler. The sufficient condition to provide deadline guarantees for task T_i is that it must be allocated C_i units of processor time in each interval $[k.P_i; (k+1).P_i)$. Component task T_i (partitioned on processor π_j) is selected to execute (by local scheduler) if its corresponding supertask T_x^j is selected by the global scheduler.

As component tasks are scheduled only in windows of supertask, then necessary/sufficient conditions for feasibility of component tasks T_i are defined as follows.

1. ***(Necessary Condition)*** The pseudo release of $\left(\sum_{i=1}^{nj} \left\lfloor \frac{t}{P_i} \right\rfloor \times C_i\right)^{th}$ window tasks must be earlier than t to provide deadline guarantees to component tasks.

2. ***(Number/Length of window task)*** Task T_x^j must have at least C_i window tasks in each interval $[k.P_i; (k+1).P_i)$ to provide deadline guarantees to task T_i. The release times and deadlines of these window tasks must also lie in the same interval.

3. ***(Embedded task Periods)*** Task T_x^j must have $1 + \left(\sum_{v=1}^{nj} \left\lfloor \frac{P_i}{P_v} \right\rfloor \times C_v\right)$ effective window tasks in any interval to ensure presence of C_i window tasks lying in interval $[k.P_i; (k+1).P_i)$ such that release times and deadlines of these window tasks also lie in same interval (k is an integer).

5.2.3.1 Necessary Condition

At any instant t integer multiple of basic time quanta, the demand bound function $DBF(T_i, t)$ defined by Baruah[16] is an approach to establish the necessary condition for feasibility analysis of component tasks on processor π_j within interval $[0, t)$.

$$\sum_{T_i \in \tau^j} DBF(T_i, t) = \sum_{i=1}^{nj} \left\lfloor \frac{t}{P_i} \right\rfloor \times C_i \tag{5.10}$$

The amount of time (i.e., $\sum_{T_i \in \tau^j} DBF(T_i, t)$) is distributed over the window tasks scheduled in the interval $[0; t)$, at a rate of one time unit per window task. Thus $\sum_{T_i \in \tau^j} DBF(T_i, t) = z$ denotes the number of window tasks that must be scheduled in interval $[0; t)$ to provide deadline guarantees for component tasks. In this case, demand bound function $\sum_{T_i \in \tau^j} DBF(T_i, t) = z$ represents the z^{th} window task of supertask.

In the best case scenario, the z^{th} window task can finish its execution in its earliest slot (i.e., Equation(5.3)) or in worst case scenario it will finish its execution in the last slot of window, which is defined as Equation(5.4). It implies that necessary condition to provide deadline guarantees to component task is that release time of z^{th} window task occurs earlier than time t.

The local scheduling of component tasks is feasible iff:

$$\forall t \geq 0 : \sum_{T_i \in \tau^j} DBF(T_i, t) = z : \; r(T_x^{j,z}) < t \tag{5.11}$$

where $r(T_x^{j,z})$ represents the release time of z^{th} window task of supertask T_x^j. To develop necessary condition, we start by verifying if pseudo deadline of z^{th} window task (i.e., $d(T_x^{j,z})$) arrives a time earlier than t, i.e.:.

$$d(T_x^{j,z}) = \left\lceil \frac{z \times P_x^j}{C_x^j} \right\rceil \leq t$$

Replacing z and C_x^j by their definitions (Equation 5.11 and Equation5.7):

$$d(T_x^{j,z}) = \left\lceil \frac{P_x^j \times \sum_{i=1}^{nj} \left\lfloor \frac{t}{P_i} \right\rfloor \times C_i}{\sum_{i=1}^{nj} \frac{P_x^j}{P_i} \times C_i} \right\rceil$$

$$d(T_x^{j,z}) = \left\lceil \frac{\sum_{i=1}^{nj} \left\lfloor \frac{t}{P_i} \right\rfloor \times C_i}{\sum_{i=1}^{nj} \frac{C_i}{P_i}} \right\rceil$$

As

$$\left\lfloor \frac{t}{P_i} \right\rfloor \leq \frac{t}{P_i}$$

we get:

$$d(T_x^{j,z}) \leq \left\lceil \frac{t \times \sum_{i=1}^{nj} \frac{C_i}{P_i}}{\sum_{i=1}^{nj} \frac{C_i}{P_i}} \right\rceil$$

$$d(T_x^{j,z}) \leq \lceil t \rceil \Rightarrow d(T_x^{j,z}) \leq t$$

It implies that the above necessary condition is always met (for any task set where utilization of task set is less than M) for any set of component tasks and supertask.

5.2.3.2 Number/Length of window task

A component task $T_i \in \tau^j$ respects its deadline, if there are C_i window tasks of T_x^j with their release times and deadline lying in interval $[P_i.k; (k+1)P_i)$. Global scheduler selects supertask T_x^j for at least C_i times during this interval to execute on a processor, and hence task T_i is guaranteed to meet its deadline constraints.

Theorem 5.1. *Each supertask T_x^j has at least C_i effective window tasks in any interval of length P_i units of time for every component task T_i ($T_i \in \tau^j$).*

Proof. The release time of 0^{th} window task of supertask T_x^j is zero and pseudo deadline of C_i^{th} window task is defined as:

$$d(T_x^{j,C_i}) = \left\lceil \frac{C_i \times P_x^j}{C_x^j} \right\rceil$$

The difference between the pseudo deadline of C_i^{th} window task and release time of 0^{th} window task is given by:

$$L^{C_i} = \left\lceil \frac{C_i \times P_x^j}{C_x^j} \right\rceil - 0$$

$$L^{C_i} = \left\lceil \frac{C_i \times P_x^j}{C_x^j} \right\rceil$$

As

$$\frac{P_i}{C_i} \geq \frac{P_x^j}{C_x^j} \Rightarrow$$

$$L^{C_i} \leq \left\lceil \frac{C_i \times P_i}{C_i} \right\rceil \tag{5.12}$$

$$L^{C_i} \leq \lceil P_i \rceil$$

as P_i is an integer $\Rightarrow$

$$L^{C_i} \leq P_i$$

It demonstrates that there are at least C_i effective window tasks in any interval of length P_i. □

If there is one effective window task in an interval $[t_1; t_2)$, it implies that either the window task has its release time and deadline within given interval $[t_1, t_2)$, or there are two window tasks; one has its release time before t_1 and deadline of second window task is after t_2, but number of slots of first window task before instant t_1 are less than or equal to number of slots of second window task after t_2

The presence of C_i effective window tasks in any interval of length P_i units of time does not guarantee that component tasks respect their deadlines, as it does not ensure presence of C_i window tasks in all interval $[P_i.k; (k+1).P_i)$ such that release times and deadlines of C_i window tasks also lie in the given interval.

Theorem 5.2. *Allocation of C_i effective window tasks in an interval of length P_i units of time does not ensure presence of C_i window tasks (window tasks that have their release times and deadlines in the given interval) in interval $[P_i.k; (k+1).P_i)$.*

Proof. Let for some value of $n+1$ such that

$$r(T_x^{j,n+1}) < k.P_i \wedge d(T_x^{j,n+1}) > k.P_i$$

$$\Rightarrow r(T_x^{j,n+2}) \geq k.P_i$$

let

$$L^{C_i+1} > P_i \wedge L^{C_i} \leq P_i$$

$$\Rightarrow d(T_x^{j,n+C_i+1}) > (k+1).P_i$$

The expression $d(T_x^{j,n+C_i}) < (k+1).P_i$ illustrates that there are C_i window tasks before the instant $(k+1).P_i$, but as $r(T_x^{j,n+1}) < k.P_i$ and $d(T_x^{j,n+C_i+1}) > (k+1).P_i)$, it implies that there are $(C_i - 1)$ window tasks with their release times and pseudo deadlines within the interval $[P_i.k; (k+1).P_i))$. □

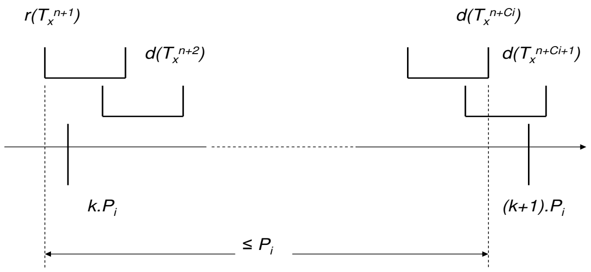


FIGURE 5.5: Number of windows in an interval.

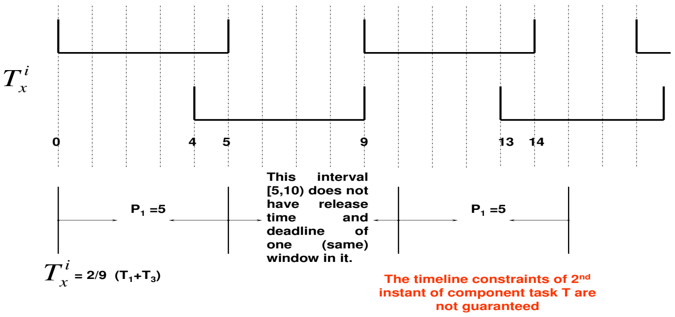

T_x^i
0
4
5
9
13
14
P_1 =5
This interval [5,10) does not have release time and deadline of one (same) window in it.
P_1 =5
T_x^i = 2/9 (T_1+T_3)
The timeline constraints of 2nd instant of component task T are not guaranteed

FIGURE 5.6: Window tasks defined for supertask.

In Fig:5.6 related to Fig:5.4, we find that there is at least one effective window task for each interval of length 5 (P_1) time units, but there is no window task with its release time and deadline lying in the second interval $[5, 10)$. It implies that supertask T_x^1 does not guarantee that 2^{nd} job of component task T_1 can be executed in interval $[5, 10)$. For example, it happens when T_x^1 is selected to execute at instant 4 for its second window, and T_x^1 is not selected in its earliest slot for execution of third window task, then task T_1 misses its deadline at 10.

Theorem 5.3. *Allocation of at least $(C_i + 1)$ effective window tasks in any interval of length P_i units of time guarantees that there will be C_i window tasks with their release times and deadlines within the interval $[P_i.k, (k + 1).P_i)$.*

Proof. Suppose that the $n + 1^{th}$ window task of supertask T_x^j has its release time before $k.P_i$ as follows:

$$r(T_x^{j,n+1}) < k.P_i \wedge d(T_x^{j,n+1}) > k.P_i$$

$$\Rightarrow r(T_x^{j,n+2}) \geq k.P_i$$

As

$$L^{C_i+1} \leq P_i$$

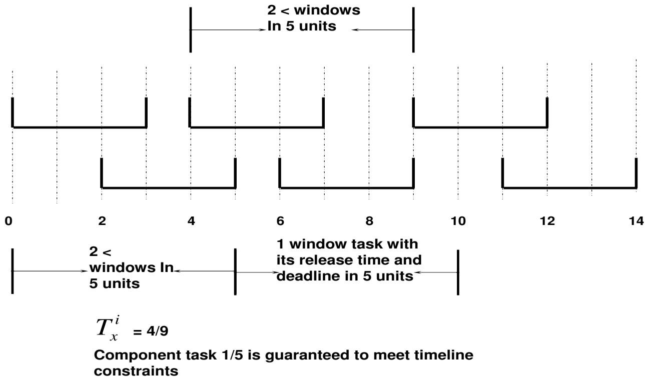


FIGURE 5.7: Schedulability condition for component task.

$$\Rightarrow d(T_x^{j,n+C_i+1}) \leq (k+1).P_i$$

It implies that there are at least C_i window tasks with their pseudo release times and pseudo deadlines within the interval $[P_i.k, (k+1).P_i)$. □

As illustrated in, Fig:5.7, we have at least two effective window tasks between any two points of interval length 5. It insures the presence of at least one window task (with its release time and deadline) in any interval $[k.5; (k+1).5)$.

- Properties of Supertask

It is demonstrated that if supertask is allocated C_i+1 window tasks for P_i units of time, then component task T_i always respects its deadline. The key point now is to establish a relation between supertask and that of its component tasks such that supertask T_x^j has $C_i + 1$ window tasks for P_i time units.

L^{C_i+1} is written as follows (as in Equation(5.12)):

$$L^{C_i+1} \leq \left\lceil \frac{(C_i+1) \times P_x^j}{C_x^j} \right\rceil$$

if

$$\frac{P_i}{C_i+1} \geq \frac{P_x^j}{C_x^j}$$

then

$$L^{C_i+1} \leq \left\lceil \frac{(C_i+1) \times P_i}{C_i+1} \right\rceil$$

$$L^{C_i+1} \leq \lceil P_i \rceil$$

$$L^{C_i+1} \leq P_i$$

This result shows that there exists a weight bound relation between weight of supertask T_x^j and those of its component tasks $T_i \in \tau^j$ which, if provided, guarantees the presence of C_i window tasks within interval $[k.P_i; (k+1).P_i)$. Therefore, if component tasks $T_i \in \tau^j$ and supertask T_x^j respect these two following conditions, then supertask as well as component tasks are guaranteed to meet their deadlines.

