

Ordonnancement temps réel de tâches parallèles pour les systèmes multiprocesseur - Etude du modèle Fork-Join

QAMHIEH Manar



ECE
PARIS ÉCOLE D'INGÉNIEURS



Table des matières

Abstract	3
1 Introduction	4
1.1 Laboratoire d'accueil	4
L'Université Paris-Est et le Laboratoire d'accueil	4
1.2 Mission and goals	5
2 Les systèmes informatique et le temps réel	7
2.1 Introduction	7
2.2 Real-time tasks	7
2.2.1 Les tâches périodiques	7
2.2.2 Les autres types de tâches	7
2.2.3 Quelques définitions	8
2.3 L'ordonnancement	8
2.3.1 Les priorités des tâches	8
2.3.2 Ordonneurs et ordonnancement	8
2.3.3 Ordonnancement à priorités statiques	9
2.3.4 Ordonnancement à priorités dynamiques	9
2.4 Multiprocessor systems	11
3 Parallelism in real-time systems	12
3.1 Parallel tasks of fork-join model	13
3.2 Scheduling parallel FJ tasks	14
3.2.1 Task Stretch Transformation TST	15
3.2.2 Segment Stretch Transformation SST	17
3.3 Analysis and results	19
3.3.1 Simulator : <i>rtmsim</i>	19
3.3.2 Analysis	21
4 Work in Progress	24
4.1 Predictability of fork-join model	24
4.2 Response time analysis	25
5 Conclusion	26
Bibliographie	27
A Simulator code	28
A.1 ParallelTask.java	28
A.2 FloatTask.java	33
A.3 BasicFjDms.java	34
A.4 FjDms.java	35
A.5 FjDmsSegmentStretch.java	37
A.6 FbbFfd.java	39

Abstract

Because multiprocessor systems are widely manufactured nowadays, it is vital to adapt the development of software as well in order to get the most advantage that model can afford. From a practical aspect, parallel programming have been used for many years so as to increase the performance of application computing and speed up massive calculations.

On another hand, real-time scheduling on multiprocessor systems usually analyse tasks of sequential model of programming, and recently it started to integrate parallel programming in real-time systems. But the usual scheduling theories and analysis and tests have to be modified to be adapted to this specific model, and other techniques have to be proposed.

In this research internship, we focus on a specific model of parallel tasks called the fork-join model, which is the same model used in OpenMP programming library in C/C++ and Fortran. and we study the scheduling of periodic implicit-deadline fork-join real-time tasks on multiprocessor systems. Since this model of tasks have strict parallel segments without laxity, we propose a partitioned scheduling algorithm which increases the laxity of the parallel segments and therefore the schedulability of tasksets of this model.

Since there is a similar algorithm has been proposed in the literature which produces job migrations. Our algorithm eliminates the use of job migrations in order to create a portable algorithm that can be implemented on a standard Linux kernel.

Results of extensive simulations are provided in order to analyze the schedulability of the proposed algorithm, and to provide comparisons with the other algorithm proposed in the literature.

Chapitre 1

Introduction

Ce stage ingénieur, orienté recherche constitue mon stage de fin d'études à l'ECE Paris Ecole d'Ingénieurs. Il a été effectué au sein du laboratoire d'informatique *Gaspard Monge* (LIGM) de *l'Université Paris-Est*.
This duration of this internship will be 6 months starting from 21st of February till the 21st of August.

1.1 Laboratoire d'accueil

L'université Paris-Est est un pôle de recherche et d'enseignement supérieur (PRES). C'est un établissement Public de Coopération Scientifique (EPCS) regroupant des universités, des grandes écoles, des organismes de recherche et plusieurs écoles d'ingénieurs et d'architecture dans un même ensemble pluridisciplinaire.

Intégrant recherche et enseignements généraux, technologiques et professionnels, le PRES est doté de compétences stratégiques déléguées par ses membres parmi lesquelles : - la coopération scientifique et la valorisation de la recherche.

- la signature commune des publications scientifiques - la formation doctorale et la délivrance du diplôme de docteur.
- la gestion des habilitations à diriger les recherches (HDR).
- la mobilité et l'ouverture internationale : doctorants à chercheurs et professeurs invités.
- la contribution au développement économique et social de son territoire.

La Recherche d'Université Paris-Est s'appuie sur ses 6 écoles doctorales, notamment chargées d'assurer la formation des doctorants et leur bonne intégration dans les équipes de recherche qui composent Université Paris-Est :

Cultures et sociétés (CS).

Mathématiques et STIC (MSTIC).

Organisations, marchés, institutions (OMI).

Sciences, ingénierie et environnement (SIE).

Sciences de la vie et de la santé (SVS).

Ville, transports et territoires (VTT).

Chacune des écoles doctorales correspond à un département scientifique qui organise la coopération entre unités de recherche et favorise le développement international ainsi que les projets communs.

Le LIGM (Laboratoire d'informatique Gaspard-Monge) est une unité mixte de recherche CNRS (UMR 8049). Le laboratoire est intégré au département Mathématiques et STIC (MSTIC) du PRES Paris-Est. Les tutelles sont UPEMLV, ESIEE Paris, école des Ponts ParisTech, CNRS. Les thèmes de recherche du laboratoire sont : Algorithmique - Combinatoire algébrique et calcul symbolique - Informatique linguistique - Géométrie Discrète et Imagerie - Signal et communication.

1.2 Mission and goals

J'ai été intégrée dans l'équipe Algorithmique au sein du groupe *AlgoTR*. Ce groupe s'intéresse à la problématique de l'algorithme temps réel. Le stage est encadré par Serge MIDONNET maître de conférence à l'IGM et Frédéric FAUBERTEAU doctorant au LIGM.