1. Condition 1: $C_x^j / P_x^j \leq 1$
2. Condition 2: $P_i/(C_i + 1) \geq P_x^j / C_x^j$ where $T_i \in \tau^j$

5.2.3.3 Embedded task Periods

In real cases, period of one task is embedded into period of another task, (i.e., period of task T_v is smaller than period of task T_i) then presence of $C_i + 1$ window tasks in interval of length P_i is not sufficient to provide time line guarantees to both components tasks T_v and T_i. In this scenario, the second condition is modified as given below:

$$Condition \;\; 2A: \qquad \frac{P_i}{1 + \sum_{v=1}^{nj} \left\lfloor \frac{P_i}{P_v} \right\rfloor \times C_v} \geq \frac{P_x^j}{C_x^j}$$

5.2.3.4 Exemptions

If the weight of component task and that of its corresponding supertask satisfy the condition 2A, then all component tasks are guaranteed to meet their deadline constraints. Component tasks are exempted from the second condition for which the following condition holds true:

$$P_i = k.\acute{P}_x^j$$

where k is a positive integer.

Theorem 5.4. *If a component task has the above mentioned relation with its supertask then this component task respects its deadline constraints.*

Proof. For some value of f, if

$$d(T_x^{j,f}) = z.P_i$$

$$\Rightarrow r(T_x^{j,f+1}) = z.P_i$$

z is a positive integer.

Two consecutive window tasks do not overlap at the period boundaries of the supertask. It implies that the number of effective window tasks in interval $[k.P_i; (k+1).P_i)$ is equal to the number of such window tasks which have their release times and deadlines in the interval as well. □

Example 5.1. *We illustrates these principles through an example. We consider the same example used by Holman [33], and show that deadline guarantees of component tasks are provided, if component tasks and corresponding supertask satisfy both conditions (1 and 2A). We have constructed two supertasks equal to the number of processors in the system, and each supertask T_x^j and its component tasks satisfy both conditions defined in previous section. Tasks T_4 = 1/3, and T_5 = 1/3 are grouped to form a supertask T_x^1, and we can verify that:*

$$(Weight\ of\ T_x^1 = 2/3) < 1$$

We have $P_4 = P_5 = P_x^1$, thus it implies that second condition is exempted. Task T_1 = 1/5, task T_2 = 2/9 and task T_3 = 1/45 are grouped to create a supertask T_x^2 and we can verify that:

$$(Weight\ of\ T_x^2 = 1/5 + 2/9 + 1/45) < 1$$

$$\frac{P_1}{1 + \left\lfloor \frac{P_1}{P_1} \right\rfloor \times C_1 + \left\lfloor \frac{P_1}{P_2} \right\rfloor \times C_2 + \left\lfloor \frac{P_1}{P_3} \right\rfloor \times C_3} \geq \frac{P_x^2}{C_x^2}$$

$$5/2 > 9/4$$

$$\frac{P_2}{1 + \left\lfloor \frac{P_2}{P_2} \right\rfloor \times C_2 + \left\lfloor \frac{P_2}{P_1} \right\rfloor \times C_1 + \left\lfloor \frac{P_2}{P_3} \right\rfloor \times C_3} \geq \frac{P_x^2}{C_x^2}$$

$$9/4 = 9/4$$

As $P_3 = 5 \times P_x^2$, it implies second condition is exempted for task T_3

In Fig:5.8, it is illustrated that component tasks respect their deadlines when these component tasks and corresponding supertask satisfy both conditions.

Hierarchical scheduling requires a partitioning of tasks into M sets to form the supertask. Partitioning approach plays an important role for providing established relation between supertask and its component tasks. First, we provide few details about different EDF based partitioning strategies.

5.2.4 Reweighting Rules

In the previous subsections, we have established a sufficient condition which is a relation between weight of a supertask and that of local task, if this condition is respected then there is no need to increase in weight of supertask for feasible schedule of local tasks. If this condition is verified for weight of any local and that of supertask, then reweighting of supertask is required to ensure deadline guarantees to local tasks. Rules for reweighting of supertask are described below.

Rule 1: if $u_x^j = 1$, then reweighting is unnecessary.

As demonstrated in above sections that if supertask and local task have established relation of weight in them then there is no need to reweighting. So:

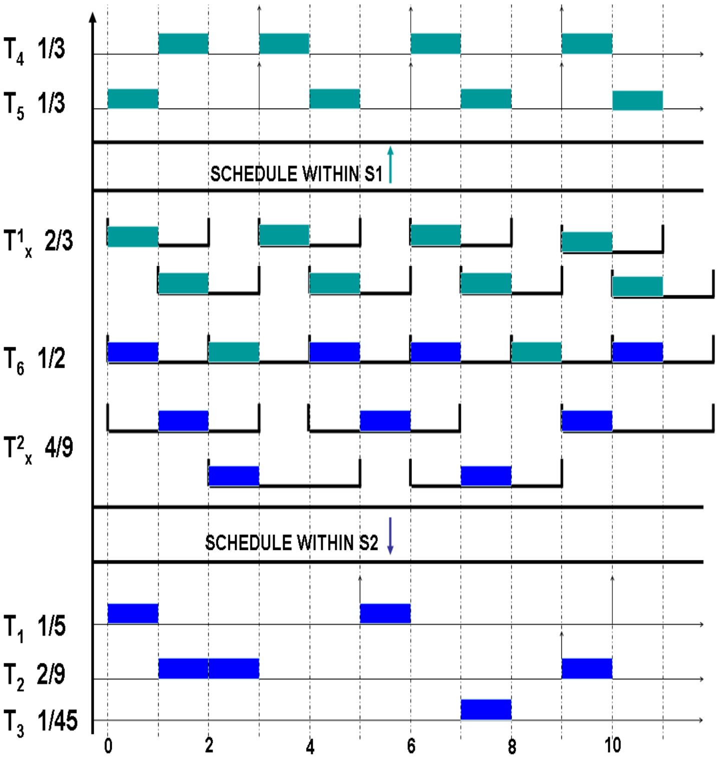


FIGURE 5.8: Demonstration of scheduling of supertasks and component tasks.

Rule 2: if

$$\forall\ T_i \in \tau^j \qquad \frac{P_i}{c + \sum_{v=1}^{nj} \left\lfloor \frac{P_i}{P_v} \right\rfloor \times C_v} \geq \frac{P_x^j}{C_x^j}$$

, then reweighting is unnecessary
where c = 0, if $P_i = k.\acute{P}_x^j$ and c = 1 otherwise.

If Rule 1 and Rule 2 are not satisfied, then we need to increase the weight of supertask to ensure deadline guarantees for local tasks.

Rule 3:

$$\hat{u}_x^j = \frac{1}{min_{\forall T_i \in \tau^j} \left[\frac{P_i}{c + \sum_{v=1}^{nj} \left\lfloor \frac{P_i}{P_v} \right\rfloor \times C_v} \right]}$$

If we consider the above example, we found that Rule 2 is respected, hence there is no need to increase the weight of supertask. According to approach presented by Holman, there is a need to calculate the new increased weight of supertask, and for above example the new increase weight calculated by Holman is $\frac{5}{9}$. Hence, $\frac{5}{9} - \frac{4}{9} = \frac{1}{9}$ is wasted for nothing.

5.2.5 Results

We have compared the performance of our proposed reweighting rules with on prosed by Anand et al (Table 5.2). We observed that in most of the cases reweighting was

Number of Tasks	Inflation	
	Anad et al.	Farooq et al.
4	37%	4%
5	22%	3%
6	27%	6%
7	14%	09%
8	23%	2%

Table 5.2: Comparison of Our proposed Reweighting rules with those proposed by Anand et al.

unnecessary according to our approach but weight of supertask is still increased by Anand et al. We have also observed that even when there is a need to increase the weight of supertask, inflation introduced by our approach is much lower than that of proposed by Anand et al.

5.2.6 Partitioning of tasks

Several polynomial-time heuristics have been proposed for task partitioning based on bin-packing approaches. We describe three of them below. While describing these heuristics, we assume that there are M processors numbered from 1 to M, and n tasks numbered from 1 to n that need to be scheduled. The tasks are not assumed to be arranged in any specific order. We can obtain different variants of partitioning approaches by sorting the tasks in a specific order before applying the heuristics. The schedulability test associated with the chosen uniprocessor scheduling algorithm can be used as an acceptance test to determine whether a task can "fit" on a processor. For instance, under EDF scheduling, a task will fit on a processor as long as the total utilization of all tasks assigned to that processor does not exceed unity.

5.2.6.1 Next Fit (NF)

In this approach, all the processors are considered in order, and we assign to each processor as many tasks as can fit on that processor.

5.2.6.2 First Fit (FF)

FF improves upon NF by also considering earlier processors during task assignment. Thus, each task is assigned to the first processor that can accept it.

5.2.6.3 Best Fit (BF)

In this approach, each task is assigned to a processor that (i) can accept the task, and (ii) will have minimal remaining spare capacity after its addition.

A complementary approach to BF is worst fit (WF), in which each task is assigned to a processor that (i) can accept the task, and (ii) will have maximal remaining spare capacity after its addition. Though this approach does not try to maximize utilization, it results in a partitioning in which the workload is equally balanced among the different processors.

Our objective is not to devise a new heuristic of partitioning of tasks but to find such technique which ensures that there is always a weight bound relation between supertask and those of component tasks. We know that

$$P_i/(C_i) \geq P_x^j/C_x^j$$

but
$P_i/(C_i+1)$ may or may not be greater than P_x^j/C_x^j. If

$$P_i \rightarrow \infty$$

then

$$P_i/(C_i) = P_i/(C_i+1)$$

and consequently

$$P_i/(C_i+1) \geq P_x^j/C_x^j$$

It implies that if tasks are sorted in decreasing order of their periods, then chances of having established relation between weight of supertask and that of component task is higher. Larger is the period of a task, better are the chances of having established relation between supertask and those of component tasks. Therefore, we propose to sort tasks in decreasing order of their period lengths before partitioning. If two task have same periods then task with shorter execution time is given higher priority. According to this approach, when tasks with shorter periods are partitioned, sum of utilization of tasks, already partitioned on processor, is high which increases the chances of verifying the second condition even for tasks with shorter periods.

In this section, we have extended the work of Moir et al. which is based on supertasking approach using Pfair scheduling algorithm as a global scheduler. It not only increases the preemptions and migrations of tasks, but it imposes also certain conditions to be verified between supertask and component tasks to provide deadline guarantees to task. In the next section, we propose to use ASEDZL scheduling algorithm as a global schedule, and illustrate that it does not impose any condition on weights of supertask and that of component tasks.

5.3 Hierarchical Scheduling Based on Slack Anticipation Approach

This approach is similar to approach described in earlier section, as it groups partitioned task one a processor π_j to form a supertask T_x^j, and M supertasks among other global tasks are scheduled globally. Local tasks are selected to execute, when their corresponding supertask is selected by global scheduler. A local scheduler on each processor and a global scheduler is used to provide deadline guarantees to all tasks. The main advantage of using ASEDZL as a global scheduler is its low runtime complexity as compared to Pfair, and it does not impose any condition on characteristics of supertask and migrating tasks to provide deadline guarantees.

5.3.1 Approach Description

There is a scheduler on every processor (referred as local scheduler here after) and a global scheduler on top of all local schedulers. Global tasks and the M supertasks are placed and sorted in $TaskQueue$, as explained in previous chapter, and are scheduled by ASEDZL scheduler. Every local scheduler on a processor is invoked whenever corresponding supertask is selected by ASEDZL scheduler. We explain both of these algorithms (global and local) in detail in following subsections.

5.3.1.1 Local Scheduler

We use an EDF based scheduler on each processor to schedule local tasks (component tasks) to execute. Conventionally, EDF selects the first task to execute from the ready queue which is a sorted list of tasks. Tasks with non zero remaining execution time are placed in this queue and sorted in the increasing order of their deadlines. However, we propose to use $TaskQueue$ instead of a ready queue to place and sort tasks at local scheduler level as well.

- Local $TaskQueue$

As detailed in previous chapter, tasks are sorted and placed in $TaskQueue$ (different from ready queue) according to the closeness of their deadlines. The tasks with zero remaining execution times are also placed in $TaskQueue$, and sorted according to their deadlines. The absolute deadline of a task is not updated until it is released for its next instant. This $TaskQueue$ (Fig:5.9) is updated only when a task is released for its next instant, and is not updated when a task completes its execution. Hence, the number of tasks in $TaskQueue$ remains constant.

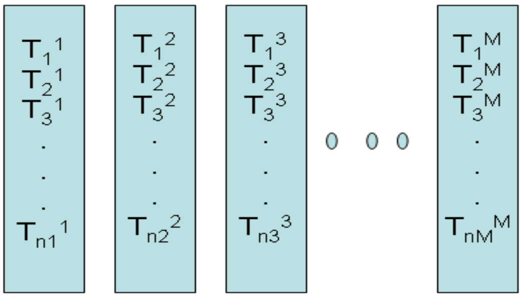


FIGURE 5.9: Local *TaskQueues*

5.3.1.2 Local Release instants

The release time of task T_i partitioned on a processor $(\pi)_j$ is called a local scheduling event and the k^{th} local scheduling event is represented as R_k^j. Local release instants are calculated to define parameters of supertask T_x^j.

5.3.1.3 Supertask

As explained in previous section, local tasks (component tasks) partitioned on a processor $(\pi)_j$ are grouped together to form a supertask T_x^j, which is then scheduled as an ordinary global task. Whenever global scheduler selects supertask T_x^j, the local scheduler selects the highest priority tasks from local *TaskQueue* to execute. Supertask constructed from component tasks has static as well as dynamic parameters.

- Static parameter

Static parameter of supertasks T_x^j is its weight u_x^j (utilization) is equal to the sum of weights of all component tasks partitioned on a processor $(\pi)_j$.

$$u_x^j = \sum_{i=1}^{nj} u_i \tag{5.13}$$

- Dynamic parameters

Dynamic parameters of supertask T_x^j are its deadline d_x^j and worst case execution time C_x^j. These dynamic parameters of supertask T_x^j are calculated when a component task is released for its next instant. The deadline d_x^j of supertask T_x^j is defined as the deadline of the most imminent deadline of a local task on processor $(\pi)_j$ with zero or non zero remaining execution time. Worst case execution time and deadlines are calculated on all local release instants. At local release instant R_{k-1}^j, deadline d_x^j is calculated as follows:

$$d_x^j = d_f^j \tag{5.14}$$

where $d_f^i = R_k^j$ represents the absolute deadline of first task in local *TaskQueue*.

The worst case execution time C_x^j of supertask T_x^j is calculated as follows:

$$C_x^j = (R_k^j - R_{k-1}^j).u_x^j \tag{5.15}$$

5.3.1.4 Global Scheduler

According to this algorithm, the objective is that the slacks of the M tasks, which are considered M high priority tasks according to EDF, are filled by subsequent tasks before these slacks appear in a schedule. To ensure that slacks of these M high priority tasks are filled, tasks are selected from $TaskQueue$ and not from ready queue.

- Global $TaskQueue$

Like local schedulers, global scheduler also has a $TaskQueue$, which contains the M supertasks from each processor along with global tasks. Global queue is again sorted according to the closeness of deadlines of all tasks it contains. Global scheduler uses ASEDZL scheduling approach. Using the same notations described in previous chapter, we elaborate the mechanism of ASEDZL for accessing this global $TaskQueue$. Tasks in global $TaskQueue$ are divided into two groups:
- T^{EDF} Task set
- T^S Task set

5.3.1.5 Release instants for ASEDZL

Release time of a global or supertask represents the release instant for ASEDZL scheduler, and is represented by R_k. Laxities of M T^{EDF} tasks are filled between two consecutive scheduling events which are release times of all tasks. Time difference between two scheduling events is called *interval_length*. If between two successive release instants, any M T^{EDF} (may include both global tasks and supertasks) task has execution requirement less than *interval_length*, then one or more than one subsequent task(s) are selected to fill this laxity because scheduler needs to execute tasks between two release instants for time units equal to :

$$TU = M \times interval\_length \qquad (U(\tau) = M) \tag{5.16}$$

ASEDZL algorithm assigns these tasks virtual deadlines and local execution requirements to schedule on M processors.

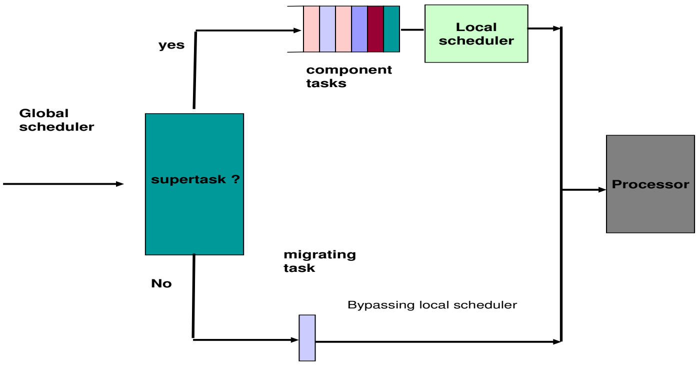


FIGURE 5.10: Interaction between Global and Local scheduler.

5.3.2 Schedulability Analysis

Our proposed algorithm is an optimal algorithm in the sense that, for a given task set with a cumulative utilization less than or equal to M, no task misses its deadline.i.e.,

$$\sum_{v=1}^{ng} \frac{C_v}{P_v} + \sum_{j=1}^{M} \sum_{k=1}^{nj} \frac{C_k}{P_k} \leq M \tag{5.17}$$

5.3.2.1 Schedulability analysis for global and supertasks

Schedulability analysis can be divided into two parts, one is schedulability at global level and other is at local level. As we have already proved the optimality of ASEDZL for scheduling of tasks at global level, it ensures that all global tasks and supertasks are guaranteed to meet their deadlines constraints.

5.3.2.2 Schedulability of Partitioned tasks

For partitioned tasks, EDF based local scheduler is used on each processor. EDF is an optimal scheduling algorithm i.e., if utilization of task set is less than 100%, then EDF provides guarantees to all component tasks. But in supertasking approach, scheduling of local tasks is dependent on decisions of global scheduler (Fig:5.10). Local scheduler selects local tasks to execute, if its corresponding supertask is selected by the global scheduler.

ASEDZL provides fairness only at deadline boundaries. It implies that supertask T_x^j is guaranteed to be allocated C_x^j time units before its deadline d_x^j. Allocation of processor time to supertask T_x^j until release instant R_k^j is calculated as follows:

$$Allocation = (R_k^j - R_0^j).u_x^j \tag{5.18}$$

$(R_k^j - R_0^j)$ represents the time until local release instant R_k^j.
Replacing $(R_k^j - R_0^j)$ with time t, and u_x^j by its definition

$$Allocation = t.\sum_{i=1}^{nj}\frac{C_i}{P_i} \tag{5.19}$$

- Minimum Demand

To provide deadline guarantees, we need to ensure that time allocated to supertask by global scheduler is more than or equal to minimum demand of component tasks until time t (t represents the local release instants on processor π_j). We use demand bound function $DBF(T_i, t)$ analysis (Equation(5.20)) to calculate the minimum processor demand for deadline guarantees for component tasks $T_i \in \tau^j$.

A real time task set is schedulable under EDF if and only if $\sum_{T_i \in \tau^j} DBF(T_i, t)$ is less than or equal to allocation of processor time (represented by *allocation*) to T_x^j by global scheduler:

$$\sum_{T_i \in \tau^j} DBF(T_i, t) = \sum_{i=1}^{nj}\left\lfloor\frac{t}{P_i}\right\rfloor .C_i \tag{5.20}$$

Allocation to supertask tasks until t is given by:

$$Allocation = t. \times \sum_{i=1}^{nj}\frac{C_i}{P_i}$$

$$Allocation = \sum_{i=1}^{nj}\left\lfloor\frac{t}{P_i}\right\rfloor \times C_i + \sum_{i=1}^{nj}\left[t - \left(\left\lfloor\frac{t}{P_i}\right\rfloor \times P_i\right)\right] \times \frac{C_i}{P_i} \tag{5.21}$$

As $\left[\left(t - \left\lfloor\frac{t}{P_i}\right\rfloor \times P_i\right) \geq 0\right]$, it demonstrates that the processor demand of partitioned tasks on a processor π_j over any time interval $[0, t)$ is less than or equal to the *allocation* of processor time to corresponding supertask T_x^j by global scheduler.

- Maximum Demand

We also need to ensure that whenever a supertask is selected by global scheduler, then there must be at least one local/component task to be executed.

Now we analyze the maximum possible processor demand D_{max} of task set (component tasks) partitioned on a processor π_j until t:

$$D_{max} = \sum_{i=1}^{nj}\left\lfloor\frac{t}{P_i}\right\rfloor \times C_i + \sum_{i=1}^{nj}\min\left[C_i, \left(t - \left(\left\lfloor\frac{t}{P_i}\right\rfloor \times P_i\right)\right)\right] \tag{5.22}$$

If the maximum possible processor demand of local tasks partitioned on a processor is greater than or equal to the *allocation* of processor time to supertask T_x^j, then it ensures that whenever supertask is selected by global scheduler there is at least one local task with non zero remaining execution time (Fig:5.11). Equation(5.21) and Equation(5.22) demonstrate that allocation to supertask T_x^j by global scheduler is less than maximum demand of component tasks on processor π_j over time interval[0,t).

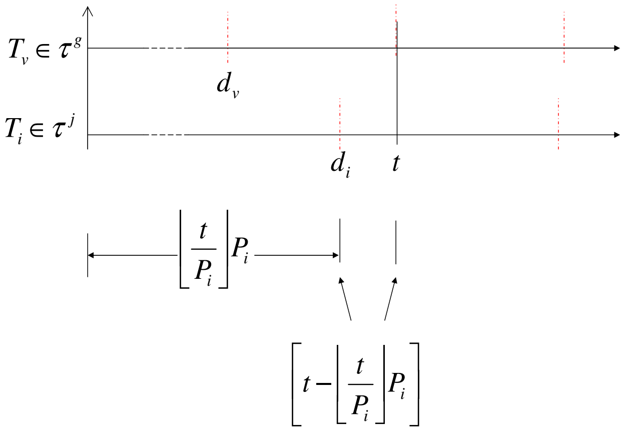

$T_v \in \tau^g$
d_v
$T_i \in \tau^j$
d_i
t
$\left\lfloor \frac{t}{P_i} \right\rfloor P_i$
$\left\lceil t - \left\lfloor \frac{t}{P_i} \right\rfloor P_i \right\rceil$

FIGURE 5.11: Maximum demand of tasks on processor

Example 5.2. *We illustrate with a simplified example to ease the understanding of our approach. Sum of utilization of all tasks given in Table 5.3 is 300%. So we need at least 3 processors to schedule these tasks. Firstly, tasks are partitioned; task T_1 is partioned on first processor, T_2 and T_3 are partitioned on processor 2 while task T_4 and T_5 are partitioned on processor 3. Tasks T_6 and T_7 are global tasks.*

Task	C	P	π
T_1	3	4	π_1
T_2	2	4	π_2
T_3	2	5	π_2
T_4	1	4	π_3
T_5	4	10	π_3
T_6	2	5	g
T_7	3	10	g

TABLE 5.3: Task Parameters

Three supertasks are constructed from the three sets of local tasks. As there is only one task on first processor then supertask T_x^1 is same as T_1 while weights of T_x^2 and T_x^3 are calculated as follows:

$$u_x^2 = u_2 + u_3 = 50 + 40 = 90\%$$

$$u_x^3 = u_4 + u_5 = 25 + 40 = 65\%$$

We detail below the steps of proposed algorithm:

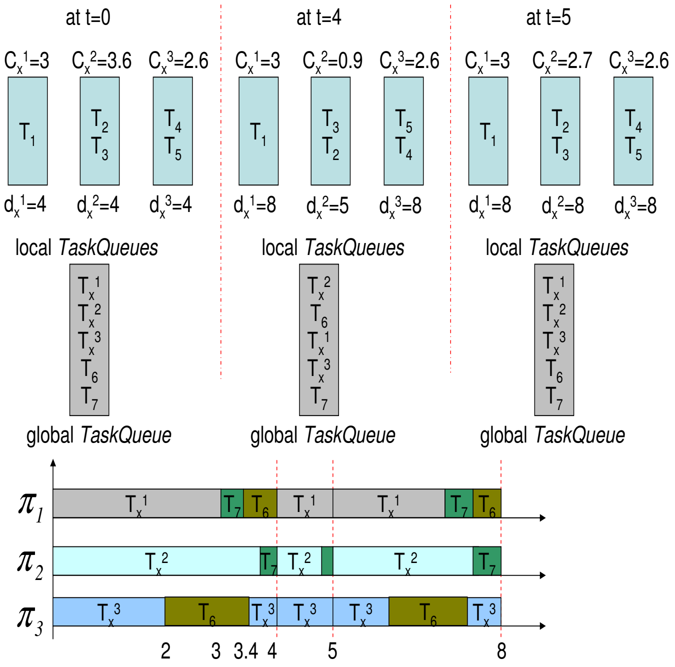


FIGURE 5.12: Demonstrating Example

1. *At t=0, the earliest deadline of tasks in the global TaskQueue is 4. So* $t = 4$ *is defined as next release instant. The earliest deadline in all local queues is also at 4. The dynamic execution requirements of supertasks* T_x^1*,* T_x^2 *and* T_x^3 *are calculated as follows:*

$$C_x^1 = 4 \times \frac{3}{4} = 3$$
$$C_x^2 = 4 \times \frac{9}{10} = 3.6$$
$$C_x^3 = 4 \times \frac{65}{100} = 2.6$$

Then tasks are scheduled according to ASEDZL. T_x^1*,* T_x^2*,* T_x^3 *are highest priority tasks so they are selected to execute.*

2. *At t=2, the virtual laxity of* T_6 *becomes zero, hence* T_6 *replaces task* T_x^3*.*

3. *At t=3,* T_x^1 *completes its execution and is replaced by* T_7*.*

4. *At t=3.4, laxity of task* T_x^3 *becomes zero so it preempts task* T_6 *and start executing on* π_3*, as* T_6 *also had zero virtual laxity so it preempts task* T_7*.*

5. *At t=3.6* T_x^2 *completes its execution and is replaced by* T_7*.*

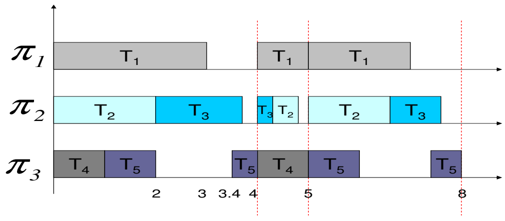


FIGURE 5.13: Execution of Local Tasks

6. *At t=4, global TaskQueue and local TaskQueue are updated and next release instant R_1 is defined which is 5. As local TaskQueue are updated as well, new deadlines and execution requirements of supertasks are calculated. The difference between current local release instant and next local release instants for T_x^1, T_x^2 and T_x^3 are 4,1 and 4 respectively.*

$$C_x^1 = 4 \times \frac{3}{4} = 3$$
$$C_x^2 = 1 \times \frac{9}{10} = 0.9$$
$$C_x^3 = 4 \times \frac{65}{100} = 2.6$$

Supertasks and global tasks are scheduled in the same way as explained in previous steps. Local scheduler selects the local tasks to execute when corresponding supertask is selected by global scheduler. Local scheduling of component tasks on processors is shown in Fig:5.13.