The first and the main goal of this research internship, is to work on the problematic of parallelism in real-time multiprocessor systems, and study the various approaches of this subject in order to choose a specific domain to work on. The necessary background is to be obtained by reading different articles and papers published about real-time scheduling theories on multiprocessor systems in general, and parallelism in specific.

But since it is a six-month internship and this subject has many aspects to be studied, the research had to focus on a specific model of parallelism. So as a second step, a specific model of parallel real-time tasks was to be chosen to work on. From a practical point of view, the fork-join model of parallel programming was an interesting choice since it is the model on which parallel programming libraries like OpenMP are based on. We studied thoroughly algorithms and techniques for this type of tasks, and proposed our own algorithm to schedule the task model.

On another hand, this internship had a development side, includes working on a real-time simulator, designed specially to expand the ability of researchers to compare the performance of different algorithms through extensive simulation and task generation. This simulator is already under active development, and my mission was to integrate parallelism into the simulator in order to verify the efficiency of the solutions proposed during the internship and after.

Being able to publish papers and participate in real-time conferences is desired goal, by which young researchers can be introduced to the other researchers around the world who are working in the same domain, and it is a great way to get involved in the new approaches and researches about real-time systems. During the internship we presented our research in the work-in-progress session of the 23rd edition of the Euromicro Conference on Real-Time Systems 2011 "ECRTS'11" [1], which took place at Porto, Portugal and was attended by a large number of researchers in the domain of real-time systems from all over the world.

Finally, the following gantt diagram 1.1 is provided to show the progress of the internship during the period from 21/2/2011 to 21/8/2011, specifying the general steps of work with their estimated durations. After the end of this internship, I'll continue working on the same subject "Real-time Scheduling of Parallel Task on Multiprocessor Systems" as a PhD student at the department of Algorithm in LIGM, Paris-est under the supervision of M.Serge MIDONNET.

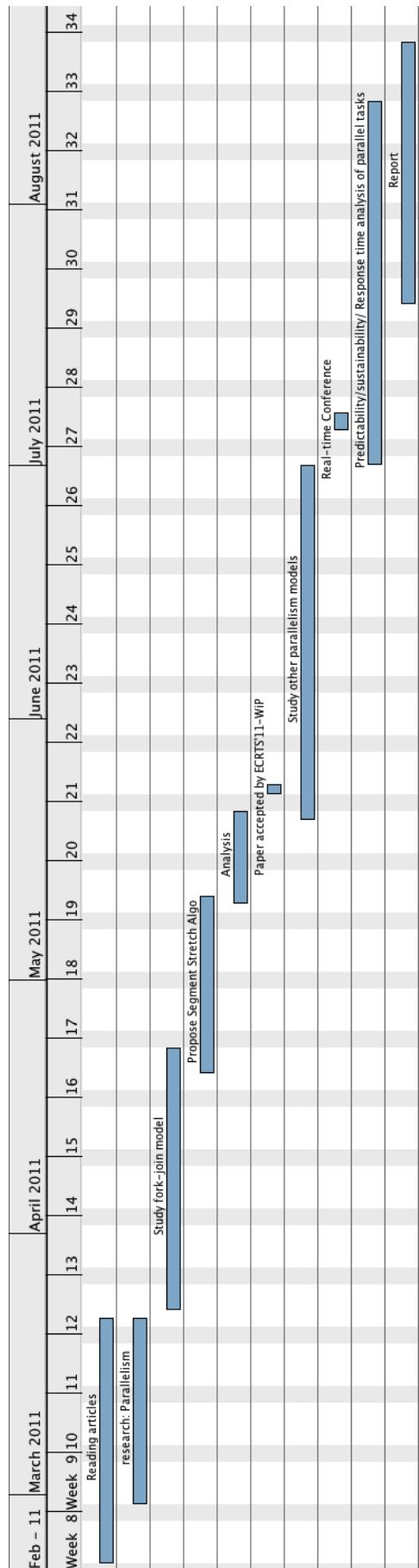


FIGURE 1.1 – gantt diagram showing the progress of the internship.

Chapitre 2

Les systèmes informatique et le temps réel

2.1 Introduction

Contrairement à ce que l'on peut penser lorsque l'on découvre le "temps réel", il ne s'agit pas d'une garantie de rapidité mais d'une garantie de "résultats". En effet, un système temps réel est un système informatique qui doit réagir à son environnement en respectant un certain nombre de contraintes temporelles. Son comportement dépend donc des résultats du calcul de chacune de ses tâches mais aussi, et surtout, du moment auquel est produit ce résultat. Ainsi, l'exactitude du résultat dépend autant de la valeur du résultat que du moment auquel il est produit.

Les applications temps réel couvrent un très grand domaine : applications embarqués pour l'automobile, système de contrôle de vol, de circulation des trains, robotique

Il existe plusieurs niveaux de contraintes pour un système temps réel (STR) :

- STR à contraintes fixes (temps réel dur) : le non respect des contraintes temporelles peut conduire à des catastrophes (exemple avec l'industrie aéronautique)
- STR à contraintes relatives (temps réel souple) : le non respect des contraintes temporelles est toléré, sans conséquences dramatiques (multimédia)
- STR à contraintes mixtes composé de tâches temps réel dur et de tâches temps réel souples.

2.2 Real-time tasks

2.2.1 Les tâches périodiques

Une tâche temps réel se différencie d'une tâche non temps réel essentiellement par l'existence d'une échéance temporelle. Plus précisément, pour chaque tâche τ_i sont définies les caractéristiques suivantes :

- la date d'activation : r_i
- la durée d'exécution ou durée de calcul (charge processeur) : C_i
- l'échéance ou délai critique : D_i . Il s'agit du temps alloué à la tâche pour terminer son exécution.
- la période d'activation : T_i . Il s'agit de la durée fixe ou minimale entre les activations de deux instances successives.

La figure ??, illustre un modèle de tâches périodiques.

Ainsi, une tâche périodique est caractérisée par le quadruplet $\tau_i=(r_i, T_i, C_i, D_i)$, où r_i est la date d'activation de la première requête de τ_i , T_i la période d'activation des suivantes, C_i le pire temps d'exécution et D_i son échéance. Ce modèle correspond aux tâches de contrôle qui vont, par exemple, périodiquement aller vérifier la valeur d'une donnée.

2.2.2 Les autres types de tâches

Il existe cependant deux autres grandes catégories de tâches : les tâches apériodiques et sporadiques.

Les tâches apériodiques sont des tâches activées à des instants aléatoires. Elles ne sont pas prévisibles, peuvent intervenir à n'importe quel instant et peuvent se réactiver plusieurs fois sans que l'on ne soit en mesure de prédire leur prochaine arrivée. Cependant, ces tâches possèdent elles-aussi une échéance. Si l'on se place dans le cas où de nombreuses tâches apériodiques sont activées quasi simultanément, il est alors impossible de garantir que les échéances de chacune de ces tâches seront respectées.

Afin de pallier à ce problème, il est nécessaire d'introduire les tâches sporadiques. Il s'agit, en quelque sorte, de tâches apériodiques mais dont la définition est plus précise. En effet, elles peuvent se caractériser par un délai minimum entre 2 activations ce qui permet une meilleure gestion/analyse de ces dernières.

2.2.3 Quelques définitions

Ce paragraphe présente rapidement les notions importantes utilisées lors du stage.

Des tâches sont *dépendantes* (resp. *indépendantes*) si elles partagent d'autres ressources ou sont reliées par des contraintes de précédence (resp. si elles ne partagent que le processeur).

Un ensemble de tâches est dit *synchrones* si leur activation est simultanée.

Une tâche est dite *préemptible* si, lorsqu'elle est en cours d'exécution, elle peut être arrêtée et remise à l'état "prêt" afin de céder sa place à une autre tâche.

2.3 L'ordonnancement

La fonction première d'un ordonnanceur dans un système informatique classique est de fournir et de répartir les ressources processeur entre les différents processus. L'objectif en est principalement double : donner l'illusion aux utilisateurs que le système est capable d'effectuer plusieurs actions simultanément et empêcher qu'une seule tâche occupe tout le processeur empêchant ainsi les autres tâches de s'exécuter. Dans un tel système, l'ordonnanceur n'a pas besoin de prédire à l'avance le comportement des tâches.

Tout ceci n'est plus valable dans le cas de systèmes temps réel. En effet, les tâches et les ressources dont elles ont besoin doivent être connues à l'avance. De plus, l'ordonnanceur doit, dans ce cas, analyser le système et vérifier qu'il est "faisable", c'est à dire que toutes les contraintes temporelles seront bien respectées lors de son exécution.

Il est également nécessaire de faire la distinction entre tâches périodiques et apériodiques. En effet, l'ordonnanceur connaît parfaitement les premières alors qu'il ne dispose que de très peu d'informations sur les dernières. Il n'est donc pas possible de gérer celles-ci de la même manière.

2.3.1 Les priorités des tâches

Afin de permettre l'ordonnancement de l'ensemble des tâches, il est nécessaire que ces dernières se voient attribuées, sous forme d'entier, une priorité représentant l'importance de chacune d'entre elles. Il est alors possible de faire la distinction entre les systèmes à priorités fixes et les systèmes à priorités dynamiques.

Dans le premier cas, les priorités des tâches sont assignées par les algorithmes lors de la conception du système ; de plus, pendant l'exécution, chaque instance hérite de la priorité de sa tâche. A contrario, les priorités des tâches dans les systèmes à priorités dynamiques sont assignées pendant l'exécution du programme et évolue dynamiquement.

2.3.2 Ordinanceurs et ordonnancement

Il existe deux grandes familles d'ordonnanceur. Les ordonneurs préemptifs peuvent interrompre une tâche au profit d'une autre plus prioritaire alors que les non préemptifs n'arrêtent jamais l'exécution de la tâche courante.

Il existe également deux grandes méthodes d'ordonnancement : hors ligne ou en ligne. Un ordonnancement hors ligne est pré-calculé avant exécution, une séquence d'ordonnancement est générée, implantée dans un table et répétée à l'infini par un séquenceur.

Dans le cas d'un ordonnancement en ligne, l'algorithme est implanté dans le noyau du système et exécuté à chaque instant. Il décide dynamiquement, avec ou sans préemption, de l'exécution des tâches. Il

consiste à assigner une priorité à chacune des tâches et à toujours choisir la tâche avec la plus forte priorité lorsqu'une décision d'ordonnancement doit être prise. Il existe de nombreuses politiques d'assignation de priorités aux tâches, politiques pouvant être statiques (les priorités sont établies une fois pour toutes et n'évoluent plus au cours du temps) ou dynamiques (les priorités varient en fonction du temps).

Ce sous-chapitre présente rapidement les grandes techniques d'ordonnancement les plus répandues sans aborder les conditions nécessaires et suffisantes de faisabilité.

2.3.3 Ordonnancement à priorités statiques

Avec ce type d'ordonnanceur, chaque tâche se voit attribuer une priorité avant son exécution. Ce type d'ordonnancement n'est pas le plus performant en terme d'utilisation de ressources processeur.

Tâche	Période	Echéance	Coût
τ_1	6	6	2
τ_2	7	4	3
τ_3	15	15	3

TABLE 2.1 – Exemple de systèmes de tâches temps réel

Algorithme Rate Monotonic (RM)

Cette politique assure la priorité maximale à la tâche de plus petite période. Les hypothèses d'optimalité d'un tel algorithme sont la présence de tâches périodiques, préemptibles et indépendantes ayant une activation en début de période, une échéance en fin de période et un pire cas d'exécution connu à priori.