5.4 Conclusions

We have demonstrated that in hierarchical scheduling policy where Pfair is used as a global scheduling algorithm, component tasks respect their deadlines, if relation between weight of supertask and those of component tasks is as per condition 2A.

We have formally established the sufficient condition to provide deadline guarantees to its component tasks i.e., minimum number of window tasks that a supertask T_x must have (in specified interval). We have derived the relation to be held to provide these minimum number of window tasks in supertask. We have not introduced an inflation factor as proposed by Holman that minimizes the utilization of the system.

We have presented another optimal real-time hierarchical scheduling algorithm for multiprocessors where ASEDZL is used as global scheduler. Unlike Pfair, execution requirements and time periods of tasks can have any arbitrary value. We have also demonstrated that no additional condition is imposed on weights of supertask and that of component tasks for deadline guarantees. It is also illustrated that number of preemptions are less than any other scheduling algorithm of the domain, as it is not mandatory to execute all global tasks between any two release instants. It is the case in EKG (**E**DF with task splitting and **K** processors in a **G**roup.) algorithm [7], where all global tasks and supertasks are executed between two release instants. We have also illustrated by an example that our algorithm does not increase the scheduling events. In very near future, we are aiming to simulate our proposed algorithm on a tool developed in context of PHERMA project. This tool helps to simulate global scheduling algorithms as well as hierarchical scheduling algorithms. Moreover, it also provides options which are some real parameters of an architecture to play with , hence this tool gives results much better than just simulation results.

Chapter 6

Conclusions

The real time scheduling algorithms like EDF, LLF and MUF are optimal in case of mono processor system, and Pfair and LLREF have optimal schedulable bounds in case of multiprocessor systems. But, Pfair and LLREF algorithms offer optimal schedulable bound at the cost of increased scheduling complexity and increased scheduling events.

The thesis that this dissertation strived to support is that *real time scheduling algorithms have schedulable bounds equal to the capacity of architecture (based on some assumptions) but can be made more efficient by minimize scheduling overheads to increase QoS (Quality of Service) of application, and this efficiency is attained by taking into account task's implicit run time parameters. Moreover, the schedulable bounds equal to capacity of architecture can be achieved by relaxing these assumptions.*

6.1 Summary

In Chapter 2, we presented the RUF scheduling algorithm, which takes into account the difference of worst case allocated time to a task and its actual execution time to maximize execution of non critical tasks. We demonstrated that static assignment of priorities to critical and non critical tasks decrease the performance of overall system. We have proposed that priorities assigned to tasks based on static parameters of tasks implicit or explicit can lead to decrease in QoS of the application. In Chapter 2, we have proposed to take into account runtime parameters of a task which is the difference in its C_i and AET_i before assigning priorities to tasks and selecting them to execute. We have proposed to assign priorities dynamically, this assignement is not based on absolute deadline or laxity of a task but on run time parameters which can not be predicted offline, to augment the efficiency of scheduler. We have compared it with MUF algorithm, as only MUF provides guarantees to critical tasks in overload transient situations, and we have illustrated that RUF outperforms MUF algorithm in terms of number of non critical tasks executed over a hyper period.

In Chapter 3, we presented a technique to minimize the scheduling overhead in a mono-processor context. There are two types of costs related to scheduler. Firstly the cost related to the complexity of the scheduling algorithm (time it takes to decide to select a task(s)). Second is related to decision of the scheduler which can force a task to be preempted (or migrated in case of multiprocessor system). EDF and RM scheduling algorithms has very low runtime complexity of algorithm but their decisions still have significant impact on overall performance of the system. In Chapter 3, we have proposed a technique to reduce the significant scheduling overhead, which is preemption of a task. We extended the work of Baruah to minimize the preemptions of a task, we have considered the two implicit parameters of a task which are its laxity and release time. Laxity of a task gives a flexibility to postpone the normal decision of scheduling algorithm without compromising on feasibility which helps minimizing preemptions of tasks, while the release time of task enable us to consider a scenario different from the worst case scenario helping to reduce the number of preemptions of a task to greater extent. We have proposed two variants of this technique, which are static and dynamic, to minimize preemptions. Both of these techniques also allow dynamic creation and deletion of tasks. Reducing number of preemptions not only increase practically achievable processor utilization, but it decreases the energy consumption of the system also. The aim of DVFS techniques is to decrease energy consumption by lowering operating frequency of the processor but the side effect is an augmentation of number of preemptions due to larger execution times of tasks. We proposed a novel DVFS management technique which decreases the frequency of processor only at those instants, when switching frequency from one level to another level limits its impact on number of preemptions and decreases the number of switching points.

In Chapter 4, we have provided a novel global scheduling algorithm ASEDZL, which is not based on the notion of fairness or fluid scheduling model. It is based on the EDF approach, similar to mono-processor EDF. We have exploited the reasons/factors for which an optimal mono-processor scheduling algorithm performs poorly if it is used to schedule tasks on multiprocessor system. We worked on these factors to bring optimality in case of multiprocessor scheduling as well. We noticed that the weight of a task is always less than or equal to the offloading factor of another task in case of mono-processor systems. Conversely, if the weight of a task is greater than the offloading factor of another task in case of multiprocessor system, it causes a task to miss its deadline even at low processor utilization. This antagonism can be eliminated by considering a modified version of EDF i.e., EDZL. We also observed that in case of mono-processor EDF scheduler, subsequent tasks have always sufficient execution time until next release instant and this property is true in case of multiprocessor EDF scheduler too. But unlike mono-processor case, multiprocessor g-EDF scheduler is not capable to execute subsequent tasks until next release instant. To provide the same behavior in multiprocessor scheduling algorithm, we have introduced few modification which are defining virtual deadlines and local execution requirements for subsequent tasks, to ensure execution of tasks until next release instant. Another intrinsic characteristic that we observed in EDF scheduler (for mono-processor) is that when a task is selected to execute with its zero laxity, it is the only ready task and it finishes its execution at end of hyper period ($U(\tau) = 1$, tasks are synchronous and release time of a task is at its deadline). The same appears in case of multiprocessor scheduling as well when global scheduler does not let any task to go in negative laxity and ensures execution of subsequent tasks until next release instant. Our proposed algorithm ASEDZL ensures both of these conditions to be fulfilled which makes it optimal for scheduling of tasks on multiprocessor architecture. g-EDF has already been shown the most cost effective (in terms of minimum number of preemptions and migrations of tasks) but with low processor utilization. As our proposed algorithms is just a simple extension to g-EDF which illustrate that ASEDZL is the most cost effective global scheduling algorithm with schedulable bound equal to capacity of architecture. To support our argument, we have compared our algorithm with LLREF and demonstrated that algorithm ASEDZL is much more efficient than Pfair and LLREF.

In Chapter 5, we presented two approaches for hybrid scheduling of tasks. Hybrid scheduling gives better results on distributed shared memory architecture. We have used the supertasking approach to schedule tasks on multiprocessor architecture. We started with the work of Moir et al. which does not ensure deadline guarantees for local tasks. We established a relation between weight of a supertask and those of a local tasks if provided ensures deadline guarantees for all tasks. The work of Moir et al. was based on using Pfair scheduling algorithm which is an optimal scheduling algorithm with very high runtime complexity, due to numerous scheduling events, higher preemptions and migrations of tasks. Therefore, we replaced Pfair with our proposed algorithm ASEDZL, and we presented a technique for scheduling of supertask and global task in such a way that it ensures deadline guarantees for all tasks. For this, we have assigned dynamic parameters to supertask instead of static parameter as it was in earlier version. This introduction of dynamic parameters ensures that supertask is always guaranteed allocation of sufficient amount of time by global scheduler which is greater than or equal to minimum demand of local tasks to meet their deadlines. We have proved the optimality of our algorithms, and we have compared it with other algorithms of the domain by examples to illustrate its improvement over other algorithms.

Last but not the least, we have also devised a technique for scheduling self-adaptive applications on future architectures in context of ÆTHER project [1]. Future architectures

[1] ÆTHER is IST-FET (Information Society Technology-Future and Emerging Technologies) European project with main objective to study novel self-adaptive computing technologies for future embedded and pervasive applications.
web: http://www.aether-ist.org/

are assumed to be intelligent enough to adapt/self-configure themselves according to varying demands of applications. To provide deadline guarantees to self adaptive application on such future architecture, we have proposed to calculate minimum demand of resources for each task statically, but tasks are allocated resources more than its minimum demand at runtime depending upon degree of parallelism at that specific level of task and on number of available resources. At run time, allocation of resources higher than minimum demand of a task for fraction of task reduces its minimum demand for remaining part of task which could allow other tasks to be admitted in the system. Architecture consider in this system is futuristic. Moreover, it deals with soft real time tasks, where approach is based on best effort techniques which makes it a bit different from rest of the thesis work, that's why we have placed this work in Annex A instead of another chapter. In Annex B, we have presented an ongoing work. In this work, we have proposed to break a multiprocessor scheduling problem into multiple mono-processor scheduling problems and proposed a solution which offers optimal schedulable bound.

6.2 Future Work

We have presented RUF scheduling algorithm to improve QoS on mono-processor architecture. We are aiming to extend this approach on multiprocessor architecture where critical tasks guaranteed to meet their deadlines while execution of non critical is maximized in overload situations. We have proved the optimality of ASEDZL on multiprocessor architecture but we aiming to take into account runtime parameters of tasks i.e., $C_i > AET_i$, to minimize power consumption of the system without compromising on deadline guarantees. In a very near future, we shall have a simulator developed in context of PHERMA(ANR) project which will help us to implement ASEDZL on it and to work on power optimization aspects. Until now, we have simulated our mono-processor scheduling algorithms and global scheduling algorithms on CoFluent tool. This tool does not support implementation of hierarchical scheduling, that's why we did not have simulation results on hierarchical scheduling algorithm. We shall implement our proposed hierarchical scheduling algorithm on this newly developed to tool to compare simulation results with existing approaches.

Conclusions

Les algorithmes d'ordonnancement temps réel de type EDF, LLF ou MUF sont optimaux dans le cas monoprocesseur et les algorithmes Pfair ou LLREF le sont dans le cas multiprocesseur. Cependant les algorithmes Pfair et LLREF atteignent cette optimalité au prix d'une importante complexité liée en particulier au nombre important d'événements d'ordonnancement à traiter. Dans cette thèse nous nous sommes attachés à montrer qu'il est possible de proposer des algorithmes d'ordonnancement optimaux (sous certaines hypothèses), c'est-à-dire ayant une borne d'ordonnançabilité égale à la capacité de traitement de l'architecture cible, et ce avec une efficacité accrue du fait d'une réduction du coût de gestion induit par l'ordonnanceur. Ce gain en efficacité est obtenu en utilisant au mieux des paramètres implicites des tâches, évalués à l'exécution. Il en découle en particulier une augmentation de la Qualité de Service (QoS) globale de l'application. Par ailleurs, en relâchant les hypothèses faites sur les algorithmes, la borne maximum d'ordonnançabilité égale à la capacité de traitement totale de l'architecture peut être atteinte dans certains cas.

6.1 Résumé des travaux

Dans le chapitre 2, nous avons présenté l'algorithme d'ordonnancement RUF qui prend en compte les différences entre le temps d'exécution pire cas des tâches et leurs temps d'exécution effectifs avec pour objectif de maximiser l'exécution des tâches non critiques tout en garantissant les échéances des tâches critiques. Nous avons montré qu'une affectation de priorités statiques aux tâches critiques et non-critiques réduit les performances du système global. Aussi nous avons proposé que les priorités des tâches soient calculées à partir de la connaissance de leurs paramètres explicites et implicites. En particulier, le calcul de la priorité d'une tâche, puis sa sélection en vue d'exécution, tient compte de la différence Ci - AETi des tâches qui ont terminé leur exécution. Cette affectation de priorité est réalisée de manière dynamique suivant un calcul qui ne tient pas compte des échéances absolues des tâches ou de leur laxité mais des paramètres implicites évalués à l'exécution. Nous avons comparé cet algorithme avec l'algorithme MUF qui fournit des garanties sur les échéances des tâches critiques dans les phases de surcharge. Nous avons illustré que l'algorithme RUF est plus performant que MUF vis-à-vis du nombre de tâches non-critiques exécutées suivant leurs échéances pendant une hyper-période.

Dans le chapitre 3, nous avons proposé une technique pour réduire le coût induit par la gestion de l'ordonnancement dans un contexte mono-processeur. Il existe en particulier deux types de coûts, l'un est relatif à la complexité algorithmique de l'ordonnanceur (celle relative à la procédure de sélection de la tâche à exécuter), l'autre est la conséquence des décisions de l'ordonnanceur de préempter une tâche. Les algorithmes RM et EDF possèdent des complexités algorithmiques réduites mais les décisions de préemptions ont un impact significatif sur les performances globales. Dans le chapitre trois nous avons proposé une technique pour réduire significativement ces coûts liés aux préemptions. Nous avons ainsi étendu des travaux de S.K. Baruah en considérant deux paramètres implicites des tâches : leur laxité et leur dates de requête. La laxité d'une tâche apporte une flexibilité pour retarder la décision normale de préemption de l'algorithme d'ordonnancement et ce sans compromettre la faisabilité de l'ordonnancement. Ceci permet ainsi de réduire le nombre de préemptions. Par ailleurs, contrairement à une analyse statique hors-ligne, considérer les instants de requêtes des tâches permet pendant l'exécution de ne prendre en compte que les tâches réellement prêtes et non pas le scénario pire cas qui consiste à considérer que toutes les tâches peuvent être prêtes à chaque événement d'ordonnancement. On peut ainsi réduire encore les préemptions suivant cette approche dynamique par rapport au calcul effectué hors-ligne. On peut également remarquer que cette technique dynamique autorise la création dynamique de tâche. La réduction du nombre de préemptions permet d'augmenter l'utilisation effective du processeur pour des traitements utiles et contribue également à réduire la consommation d'énergie. L'objectif des techniques DVFS est de minimiser la consommation d'énergie en diminuant la fréquence et la tension d'alimentation du processeur mais l'effet de bord est une augmentation du nombre de préemptions liée à l'allongement des temps d'exécution des tâches par rapport à leurs périodes. Nous avons proposé une nouvelle technique de gestion de DVFS qui réduit la fréquence du processeur uniquement aux instants où l'impact sur le nombre de préemptions est limité, tout en réduisant le nombre de changements de fréquence.

Dans le chapitre 4 nous avons développé un nouvel algorithme d'ordonnancement global ASEDZL qui n'est pas basé sur la notion d'équité ou sur un modèle d'ordonnancement fluide. Il est basé sur l'approche EDF et similaire au cas monoprocesseur. Pour ce faire, nous avons exploité les raisons et les facteurs qui font qu'un algorithme optimal en monoprocesseur peut donner de faibles performances pour ordonnancer des tâches dans un contexte multiprocesseur. Nous avons travaillé sur ces facteurs pour obtenir un ordonnancement optimal. En particulier, dans le cas monoprocesseur, nous avons mis en évidence que le facteur d'utilisation du processeur par une tâche est toujours inférieur ou égal au facteur de non-utilisation du processeur par une autre tâche. Inversement, si dans le cas multiprocesseur le facteur d'utilisation d'une tâche est plus grand que le facteur de non-utilisation d'une autre tâche alors cela entraîne qu'une tâche ne vérifiera pas son échéance, même si le taux d'utilisation des processeurs est faible. Cet antagonisme peut être éliminé en considérant l'approche EDZL qui est une version modifiée d'EDF. Nous avons également remarqué que dans un ordonnancement EDF monoprocesseur, les tâches à venir disposeront toujours d'un temps d'exécution suffisant et ce jusqu'au prochain instant de requête d'une tâche. Cette propriété reste vraie dans le cas EDF multiprocesseur. Cependant à l'inverse du cas monoprocesseur l'ordonnancement EDF multiprocesseur n'est pas capable d'exécuter pour un temps suffisant les tâches à venir jusqu'à la prochaine requête d'une tâche. Pour obtenir un comportement équivalent de l'ordonnanceur mono et multiprocesseur nous avons introduit plusieurs modifications dont des échéances virtuelles et des besoins locaux d'exécution pour les tâches à venir de manière à assurer un temps d'exécution suffisant pour ces tâches. Une autre particularité spécifique observée dans l'ordonnanceur EDF monoprocesseur est que lorsqu'une tâche avec une laxité nulle est sélectionnée pour exécution, cette tâche est la seule à être prête et elle finira son exécution à la fin de l'hyperpériode (suivant les hypothèses : $U(\tau) = 1$, les tâches sont synchrones et ont leur requêtes sur échéances). La même situation apparaît en multiprocesseur si l'ordonnanceur global vérifie d'une part que toutes les laxités des tâches ne sont jamais négatives et d'autre part que toutes les tâches à venir bénéficient d'un temps d'exécution jusqu'à la prochaine requête d'une tâche. Avec l'algorithme ASEDZL ces conditions sont vérifiées ce qui le rend optimal pour ordonnancer des tâches sur une architecture multiprocesseur. L'algorithme EDF global (g-EDF) est montré comme le plus efficace en termes de nombre de préemptions et de migrations de tâches mais au prix d'une faible utilisation possible des processeurs. Comme l'algorithme ASEDZL est une extension de g-EDF, on peut affirmer qu'il constitue l'algorithme d'ordonnancement global le plus efficace avec une borne d'ordonnançabilité égale à la capacité de traitement de l'architecture. Pour illustrer cet argument nous avons comparé notre algorithme à LLREF et ainsi montré qu'ASEDZL est plus efficace que Pfair et LLREF. Nous avons présenté dans le chapitre 5 deux approches d'ordonnancement hybride de tâches. L'ordonnancement hybride de tâches donne a priori de meilleurs résultats sur des architectures à mémoire partagée distribuée par rapport à des algorithmes globaux ou partitionnés. Nous avons utilisé l'approche par "supertâches" proposée par Moir et al. qui toutefois ne garantit pas les échéances des tâches locales à une supertâche. Nous avons établi une relation entre les taux d'utilisation des tâches locales et celui de la supertâche associée de manière à garantir les échéances des tâches locales. Les travaux de Moir et al. sont basés sur l'utilisation de l'ordonnancement optimal Pfair qui possède une importante complexité de calcul à l'exécution. Aussi, nous avons substitué Pfair par ASEDZL et nous avons présenté une technique d'ordonnancement des supertâches et des tâches globales de telle sorte que toutes les échéances des tâches soient garanties. Pour arriver à ce résultat nous avons associé des paramètres dynamiques aux supertâches au lieu de paramètres statiques comme effectué dans les travaux précédents. Ceci assure que les supertâches bénéficient de la part de l'ordonnanceur global d'un temps d'exécution suffisant supérieur ou égal à la demande minimum des tâches locales et ce afin qu'elles vérifient leurs échéances. Nous avons montré l'optimalité de notre algorithme et l'avons comparé avec d'autres algorithmes du domaine pour illustrer les améliorations obtenues.

Enfin, nous avons proposé dans le cadre du projet européen THER1 une technique d'ordonnancement auto-adaptatif d'applications pour de futures architectures. On peut penser que les architectures du futures seront " intelligentes " pour pouvoir s'adapter/s'auto-configurer en fonction de la demande variable en traitement des applications. Pour obtenir des garanties sur les échéances de ces applications auto-adaptatives nous avons proposé de calculer de manière statique la demande minimum en ressources pour chaque tâche puis, à l'exécution d'allouer à chaque tâche plus de ressources que cette demande minimum en fonction du degré de parallélisme intrinsèque à la tâche observé et du nombre de ressources disponibles. L'allocation à l'exécution d'un plus grand nombre de ressources que la demande minimum permet de réduire la demande minimum pour la partie restante à exécuter de la tâche avec pour retombée de pouvoir ensuite admettre dans le système de nouvelles tâches. L'architecture considérée dans cette étude est quelque peu futuriste et par ailleurs l'approche proposée ne concerne que des tâches temps réel souple ce qui est un peu différent des autres travaux réalisés dans cette thèse. Ceci explique que ce travail est décrit en Annexe A et non pas dans un chapitre à part entière. Dans l'Annexe B nous avons également présenté un travail en cours qui vise à transformer un problème d'ordonnancement multiprocesseur en plusieurs problèmes d'ordonnancement monoprocesseur et sur cette base, nous avons proposé une solution pour obtenir une borne d'ordonnançabilité optimale.

6.2 Travaux futurs

L'algorithme d'ordonnancement RUF améliore la Qualité de Service pour des architectures monoprocesseurs. Nous proposons d'étendre cette approche au cas multiprocesseur avec la garantie que les tâches critiques vérifient leurs échéances et les exécutions des tâches non-critiques sont maximisées pendant les phases de surcharge. L'optimalité d'ASEDZL permet de considérer des paramètres dynamiques des tâches (par exemple $C_i > AET_i$) dans le but de minimiser la consommation d'énergie du système sans remettre en cause la garantie offerte sur le respect des échéances des tâches. Dans un futur proche nous utiliserons le simulateur développé dans le cadre du projet PHERMA (ANR) afin d'implémenter l'algorithme ASEDZL et travailler sur ces aspects basse consommation. Jusqu'à présent nous avons utilisé l'outil CoFluent pour simuler les algorithmes d'ordonnancement monoprocesseur ou globaux. Cet outil supporte difficilement l'implémentation d'algorithmes hiérarchiques aussi nous envisageons d'utiliser ce nouveau simulateur développé dans PHERMA pour les réaliser et ainsi comparer les résultats aux approches existantes.

Appendix A

ÆTHER: Self adaptive Resource Management Middleware for Selfoptimizing Resources

A.1 Introduction

The development of reconfigurable devices that could make themselves domain-specialized at run time is becoming more and more common. Future reconfigurable architecture will have these computing devices as basic blocks, and reconfigurable architecture could make assemblies of these devices, on the fly, to execute concurrent applications. The migration from completely generic lookup tables and highly connected routing fabrics to self adaptive specialized coarse-grain reconfigurable devices and very structured communication resources presents designers with the problem of how to best customize the system based upon anticipated usage. Then there is a need of not only exploiting parallelism from applications at micro-thread level, dynamically, but system also starves for a dynamic and self adaptive middleware to schedule these micro-threads on thousands of such computing devices. In this chapter we focuses at the problem of dynamic allocation and scheduling of resources of numbers of applications on such architecture.

ÆTHER is IST-FET (Information Society Technology-Future and Emerging Technologies) European project with main objective to study novel self-adaptive computing technologies for future embedded and pervasive applications. In order to provide high performance computation power to serve the increasing need of large applications, people strive to improve a single machine's capacity or construct a distributed system composed of a scalable set of machines. Compared to the former, where the improvement is mainly up to the hardware technology development, the construction of distributed systems for resource collaboration is more complex. Some of well-known existing distributed systems composed of heterogeneous resources are Condor[47], NetSolve [21], Nimrod [73], Globus and the Grid [40] computation environment. Sabin et al. [70] propose a centralized metasheduler which uses backfill to schedule parallel jobs in multiple heterogeneous sites. Similarly, Arora et al. [8] present a completely decentralized, dynamic and sender-initiated scheduling and load balancing algorithm for the Grid environment. All these approaches don't deal with application model where concurrent threads are created and managed at run. These methods do not target future architecture where each resource of a processor has capability of self optimizing and interconnects with other resources to form assemblies to execute concurrent application.

ÆTHER system is hierarchical both at function and architecture level. At function level, application is written which self adapts itself to well suit with the application objective and to cope with dynamic changes happening in environment. Applications are quite dynamic in nature where concurrent threads are instantiated dynamically according to number of resources of system. Architecture of the system is not traditional as it is not single unit of computation. It is a network of given number of SANEs. A SANE is self adaptive networked entity which can self optimize itself.

In this chapter, we propose an algorithm for scheduling of hard and soft real-time tasks in an architecture which is composed of multiple processing units. These computing units are self adaptive and have the capability to optimize according to application needs. Application helps create/instantiate concurrent threads at run time. The primary goal of the proposed algorithm is to maximize the schedulability of soft tasks without jeopardizing the schedulability of tasks. The algorithm has the inherent feature of degrading/upgrading QoS, by dynamic managing concurrency of tasks by allocating more resources to most appropriate task i.e., (hard real time tasks are not always preferred over soft real time tasks). This algorithm helps to maximize execution of concurrent execution of tasks and distribute resources to thousands of concurrent threads of a task where objective is to ensure timeline guarantees of tasks.

A.1.1 Definition Of The Problem

We have n tasks $\tau = T_1, T_2, ..., T_n$ and R_x parallel resources (SANEs). These parallel resources self adapt themselves according to task executing on it. These resources form assemblies and configure/self-adapt [61] connecting fabric between SANE elements to execute tasks while tasks have the capability of dynamically managing (creating threads at runtime) concurrency in it depending upon the availability of resources. System resources are limited so that demands of all tasks can not be satisfied simultaneously. Execution time of task monotonically decreases with each resource allocated to it until the maximum level of parallelism in task. For some tasks where concurrent parts of tasks are dependent, execution time of a task is monotonically decreasing until a point (called threshold) and it starts increasing monotonically after this point for each allocated resource. Task T is

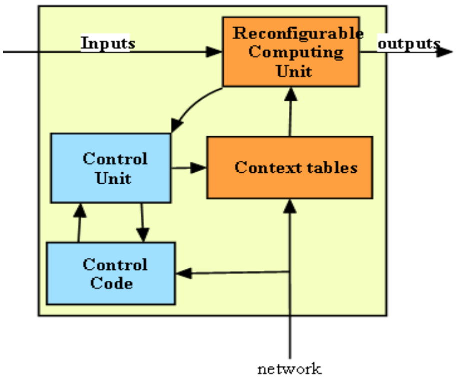


FIGURE A.1: Self Adaptive Networked Entity (SANE)

represented as sequence of concurrent execution of μthreads (μT). Concurrency level at each sequence of task may be different. We are interested in scheduling these tasks on Rx parallel resources such that maximum tasks could respect their deadline constraints and utilization of resources could be maximized. In this chapter,we define ÆTHER architecture. we detailed our proposed approach and explain that how tasks are provided real time guarantees by calculating minimum resource demand of each task and how resources are allocated dynamically to increase Quality of Service (QoS).