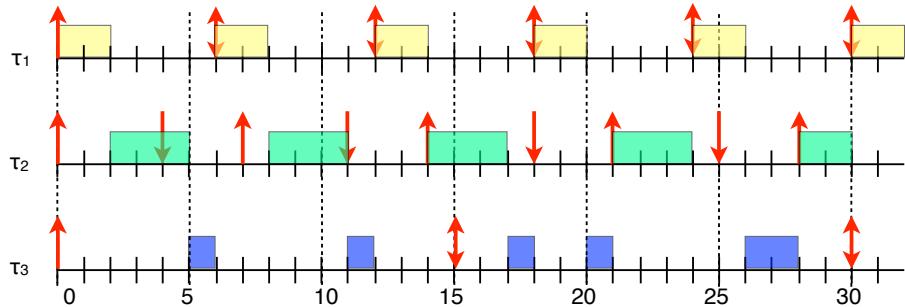


FIGURE 2.1 – Ordonnancement RM du système de tâches du tableau 2.1

Chaque activation de tâche réveille l'ordonnanceur qui choisit la tâche éligible de plus courte période. Une telle politique permet un taux d'utilisation du processeur inférieur à 69% pour les scénarios les moins favorables, valeur pouvant atteindre 88% pour des systèmes générés aléatoirement.

La figure 2.1 fournit un exemple d'ordonnancement avec RM des tâches présentées dans le tableau 2.1. On remarque que les tâches τ_2 et τ_3 dépassent leurs échéances, on parle alors de "défaillances temporelles".

Algorithme Deadline Monotonic (DM)

Cette politique assure la priorité maximale à la tâche dont l'échéance relative est la plus petite. Les hypothèses d'optimalité sont identiques à Rate Monotonic à la différence que l'échéance doit être inférieure à la période. Cet algorithme classe les tâches par échéances croissantes.

La figure 2.2 fournit un exemple d'ordonnancement avec DM des tâches présentées dans le tableau 2.1. On remarque que τ_2 devient plus prioritaire que τ_1 et donc ne rate pas son échéance. En revanche, τ_3 est toujours en défaillance temporelle.

2.3.4 Ordonnancement à priorités dynamiques

Ce type d'ordonnanceur permet aux priorités des tâches de changer entre deux requêtes.

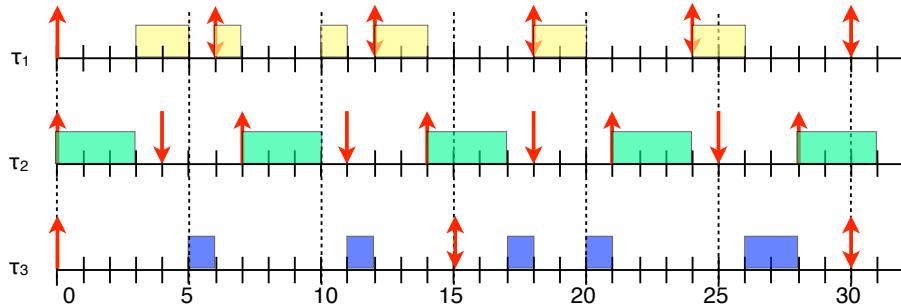


FIGURE 2.2 – Ordonnancement DM du système de tâches du tableau 2.1

Earliest Deadline First (EDF)

Le principe de cette politique d'ordonnancement est qu'à chaque instant, la priorité maximale est donnée à la tâche dont l'échéance est la plus proche. Les hypothèses d'optimalité sont identiques à l'algorithme Rate Monotonic. Cette politique permet d'atteindre un taux d'utilisation du processeur de 100%.

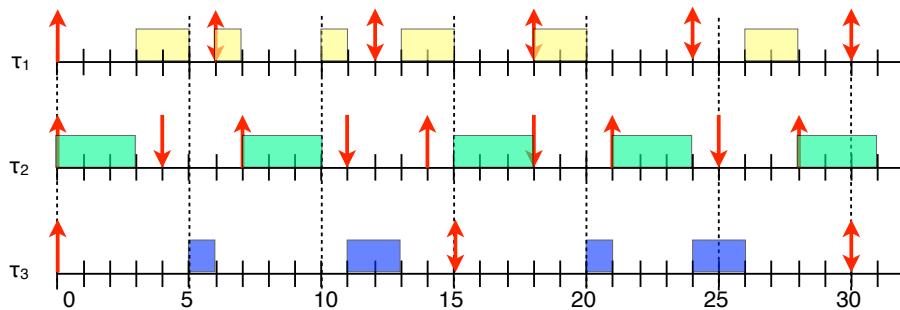


FIGURE 2.3 – Ordonnancement EDF du système de tâches du tableau 2.1

La figure 2.3 fournit un exemple d'ordonnancement avec EDF des tâches présentées dans le tableau 2.1.

Least Laxity First (LLF)

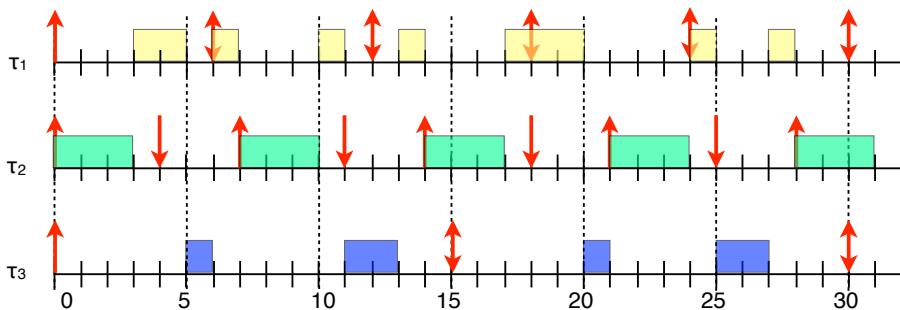


FIGURE 2.4 – Ordonnancement LLF du système de tâches du tableau 2.1

Le principe de cet algorithme est qu'à chaque instant, la priorité maximale est donnée à la tâche dont la laxité dynamique est la plus petite. La laxité dynamique est le résultat de : échéance - temps de calcul restant - temps courant. Les conditions d'optimalité sont similaires à celles d'EDF.