A.2 ÆTHER Architectural Model

In the ÆTHER project, SANE [61](Self Adaptive Networked Entity) is introduced as basic computing entity aims to be networked with other entities of the same type to form complete systems. Each of these entities is meant to be self-adaptive, which implies that they can change their own behavior to react to changes in their environment or to respect some given constraints. As shown in Fig:A.1, the controller changes the state of the SANE hardware implementation by changing some parameters of the currently loaded task as well as changing the task to another implementation of the same task or a completely different task. The computing tasks are loaded in the computing engine. They can be described as bit-streams if the computing engine is viewed as an FPGA fabric or as binary files for a soft-processor. The existence of the monitoring process associated with an adaptation controller provides the SANE with the self-adaptation ability. The latest part of the SANE is the communication interface. It is dedicated to collaboration among the SANE hardware elements that compose the architecture. The collaboration process is done through a publish/discover mechanism that allows a SANE hardware element to publish its own abilities and parameters and to discover the computing environment formed by the other SANE hardware elements in its immediate local neighborhood. This mechanism enables the SANE hardware elements to exchange their tasks or just to clone their states to other SANE hardware elements. The SANE processor (Fig:A.2) is a runtime

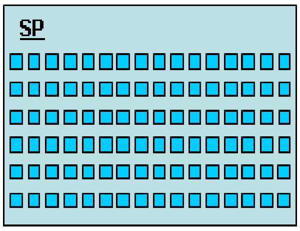
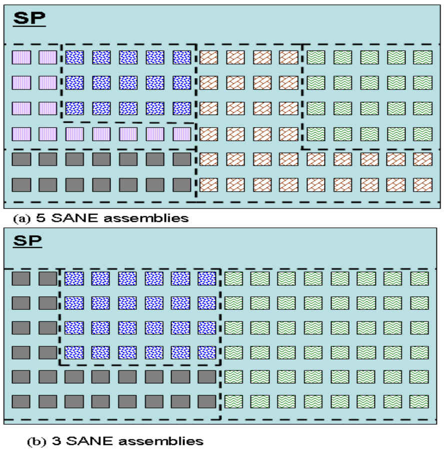


FIGURE A.2: SANE processor composed of SANE Elements



(a) 5 SANE assemblies

(b) 3 SANE assemblies

FIGURE A.3: SANE Assemblies

reconfigurable architecture composed of thousands of SANE elements. These elements are interconnected dynamically and self-adaptively to execute concurrent applications. SANE processor is a multi-core processor with SANE elements as its processing cores. SANE processor has more flexibility than simple multi-core processor due to its capability of not only reconfiguring its cores (SANEs) but also restructuring the interconnection between them. SANE elements (equal to resources allocated to task T_i dynamically) are combined to form a SANE assembly to execute task T_i. As number of concurrent tasks at run time varies dynamically so assemblies formed at run time are different as well. Moreover, resources assigned to each task vary. Number of SANE assemblies in one SANE processor changes at run time (Fig:A.3(a),(b)).

A.3 Application Model

There are two types of parallelism to be exploited: task parallelism and data parallelism. In ÆTHER project, parallelism is exploited through a coordination language. S-Net [9], used in ÆTHER project, is a coordination language that orchestrates asynchronous components that communicate with each other and their execution environment solely via typed streams. The application program units are presented in an appropriate fully-fledged programming language, such as C, Java, etc., while the aspects of communication,

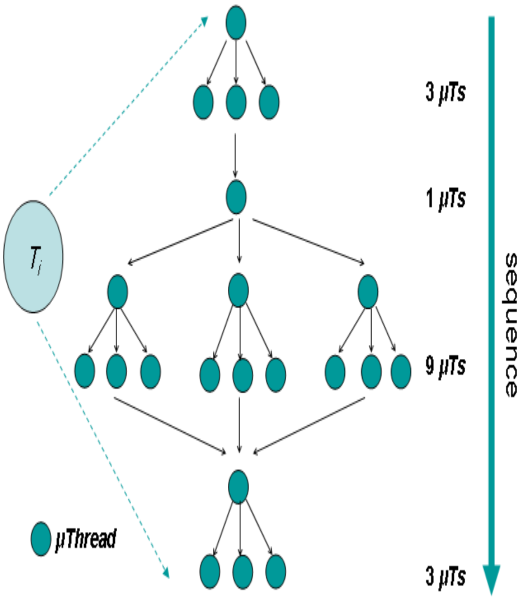


FIGURE A.4: Representation of a task

concurrency and synchronization (referred to by the term coordination) are captured by a separate, coordination, language. S-Net program is compiled down to μthreadedC (μTC) [18], which is used as user programming language. μTC is a rather profound but simple extension to the C language, allowing it to capture massive thread-based concurrency. μTC is capable of expressing static heterogeneous concurrency and dynamic, homogeneous concurrency. Only a small number of constructs are added to C, along with the semantics of the synchronizing memory. The constructs map onto low-level operations that provide the concurrency controls in a μthreaded ISA, and allow concurrent programs to be dynamically instanced and preempted, either gracefully or with a prejudice.

A.3.1 Task Structure

A task T_i is represented as a sequence of concurrent execution of μthreads. Each task T_i consists of a finite series of sequences $T_{i,1}, T_{i,2}, T_{i,3}, ..., T_{i,k}$. Each sequence $T_{i,j}$ consists of (max) $N_{i,j}$ concurrent μTs of execution and each μT must run for at most $C_{i,j}$ time units; such value is called the worst-case computation time of μT, $S_{i,j}$ is the slow down factor if there is dependency between two μTs. The number of μTs in any sequence is limited by a number and is known a prior. The μTs are instanced dynamically depending upon the resources availability. In Fig:A.4, we can observe that in a task structure, we have different number of μTs at different levels. These μTs represents the maximum limit but threads created at runtime may have different values bounded by these maximum limits. The resources are allocated to different tasks depending upon its criticalness, its execution time and efficiency of a resource to execute a thread. It is assumed that one resource is capable of executing a μT, but resources can optimize self adaptively to execute more than

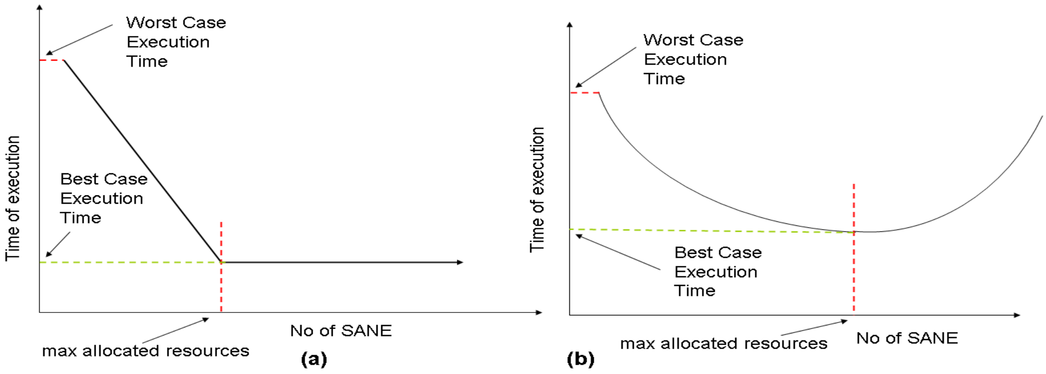


FIGURE A.5: Execution time of task

one μT at a time. In this case middleware will change the task structure dynamically and self adaptively that helps to make better decisions about resource allocation. Initially, execution time of a task is calculated considering that one SANE is capable of executing only one μT at a time.

A.3.1.1 Worst Case Execution Time

The worst case execution time of task is function of allocated resources at run time. Worst case execution time can be calculated by considering only one μT in execution at one time i.e., with no parallelism.

$$C_i = \sum_{j=1}^{k} N_{i,j} * C_{i,j} * S_{i,j}$$

A.3.1.2 Best Case Execution Time

Best case execution time is calculated by assigning resources equal to are less than maximum μTs at that level.

$$C_{i,b} = \sum_{j=1}^{k} S_{i,j} * \left\lceil \frac{N_{i,j}}{R_b} \right\rceil * C_{i,j}$$

The worst case execution time and best case execution of a task depends on number of minimum (Rim,s) and maximum resources that can be allocated to it. If there is no dependency between μTs of one family then there time of execution will be decreased linearly. But there will be a point after which increase in allocation of resources will not further decrease time of execution. It will remain constant (Fig:A.5 (a)). This point is called saturation point. In some cases when there is strong dependency between μTs of a family then time of execution will not be decreased linearly and there will be a threshold point after which time of execution of family of μTs start increasing instead of decreasing if more resources are assigned than its threshold point (Fig:A.5 (b)). During execution of a task, hardware resource (SANE element) self adaptively optimize its power to execute

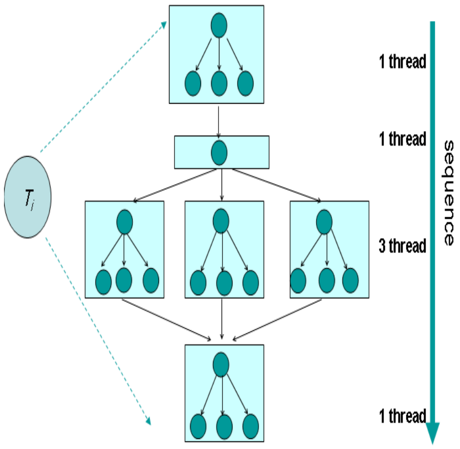


Figure A.6: Varied task structure

more than one μT of a family. In this scenario, task structure will be changed due to positive feedback from hardware.

A.3.2 Self Adaptive Task Structuring

SANE elements, having capabilities of self optimization, require a dynamic and self adaptive middleware to cope with these optimizations of hardware that could distribute resources in an efficient manner. Optimization achieved by hardware for any task can be of different types. SANE element

- could optimize itself to provide dedicated functionality implemented in hardware for a task.
- could self optimize to execute complete or partial family of μTs concurrently.

A.3.2.1 Dedicated SANE for a Task

If a SANE element has dedicated hardware for a task or it has self-optimized itself at runtime to execute all sequences of task, then there will be no more modifications in structure of this specific task.

A.3.2.2 Optimized SANE for Family of μTs of a Task

SANE element may make itself specialized only for parts of task instead of complete task. It may provide better results for a family of μTs. In this scenario, task structure will be changed (Fig:A.7) and there will be a need to recalculate the execution time of tasks with new parameter i.e., number of sequences, number of threads at each level (Fig:A.7) and worst case execution time of modified thread. Resources are allocated to different tasks depending upon their structure and its deadline constraints.

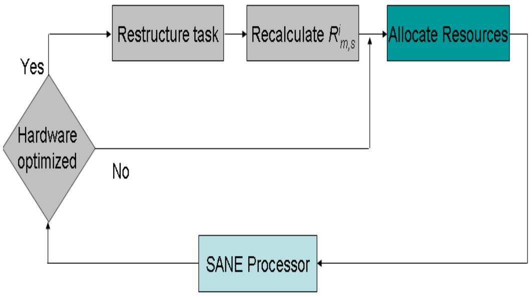


FIGURE A.7: Restructuring of Task self adaptively

A.4 Resource Allocation and Scheduling

Ideal scheduling algorithm is one where each task is assigned resources proportion to its weight during the whole length of its execution. Ideal scheduling algorithm is impractical

- in case of general purpose processors where computational capacity of one processor can't be assigned to different tasks in proportion to their weights.
- in real cases where application does not have μTs equal in number (or more) that corresponding to its weight. The number of μTs that an application can execute in parallel varies during its execution.

In case of SANE processors, resources (SANE Elements) may be assigned proportionally and dynamic concurrent application model help to execute a task in parallel. If minimum number of μTs in any sequence of a task are integer multiple of number of resources corresponding to its weight then this task can be scheduled ideally. But if a task T_i has number of μTs less than Rim,s (resources proportion to its weight) at some level and greater than Rim,s at other levels (which is the case most of time), then this task can't be scheduled ideally. To provide guarantees for this task, reservation of resources more than its weight is needed. It will cause certain resources left unused and there will be wastage of resource utilization.

A.4.1 Wasted Resource Utilization

If schedulability analysis is carried out based on minimum resources $R^i_{m,s}$ then there are chances that these resources may not be fully used during execution of a task. As few sequences of a task may need less resources than $R^i_{m,s}$ (Figure 8) hence certain percentage of resources will not be used during execution of a task.