Cet algorithme est meilleur que l'algorithme EDF dans le cas d'utilisation de multiprocesseur mais son implantation est très complexe.

La figure 2.4 fournit un exemple d'ordonnancement avec LLF des tâches présentées dans le tableau 2.1. Jusqu'à l'instant 14, l'ordonnancement est équivalent à EDF. Les laxités des tâches τ_1 , τ_2 , τ_3 à ce moment là sont alors respectivement 3, 1, 13. La tâche τ_2 préempte donc la tâche τ_1 .

2.4 Multiprocessor systems

A multiprocessor system is a computer system in which 2 or more identical CPU units are connected to one another through shared memory and run under the same operating system. This concept increases the performance of the system by allowing it to execute multiple concurrent processes at the same time. As a result, certain concepts have been created like multitasking and parallel programming.

The idea of multiprocessors is not new, these systems have been designed and sold since the late 1950s, and it was exactly 1958 when IBM introduced the first multiprocessor system followed by other companies (Digital Equipment Corporation, Control Data Corporation, ...). While multiprocessors were being developed, technologies also advanced to be able to build smaller processors operating at higher clock rates, which made the uniprocessor systems much popular for a while. Nowadays, manufacturers returned to developing multiprocessor systems in order overcome the physical constraints of processor's miniaturization (power consumption and heat).

Real-time scheduling on multiprocessor systems is described to be more difficult than uniprocessor scheduling, due to the necessity of synchronization between processors, and task/job migration liabilities. So, theories and techniques have to be developed to be used on this specific model of programming.

Multiprocessors can be divided into 2 categories :

- Heterogeneous : where processors of the same system are different.
- Homogeneous : where processors of the same system are identical.

According to real-time scheduling algorithms, multiprocessor systems are classified as the following :

- Global scheduling : In which tasks are allowed to migrate between processors.
- Partitioned scheduling : No migration is allowed.

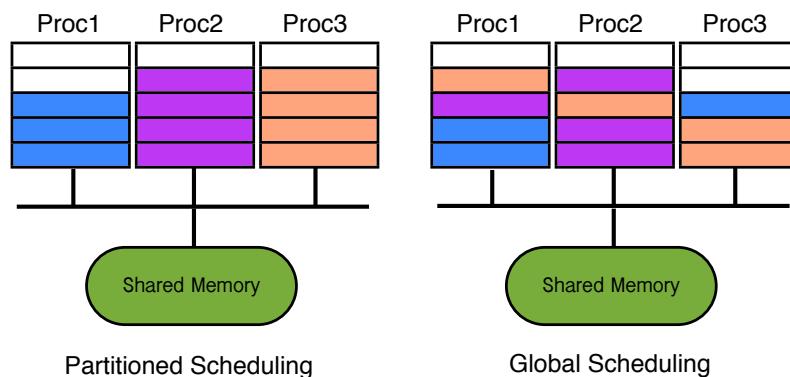


FIGURE 2.5 – Global vs. partitioned scheduling.

Chapitre 3

Parallelism in real-time systems

Physical constraints of the manufacturing process, such as chip size and heating, are driving a tendency for chip manufacturers to build multi-processors and multi-core processors to further increase computational performance . Because of this, parallel programming has received increased attention, despite being in use for many years.

Traditionally, parallel programming is the model of programming in which processes execute on multiple processors simultaneously. These programs are typically harder to write than sequential ones, since more synchronization and in-dependency constraints are needed in the parallel design, and system limitations like a shortage in processors require the use of certain techniques like partitioning.

There are different models of parallel programming. For example, MapReduced, which is developed by Google for the purpose of processing large amounts of raw data, distributed on a number of processing machines by using the same computations on each. Another model is the fork-join model which consists of both, sequential and parallel regions, starting always by a single thread, which forks into several parallel parts and then they all join again in a single thread, and so on..., we used this model in our research and it is explained in details later.

Parallelism in real-time systems is an immature subject, and it has been studied and analysed recently. As in normal sequential tasks, parallel real-time tasks will have temporal constraints, like period, deadline, offset,But because of the parallelism, they will need a more-adapted scheduling algorithms and analysis techniques. In literature [2, 3], we found that parallel tasks can be described as :

- Rigid : if the number of processors assigned to the task is specified by the model externally to the scheduler, and it is fixed during execution.
- Moldable : if the number of processors assigned to the task is specified by the scheduler and it is fixed during execution.
- malleable : if the number of processors assigned to the task can be changed by the scheduler during the execution.

In general, real-time tasks can be classified into 3 categories according to the value of their deadlines regarding the period of the task :

- Implicit deadline : where the deadline of the task is equal to the period ($D_i = T_i$).
- Constrained deadline : where the deadline of the task is less or equal to the period ($D_i \leq T_i$).
- Arbitrary deadline : there is no restraints on the value of the deadline, it could be less, more or equal to the period of the task.

The general mission in this research internship was to study the schedulability of periodic parallel real-time tasks on multiprocessor systems, and in order to achieve this goal, a bibliography of papers regarding this subject had to be read. for example, Collette *et al.* [4] in which a study of schedulability of sporadic tasks on multiprocessor systems is proposed, as well as a task model which integrates job parallelism.

Another paper by Goossens *et al.* [2] studies the predictability of periodic parallel rigid tasks and provide predictable schedulers. The parallel model of forj-join is described in the paper of Lakshmanan *et al.* [5], this model is widely used in practical parallel programming, and it is the same model we are interested in, and we consider as the basis of our research as it will be described later.