A.4.1.1 Homogenous Concurrent μthreads

μTC has the capability of controlling the concurrency of μTs dynamically if μTs at that level are homogenous. In this case, if we have more than one homogenous families of μTs

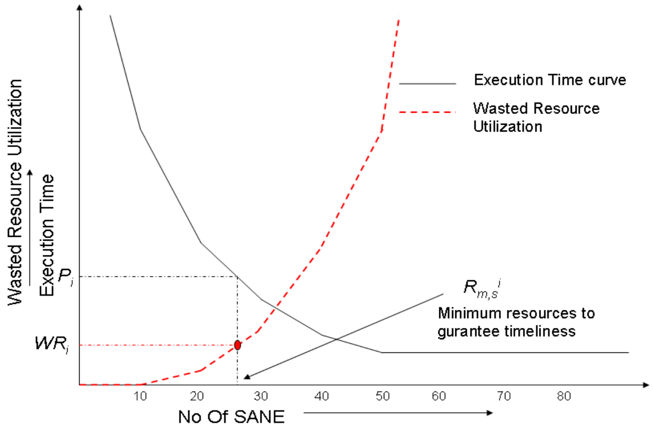


FIGURE A.8: Wasted Resource Utilization

at same level, then these families can be considered as a single family. If two or more than two families are concurrent then number of total μTs at that level $N_{i,p}$ is sum of all those μTs.

$$N_{i,p} = N_{i,n} + N_{i,m}$$

Each μT of these two families will have same worst case execution time. Wasted Resource Utilization by task T_i when it is assigned $R^i_{m,s}$.

$$WR_i = \sum_{p=1}^{k} max(0, (R^i_{m,s} - N_{i,p}) * C_{i,p})$$

where k represents number of sequential families of task T_i.

A.4.1.2 Heterogeneous Concurrent μthreads

If families of μTs at any level are heterogeneous i.e., worst-case execution time of μT in one family is different from that of other, then calculation of wasted resource utilization may have different value than calculated in above equation and is calculated in the following way.

$$WR_i = \sum_{p=1}^{k} max(0, (R^i_{m,s} - N_{i,p}) * max(C^1_{i,p}, C^2_{i,p}, ..., C^l_{i,p},))$$

Where $(C^1_{i,p}, C^2_{i,p}, ..., C^l_{i,p},)$ are worst case execution times of μTs of concurrent heterogeneous families at level p and l represents total number of families at that level. If Wasted Resource Utilization (WR_i) by task T_i is zero when it is assigned then:

$$R^i_{m,s} = \left\lceil \frac{C_i}{P_i} \right\rceil$$

otherwise

$$R^i_{m,s} > \left\lceil \frac{C_i}{P_i} \right\rceil$$

With the new static value of $R^i_{m,s}$ task will respect its real time constraints but wastage of resources will be increased.

A.5 Minimum Resources for Each Task

A task is represented as a sequence of concurrent executions of μthreads. At each level a task has different number of μTs and worst case execution time of a μT at one level may be different from that of a μT at other level. Minimum number of resources ($R^i_{m,s}$) are calculated that should be allocated to a task to provide timeline guarantees. If worst case execution time calculated for a task T_i is higher than its deadline P_i, then $R^i_{m,s}$ (calculation before release of task instant) that a task should be allocated can be calculated by an iterative process. We have a function associated with each level of a task. With the help of this function we can calculate the worst case and best case execution time of this family of μTs. Calculation of minimum resources that does not correspond exactly to its weight (higher than its weight) is calculated as follows: (WR represents wastage of resources (section 5)) A task T_i can be allocated more than $R^i_{m,s}$ at run time. If a task T_i use more than $R^i_{m,s}$ for certain duration, then minimum resource for rest of task may have a smaller value and is calculated dynamically (dynamic minimum demand $R^i_{m,d}$).

Algorithm 9 Calculation of Minimum Resources

```
$R^i_{m,s} = \left\lceil \frac{C_i}{P_i} \right\rceil$;
$WR_i = \sum_{k=1}^{n} max\left[0, \left(R^i_{m,s} - N_{i,k}\right) \times C_{i,k}\right]$;
if $\left[\frac{C_i}{R^i_{m,s}} + WR_i\right] > P_i$ then
    $R^i_{m,s} = \left\lceil \frac{C_i}{P_i - WR_i} \right\rceil$;
    go to step 3;
else
    return $R^i_{m,s}$;
end if
```

A.5.1 Task Preemption

If a task T_i has used more than Rim,s, then this task T_i can be preempted as well. Time PT_i for which a task can be preempted by low priority task depends upon the time duration for which this task has used resources more than $R^i_{m,s}$.

$$PT_i = P_i - \left[t_c + \frac{C_i^{rem}}{R^i_{m,s}} + WR_i\right]$$

where t_c represents the current time. A task can be preempted for a longer time than calculated in above equation, if it could be allocated more than $R^i_{m,s}$. It is possible in two situations:

1. $\sum_{i=1}^{n} R^i_{m,s} < R_x$
2. When a set of tasks have different values of $R^i_{m,s}$ and $R^i_{m,d}$.

In both of these cases task T_i can be preempted for a longer duration. Time for which this task can be pre-empted is calculated as follows:

$$PT_i = P_i - \left[t_c + min\left(\frac{C_i^{rem}}{R^i_{m,s}}, \frac{C_i^{rem}}{R^i_{m,s} + \sum_{j=1}^{n} R^i_{m,s} - R^i_{m,d}} \right) + WR \right]$$

A.6 Algorithm

Real-time applications are classified into two major categories of hard and soft real-time tasks (HRT and SRT tasks respectively). Hard real-time tasks have critical deadlines that are to be met in all working scenarios to avoid catastrophic consequences. In contrast, soft real-time tasks (e.g., multimedia tasks) are those whose deadlines are less critical such that missing the deadlines occasionally has minimal effect on the performance of the system. Tasks are allocated resources depending upon its value of $R^i_{m,s}$. The calculation of minimum resources for a task changes at run time, if it was allocated more than $R^i_{m,s}$ during its execution. If there is no such HRT task, where $R^i_{m,s}$ and $R^i_{m,d}$ have different

Algorithm 10 Dynamic Allocation of Resources

```
Whenever a new family starts
if ($R_F \geq N_{i,j}$) then
    allocate $N_{i,j}$                          {$R_F$ = free resources}
    return;
else if $R_F \geq R^i_{m,s}$ then
    allocate $R_F$
    return;
else if $R_F < R^i_{m,s}$ then
    for $j = 1$ to $n$ do
        if $R^j_{allocated}(HRT) > R^i_{m,s}$ then
            liberate ($R^j_{allocated}(HRT) - R^i_{m,s}$)      {$R^j_{allocated}$ = allocated resources to $T_j$}
        else if $R_F \geq R^i_{m,s}$ then
            break;
        end if
    end for
else if $R_F \geq R^i_{m,s}$ then
    allocate $R_F$
    return;
else if lower priority task SRT is running then
    preempt it
else
    allocate $R_F$
    return;
end if
if a family finishes its execution (or is preempted) then
    recalculate $R^i_{m,s}$
end if
```

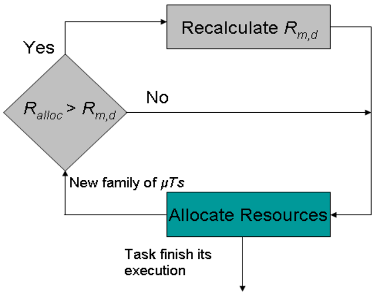


FIGURE A.9: Recalculation of Rm

values then SRT task can't preempt HRT task. In this case low priority SRT task can be preempted only by higher priority SRT task.

A.7 Schedulability

To get the system's behavior deterministic for HRT tasks, we must be sure that at any time minimum demand of resources for all HRT task does not exceed number of resources in architecture. Schedulability analysis for HRT tasks

$$\sum_{i=1}^{k} R_{m,s}^{i} \leq R_x \tag{A.1}$$

If the sum of minimum resources is less than R_x then extra resources can be used to schedule SRT tasks. The allocation/reservation of $R_{m,s}^{i}$ to task T_i introduce wasted slots of resources, these wasted slots can be exploited to schedule SRT tasks. A necessary condition for scheduling of SRT tasks: Where k represents number of HRT tasks and there are m SRT tasks in the system.

$$\sum_{j=1}^{m} \frac{C_j}{P_j} \leq \sum_{i=1}^{k} \frac{WR_i}{P_i} + \left\lfloor \frac{R_x - \sum_{i=1}^{k} R_{m,s}^{i}}{R_x} \right\rfloor \tag{A.2}$$

A.8 Conclusions

In this chapter we have presented an approach for scheduling of HRT and SRT tasks where each task requires more than one resource to finish its execution. We have provided a model that allocates resources to tasks dynamically, that redefines its demand of minimum resources self-adaptively. This model restructures the task as well if SANE hardware optimizes itself. It provides a means of efficiently exploiting unprecedented computational power of SANE processor.

A.9 Acknowledgment

This work was partially supported by the European Commission under Project ÆTHER No. FP6-2004-IST-4-027611, and by French Ministry of Higher Education and Research under contract No. 84-2005. http://www.aether-ist.org.

Appendix B

Dynamic Scheduling of Global and Local Tasks in Their Reserved Slots

In this section, we present a hierarchical scheduling algorithm, which is not based on the idea of supertasking or group based scheduling. The basic idea is to break the problem of multiprocessor scheduling into multiple mono-processor scheduling at different levels of hierarchy. Andersson, Baruah, and Jonsson [6] provided a bound on the utilization of platform capacity, which states that, for a periodic task set with implicit deadlines, the utilization guarantee for EDF or any other static-priority multiprocessor scheduling algorithm - partitioned or global - can not be higher than (M + 1)/2 for an M-processor platform.

This utilization bound can be treated as a sufficient condition for schedulability of a given task set. However, when a task set is partitioned under this bound, the platform is under-utilized with a significant margin. We propose to exploit this under-utilization of each processor by grouping processors in such a way that sum of under-utilization of processors of group is greater than or equal to 1 and less than 2. Under-utilization on one processor is defined as:

$$1 - \sum_{j=1}^{ni} \frac{C_j}{P_j}$$

the i^{th} processor group is represented by Π_g^i. If sum of under-utilization of tasks is greater than 1, then at least one (maximum two) processors of the group as shared with other groups. For each processor, we have a mono-processor EDF based scheduler. Global tasks are also grouped in the same fashion. Total utilization of global tasks must be greater than or equal to 1 and less than or equal to 2. If total utilization is greater than 1, then one or two tasks are shared with other groups.

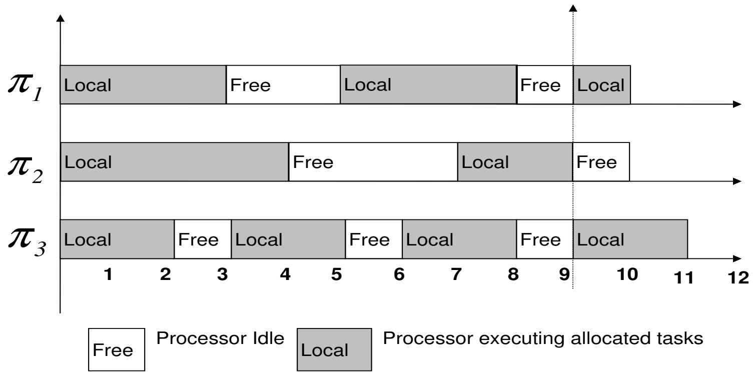


FIGURE B.1: Under-utilization distribution at runtime

B.1 Algorithm Description

In a post-partitioned scenario with independent periodic task set, every processor is treated as an isolated mono-processor which is executing tasks under a mono-processor scheduling policy. The sum of utilization of tasks partitioned on a processor is less than or equal to 1. If this sum is less than one, then there will be idle processor time windows in schedule. Fig.(B.1) illustrate that the idle processor time windows, caused by the under-utilization, appear randomly during runtime. Since random appearing of idle time windows can not be utilized by a periodic task set, therefore, we need to force these idle time windows to be periodic. To achieve this objective, we add a dummy task on every processor. Let's call this dummy task as T_d^i on processor Π_i. We propose to schedule local and global tasks between two release instants. Therefore, we enforce appearing of dummy tasks between any two release instants on all those processors, which has positive under-utilization. Dummy tasks on processors of same group appear on different time windows between two release instants. If sum of under-utilization of processor of a group is greater than 1, then a part of dummy task on one processor of the group is overlapped with other dummy task on other processor of the same group. This overlapped time of dummy task on one processor is assigned to second group.