3.1 Parallel tasks of fork-join model

Parallel Fork-join (FJ) model is an efficient parallelism design and is commonly used in parallel programming, like the OpenMP library [6], in which certain regions of a task are executed in parallel over a fixed number of processors at the same time. The design consists of a collection of segments, both sequential and parallel. The task always starts as a single process called the master thread, which executes sequentially till it forks into a number of parallel segments forming a parallel region, after all the parallel segments in the region finish their execution, they join together in the master thread, and so on... . According to this model, the number of parallel segments in all the parallel regions in the same task is fixed by the model, but each task may have a variant number of parallel regions, which makes it a rigid model of real-time parallelism.

The fork-join model can be presented as following :

$$\tau_i = ((C_i^1, P_i^2, C_i^3, \dots, P_i^{k-1}, C_i^k), n_i, T_i, D_i) \quad (3.1)$$

where :

- k is the total number of segments (sequential and parallel) and it is an odd number according to definition of the model,
- n_i is the number of parallel threads on which parallel segments will be executed. $n_i > 1$ for parallel segments, and equal to 1 for sequential segments.
- C_i^j is the Worst-Case Execution Time (WCET) of a sequential segment, where j is an odd number and $1 \leq j \leq k$,
- P_i^j is the WCET of a parallel segment, where j is an even number and $1 \leq j \leq k$,
- T_i is the period of the task.
- D_i is the relative deadline of the task.

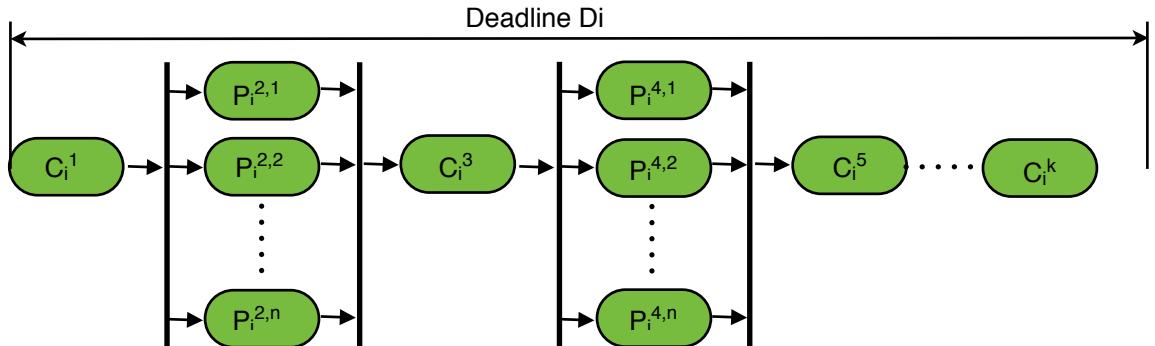


FIGURE 3.1 – Fork-Join structure model.

Another notation can be used to describe the parallel real-time tasks of fork-join model, as follows :

$$\tau_i = \begin{cases} \tau_i^{2j+1}, & \text{where } 0 \leq j \leq \frac{k-1}{2} \text{ (sequential segment)} \\ \tau_i^{2j,l}, & \text{where } 0 \leq j \leq \frac{k-1}{2} \text{ and } 1 \leq l \leq n_i \text{ (parallel segment)} \end{cases} \quad (3.2)$$

Definitions :

The *master thread* of parallel fork-join task is the thread that starts the task, which then forks into several parallel threads, and after execution they join into the master thread again.

The master string of parallel fork-join task is a collection of segments that executes within the master thread of the task, starting by the first sequential segment and then the intermediate parallel and sequential segments, and ending by the last sequential segment. In numerical notations, master string can be represented as :

$$(\tau_i^1, \tau_i^{2,1}, \tau_i^3, \dots, \tau_i^{(k-1),1}, \tau_i^{k,1}).$$

The minimum execution length η_i represents the minimum time a fork-join task τ_i needs to execute when all parallel segments are executed in parallel, it equals to the sum of worst-case execution time of all segments in the master string of task τ_i :

$$\eta_i = \sum_{s=0}^{\frac{k-1}{2}} (C_i^{2s+1}) + P_i^{2s+1} \quad (3.3)$$

The maximum execution length C_i , which is the sum of worst-case execution time of all sequential and parallel segments in task τ_i :

$$C_i = \sum_{s=0}^{\frac{k-1}{2}} (C_i^{2s+1}) + m_i \sum_{s=1}^{\frac{k-1}{2}} P_i^{2s,1} \quad (3.4)$$

Parallel real-time tasks of fork-join model as any other real-time tasks, have global deadlines and periods. But the difference between them and normal sequential tasks is that they consist of segments, both sequential and parallel, that they depend on each other. For example, the activation instant of a segment is a dynamic value depends on the execution time of the previous segments. But if we consider each segment as an independent sub-task, the deadline of this sub-task will have an equal value to the worst-case execution time, and by using this definition, we can consider those sub-tasks as tasks with no laxity, which we have to improve by transformation.

In our work, we considered task model of partitioned periodic implicit-deadline parallel tasks of fork-join model on homogeneous multiprocessor system.

3.2 Scheduling parallel FJ tasks

As a definition, a real-time taskset is said to be feasible if there exists a scheduling algorithm by which the taskset is schedulable over all possible sequences of jobs of the taskset, and execute correctly with respect to the time constraints (deadline, period, offset, ...). Scheduling algorithms and analysis are used to guarantee the performance of the executing tasks in the system while making the best use of processing capacity.

However, scheduling parallel tasks on multiprocessors is different and more complicated than scheduling sequential ones, since parallel tasks have to be executed on multiple processors at the same time, and there exists dependency relations between the different subtasks in the same task. As shown in the fork-join model, the activation instant of segments in the tasks depends on the previous segments, and the fork event occurs when the execution of the previous sequential segments is done.

Figure 3.2 represents a simple example demonstrating the differences between scheduling sequential and parallel tasks, that the first part of the figure shows 3 sequential tasks :

$$\tau_1 = \tau_2 = \tau_3 = (5, 5, 5).$$

These 3 tasks are identical, but each have a fixed task priority according to the indices, where the tasks with low index value have a higher priority, therefore $P_1 > P_2 > P_3$. According to this and as shown in the figure, the 3 tasks are scheduled on a system of 3 homogeneous processors, all the tasks are executing correctly in respect to their deadlines and other time constraints, and the taskset is said to be schedulable.

In the second part of the figure, we converted the same 3 sequential tasks into 3 parallel FJ tasks with the same priority assignment as before :

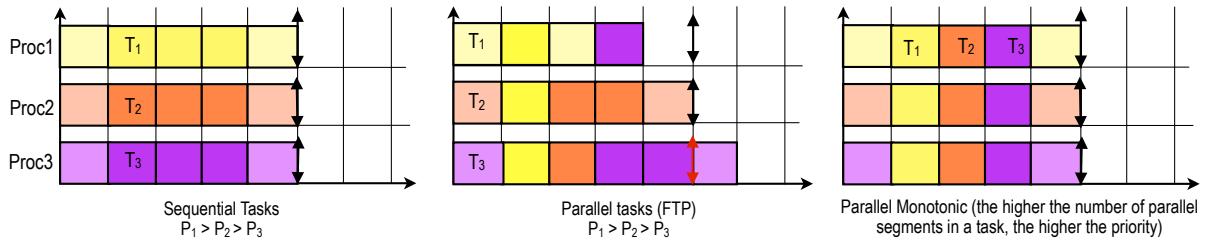


FIGURE 3.2 – Sequential scheduling vs. parallel scheduling

$$\tau_1 = \tau_2 = \tau_3 = ((1, 1, 1), 3, 5, 5), \text{ with } P_1 > P_2 > P_3.$$

With respect to the fixed-priorities of the tasks, the taskset is no longer schedulable on the same homogeneous processors, that the scheduler starts by the task with the highest priority τ_1 which executes successfully on the 3 available processors of the system with laxity of 2. Then τ_2 executes successfully on 2 processors, and blocked only for 1 unit of time by the parallel segments of τ_1 but with no laxity. And finally τ_3 has to wait for the other tasks to finish execution so as it will be able to execute but without being able to finish within its deadline, causing the taskset to be non-feasible.

But if we choose another priority assignment more-adapted for parallel tasks, the taskset will be schedulable. As shown in the third part of Figure 3.2, giving each segment a priority depending on the level of parallelism in the taskset, then the sequential segments will have lower priorities than parallel segments, and since parallel segments all have the same level of parallelism (number of processors to execute on), then in this case, all the parallel regions will execute on the 3 processors without being preempted by the final sequential segments of other tasks, and the taskset is schedulable now.

3.2.1 Task Stretch Transformation TST

Scheduling parallel fork-join tasks is studied recently in a paper published by Lakshmanan *et al.* in [5] and transformation algorithm is provided, because according to the authors, scheduling this model of parallel tasks on multiprocessor systems has almost the same utilization bound of uniprocessor systems, which is an of scheduling this model of tasks. This is considered as a limitation of the model on real-time systems, and has to be improved.

Based on this observation, they proposed a preemptive fixed-priority scheduling algorithm for periodic parallel fork-join tasks, in which they try to avoid the structure of FJ tasks by stretching it when possible.

The transformation will be better explained by an example, Figure 3.3 shows a taskset of 2 periodic, implicit-deadline, parallel fork-join tasks on a system of 3 processors with fixed task priority assignment :

$$\begin{aligned} \tau_1 &= ((1, 2, 2, 3, 1), 3, 17, 17), \\ \tau_2 &= ((15), 1, 18, 18), \\ P_1 &> P_2. \end{aligned}$$

The priorities of the tasks are assigned to each task by applying deadline monotonic, and since τ_1 has a deadline less than τ_2 , then it has a higher priority.

Task τ_2 is a parallel task with one segment only that executes on one processor, which means that this task will execute like a sequential one. Scheduling this taskset under the fork-join structure is not possible as shown in the figure, that τ_1 will pre-empt the execution of τ_2 because it needs 3 processors to execute its parallel segment. And since τ_2 has a lower priority, its execution will be suspended causing it to miss its deadline ($D_2 = 18$). Even though, if the priorities are inverted, τ_2 had a higher priority and will execute first on an exclusive processor, the parallel segments of τ_1 will be blocked, and its execution will be delayed and then it will miss its deadline.

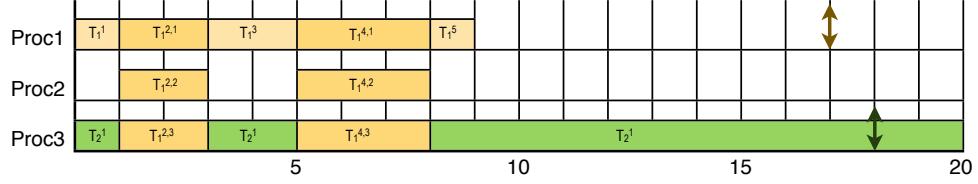


FIGURE 3.3 – Parallel fork-join task example.

Therefore a transformation is proposed, by which the same taskset can be scheduled, as shown in Figure 3.4. The idea is to reduce the parallelism of the parallel tasks, by trying to convert the parallel fork-join task into sequential when it is possible, and stretching the master string of the parallel task to a full master string, containing parallel segments within, so that the parallel task will execute on a single processor with 100% of its utilization.