B.1.1 Local Scheduler

A slot is always reserved (at fix location)for dummy task. Scheduler selects the highest priority local task to execute, when slot reserved for dummy task has either finished or it has not appeared yet. Length of the this slot reserved for dummy task is different between different release instants (Fig:B.2), and depends upon the value of *interval_length*. Length

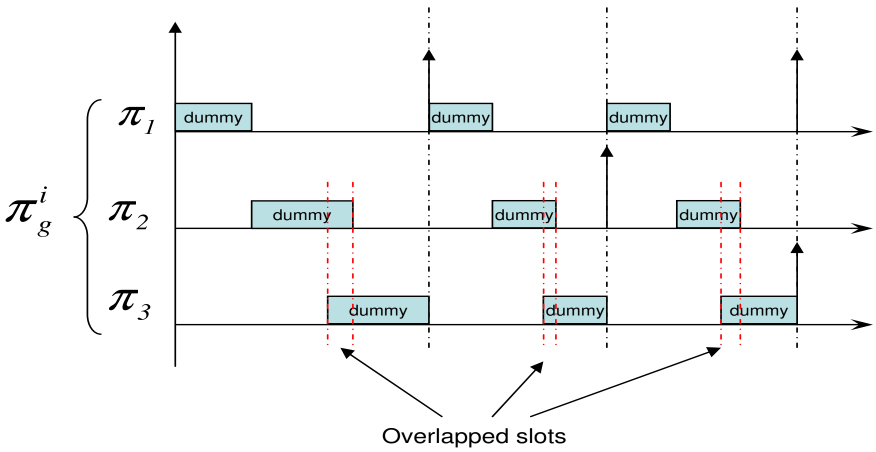


FIGURE B.2: Distribution of under-utilization between release instants

of this slot C_d^i, is calculated as follows:

$$C_d^i = interval\_length \times \left(1 - \sum_{k=1}^{ni} C_k/P_k\right)$$

B.1.2 Global Scheduler

As, we have explained earlier that global tasks are grouped together into k groups, such that sum of utilization of each group is greater than or equal to one, and less than two. Shared task is broken into two tasks of same period as that of original task, and execution of this task is such that sum of utilization of tasks of one group comes out to be 100%. Now, we have k mono-processor EDF based schedulers at global level, and tasks selected by this scheduler executes at time reserved for dummy tasks on different processor of the group. There is always a slot reserved for dummy task on one processor of the group. Execution of shared task ,broken into two tasks, is mutually exclusive, and is ensured by the middleware (Fig:B.3). This middleware provides the synchronization between local and global tasks.

B.2 Schedulability Analysis

Task set is said to be schedulable, if all tasks respect their deadlines. We have two types of subsets of tasks, partitioned and global tasks. Global tasks are scheduled by k mono-processor EDF schedulers, and there is always slots reserved for it. Hence, global tasks respect their deadlines. Local tasks are scheduled only at those slots, when these slots are not reserved for dummy tasks. So, there is a need to prove the schedulability for local tasks. At each release instant R_k, local tasks are given processor time in proportion

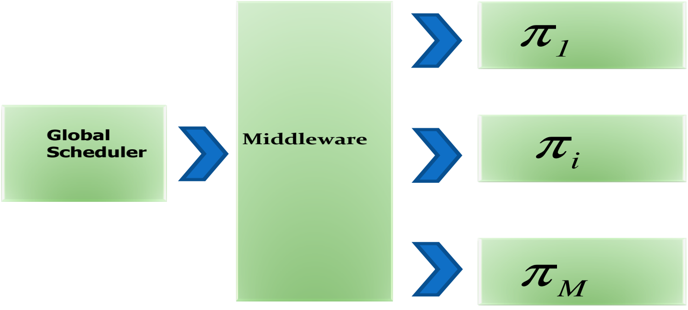


FIGURE B.3: Hybrid schedulers' hierarchy

to their overall weights of partitioned tasks until next release instant R_{k+1}. To provide timeline guarantees, we need to ensure that time allocated to local task is more than or equal to minimum demand of local tasks of one processor until time t (t represent the local release instants on processor Π_j). We perform demand bound function $DBF(T_j, t)$ analysis to calculate the minimum processor demand until any time t. Let time given to local tasks on processor is represented by *Allocation*, then we know that:

$$Allocation = t \times \sum_{j=1}^{ni} \frac{C_j}{P_j}$$

and minimum demand for timeline guarantees of local task is:

$$\sum_{j=1}^{ni} DBF(T_j, t) = \left(\sum_{j=1}^{ni} \left\lfloor \frac{t}{P_j} \right\rfloor \times C_j \right)$$

These two equation (above) demonstrates that $\sum_{j=1}^{ni} DBF(T_j, t) \leq Allocation$. It implies that local tasks are guaranteed to meet their deadline constraints.

B.3 Runtime Optimizations

Global tasks of one group are scheduled on a specific group of processors. It implies that a task selected by one global EDF scheduler may migrate from one processor of the group to another processor of the group. Moreover it can cause preemption of local tasks processors of the group. We can minimize these migrations of global task and preemptions of local task by taking into account run time parameters of both local and global tasks. If global task takes processor time less than its worst case allotment, then slot reserved on next processor of the group can be eliminated (until next release instant). If local tasks takes processor time less than their worst case allotments, then reserved slot on one processor can be enlarged to remove reserved slot on other processors of the group.

B.4 Comparison

We compare this algorithm with EKG [7] approach analytically. Our proposed algorithm introduce less number of preemptions than appearing EKG based schedule, as number of global tasks executing between two release instants is ng in case of EKG, while our algorithm executes global tasks less than or equal to ng between two release instants.

Bibliography

[1] http://www.cofluentdesign.com.

[2] James H. Anderson. Towards a more efficient and flexible pfair scheduling framework. 1999.

[3] James H. Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *J. Comput. Syst. Sci.*, 68(1):157–204, 2004.

[4] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 337–346, 2000.

[5] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA'00)*, page 337, Washington, DC, USA, 2000. IEEE Computer Society.

[6] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. Technical report, Chapel Hill, NC, USA, 2001.

[7] Bjorn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, Washington, DC, USA, 2006. IEEE Computer Society.

[8] M. Arora, S. K. Das, and R. Biswas. A de-centralized scheduling and load balancing algorithm for heterogeneous grid environments. In *Proceedings of Workshop on Scheduling and Resource Management for Cluster Computing*, pages 499–505, Vancouver , Canada, august 2002.

[9] A.Shafarenko. The principles and construction of snet. 2006.

[10] Hakan Aydi, Pedro Mejía-Alvarez, Daniel Mossé, and Rami Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 95, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Hakan Aydin, Rami Melhem, and Pedro Mejia-alvarez. Optimal reward-based scheduling for periodic real-time tasks. In *IEEE Transactions on Computers*, pages 111–130. IEEE Computer Society Press, 1999.

[12] Senior Member-Theodore P. Baker. An analysis of edf schedulability on a multiprocessor. *IEEE Trans. Parallel Distrib. Syst.*, 16(8):760–768, 2005.

[13] S. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, 2003.

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[15] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 137–144, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Sanjoy Baruah and Nathan Fisher. The feasibility analysis of multiprocessor real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 85–96, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Sanjoy K. Baruah, Johannes Gehrke, and C. Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 280–288, Washington, DC, USA, 1995. IEEE Computer Society.

[18] Kostas Bousias and Chris R. Jesshope. The challenges of massive on-chip concurrency. In *Asia-Pacific Computer Systems Architecture Conference*, pages 157–170, 2005.

[19] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 288, Washington, DC, USA, 1995. IEEE Computer Society.

[20] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *IEEE Real-Time Systems Symposium*, pages 295–304, 2000.

[21] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical report, Knoxville, TN, USA, 1995.

[22] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 101–110, Dec. 2006.

[23] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd, and Toshiba Corporation. Advanced configuration and power interface specification, 2006.

[24] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[25] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 144–152, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. *Wirel. Netw.*, 8(5):507–520, 2002.

[27] Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 183, Washington, DC, USA, 2001. IEEE Computer Society.

[28] Ricardo Gonzalez, Benjamin M. Gordon, and Mark A. Horowitz. Supply and threshold voltage scaling for low power cmos. *IEEE Journal of solid-State Circuits*, 32:1210–1216, 1997.

[29] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[30] Flavius Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 46–51, New York, NY, USA, 2001. ACM.

[31] Rami Melhem Hakan Aydin, Bruce Childers. Compiler-assisted dynamic power-aware scheduling for real-time, 2001.

[32] Philip Holman and James H. Anderson. Guaranteeing pfair supertasks by reweighting. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 203, Washington, DC, USA, 2001. IEEE Computer Society.

[33] Philip Holman and James H. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems(ECRTS'03)*, pages 41–50, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Philip Holman and James H. Anderson. Group-based pfair scheduling. *Real-Time Syst.*, 32(1-2):125–168, 2006.

[35] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. Design and implementation of power-aware virtual memory. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[36] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 37–46, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[37] Ravindra Jejurikar and Rajesh Gupta. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 78–81, New York, NY, USA, 2004. ACM.

[38] Ravindra Jejurikar and Rajesh Gupta. Optimized slowdown in real-time task systems. *IEEE Trans. Comput.*, 55(12):1588–1598, 2006.

[39] Mehdi Kargahi and Ali Movaghar. Non-preemptive earliest-deadline-first scheduling policy: A performance study. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–210, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, November 1998.

[41] Woonseok Kim, Jihong Kim, and Sang Lyul Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 393–398, New York, NY, USA, 2004. ACM.

[42] C. M. Krishna and Yann-Hang Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. *IEEE Trans. Comput.*, 52(12):1586–1593, 2003.

[43] Wei kuan Shih and Jane W. S. Liu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24:58–68, 1991.

[44] Sylvain Lauzac, Rami Melhem, and Daniel Mosse. An efficient rms admission control and its application to multiprocessor scheduling. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 511–518, Washington, DC, USA, 1998. IEEE Computer Society.

[45] J. P. Lehoczky, L. Sha, and Y. Ding. Rate-monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of the 11th IEEE Real-time Systems Symposium*, pages 166–171, Dec. 1989.

[46] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442, 1995.

[47] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[48] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[49] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 25–33, 2000.

[50] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with pace. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–61, New York, NY, USA, 2001. ACM.

[51] Lars Lundberg. Analyzing fixed-priority global multiprocessor scheduling. In *RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, page 145, Washington, DC, USA, 2002. IEEE Computer Society.

[52] Lars Lundberg and Håkan Lennerstad. Global multiprocessor scheduling of aperiodic tasks using time-independent priorities. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 170, Washington, DC, USA, 2003. IEEE Computer Society.

[53] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 294–303, 1999.

[54] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[55] Linwei Niu and Gang Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *CASES '04: Proceedings of the 2004 international*

conference on Compilers, architecture, and synthesis for embedded systems, pages 140–148, New York, NY, USA, 2004. ACM.

[56] Ray Obenza and Geoff. Mendal. Guaranteeing real time performance using rma. In *The Embedded Systems Conference*, San Jose, CA, USA, 1998.

[57] S.-H. Oh and S.-M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *RTCSA '98: Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, page 31, Washington, DC, USA, 1998. IEEE Computer Society.

[58] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Syst.*, 9(3):207–239, 1995.

[59] Yingfeng Oh and Sang H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical report, Charlottesville, VA, USA, 1995.

[60] Richard P., Goossens J., and Fisher N. Approximate feasibility analysis and response-time bounds of static-priority tasks with release jitters. In I. Puaut, editor, *15th International Conference on Real-Time and Network Systems*, pages 105–112, March 2007.

[61] Katarina Paulsson, Michael Hbner, Jrgen Becker, Jean-Marc Philippe, and Christian Gamrat. On-line routing of reconfigurable functions for future self-adaptive systems - investigations within the ther project. In Koen Bertels, Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors, *FPL*, pages 415–422. IEEE, 2007.

[62] Trevor Pering and Prof Robert Brodersen. Energy efficient voltage scheduling for real-time operating systems. In *In Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS98, Work in Progress Session*, 1998.

[63] Trevor Pering, Thomas Burd, and Robert Brodersen. Voltage scheduling in the iparm microprocessor system. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 96–101. ACM, 2000.

[64] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81. ACM, 1998.

[65] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 35(5):89–102, 2001.

[66] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259. ACM, 2001.

[67] Johan Pouwelse, Koen Langendoen, and Henk Sips. Energy priority scheduling for variable voltage processors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 28–33. ACM, 2001.

[68] Gang Quan, Linwei Niu, Xiaobo Sharon Hu, and Bren Mochocki. Fixed priority scheduling for reducing overall energy on variable voltage processors. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 309–318, Washington, DC, USA, 2004. IEEE Computer Society.

[69] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1997.

[70] Gerald Sabin, Rajkumar Kettimuthu, and Arun Rajan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In *in the Proc. of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes In Computer Science*, pages 87–104, 2003.

[71] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pages 53–60, Jun 1998.

[72] T. Sakurai and A. Newton. Alpha-power law mosfet model and its application to cmos inverter delay and other formulas. 25:584–594, 1990.

[73] R. Sosic and J. Giddy. Nimrod: A tool for performing parametised simulations using distributed workstations. In *4th IEEE Symposium on High Performance Distributed Computing*, 1995.

[74] D. B. Stewart and Pradeep Khosla. Real-time scheduling of sensor-based control systems. In *IEEE Workshop on Real-Time Operating Systems and Software (RTOS '91)*, pages 144 – 150, May 1991.

[75] Hsin-Wen Wei, Yi-Hsiung Chao, Shun-Shii Lin, Kwei-Jay Lin, and Wei-Kuan Shih. Current results on edzl scheduling for multiprocessor real-time systems. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 120–130, Washington, DC, USA, 2007. IEEE Computer Society.

[76] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating*

Systems Design and Implementation, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[77] Le Yan, Jiong Luo, and Niraj K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 30, Washington, DC, USA, 2003. IEEE Computer Society.