The slack time of the master thread S_i , which is defined as the difference between the deadline and the minimum execution time eta_i 3.3, will be filled by parallel segments from all the parallel regions of the task. And the master thread will be stretched till it has an execution time equal to the deadline of the task (which has the same value of the period). The following equation shows how many parallel segments can fit in the slack time of task τ_i :

$$f_i = \frac{S_i}{\sum_{s=1}^{\frac{k-1}{2}} C_i^{2s}} \quad (3.5)$$

The number of the complete parallel segments that fits in the master string equals $\lfloor f_i \rfloor$. And if $f_i - \lfloor f_i \rfloor \neq 0$, then one parallel segment of each parallel region in τ_i will start execution on any available processor (defined by the scheduler), and then it will migrate to execute on the processor of the master string so as to fill it.

There exists 2 cases to be considered when applying the transformation, depending on the maximum execution length of parallel fork-join task 3.4 :

- if $C_i \leq D_i$: this means that all the parallel segments will fit in the master string, and the parallel fork-join task will be completely transformed into a sequential task that executes on one processor only.
- if $C_i > D_i$: The master string will be stretched up to its deadline, by filling the slack time of τ_i by parallel segments. The rest of parallel segments will execute on other processors.

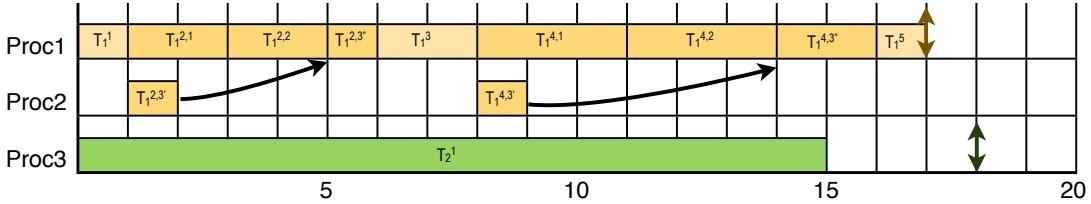


FIGURE 3.4 – Task stretch transformation example.

Applying the transformation to the previous example, the slack time of the master thread of τ_1 equals :

$$S_1 = \frac{17-9}{2+3} = \frac{8}{5} = 1\frac{3}{5}$$

The number of complete parallel segments to fit into the master thread is : $\lfloor \frac{8}{5} \rfloor = 1$, which is the same as shown in Figure 3.4, where both $\tau_1^{2,2}$ and $\tau_1^{4,2}$ (one segment from each parallel region) execute on the 1st processor among the master string. and since there is a difference between the value and its floor value (1.6 is a float number), then migration will happen, and the remaining free space in the master

string (3 units of time) will be filled by parts of parallel segments from each parallel region equally, as shown in the example, segment $\tau_1^{2,3}$ will execute the last unit of time of its WCET in the master string, and $\tau_1^{4,3}$ will fill the remaining space (2 units of time). This ratios are proportional to the execution time of the migrated segment, the bigger the value is the bigger the ratio is. The transformation algorithm is described in details in the original paper [5].

The rest of the parallel segments which are not placed in the master string will be scheduled using a standard partitioned constrained-deadline task scheduling algorithm called FBB-FFD (Fisher Baruah Baker - First Fit Decreasing) [7], which uses the first fit decreasing bin packing heuristic, this heuristic is described as a greedy approximation algorithm, in which it tries to place tasks in the first available processor that satisfies the algorithm's condition, if not, it will place it on an empty processor or the placement will fail.

The input tasks to be scheduled by FBB-FFD are in decreasing order of deadline monotonic priority (DM), this scheduling algorithm is using the following approximate condition to choose processor to place tasks on :

$$d_i - \sum_{\tau_j \in \tau(\pi_k)} RBF^*(\tau_j, d_i) \geq e_i \quad (3.6)$$

where τ_i is the task to be scheduled on processor k , π_k is the set of tasks already placed on processor k and $RBF^*(\tau_j, d_i) = e_j + \frac{e_j}{P_j} * d_i$.

As an advantage to the task stretch transformation, the output master string executes on one processor up to its implicit deadline (and period) having 100% of the processor's utilization. Also this algorithm enhances the schedulability of parallel tasks of fork-join structure, by increasing the local deadlines of parallel segments and getting rid of their strict execution time, as shown in the previous example (Figure 3.3 and 3.4), parallel segment $\tau_1^{2,3'}$ has a deadline of 4 time units instead of 2, which was exactly the worst case execution time of that parallel segment. Then it has to migrate to the master string so as to fill the master thread. This laxity in the deadline will increase the chances of the parallel segments to be scheduled using *FBB-FFD*.

The number of job migrations in this algorithm could be 0, if the algorithm succeeded in scheduling all the parallel segments into the master string (described previously in case $C_i \leq D_i$), creating a sequential task that will be executed on one processor. Other than that, the number of job migrations will be equal to the number of parallel regions in the task, as shown in Figure 3.4, both $\tau_1^{2,3}$ and $P_1^{4,3}$ are used to fill the slack time in the master string, and they both will migrate.

Task Stretch Transformation (TST) has a constraint when it comes to practical implementation, that and according to the paper, job migration can be easily implemented on a specific Linux system called *Linux/RK* [8] (stands for Linux Resource Kernel), which is a real-time extension to the Linux kernel to support the abstractions of a resource kernel, developed by the *Real-time and Multimedia Systems Laboratory* led by *Dr.Raj Rajkumar* at *Carnegie Mellon University, USA*. But our idea is to implement this algorithm directly on a standard Linux enhanced with the *PREEMPT_RT* kernel patch.

3.2.2 Segment Stretch Transformation SST

In order to enhance the task stretch transformation described before, and to eliminate the use of job migration, some modifications have to be done on the original pseudo-code, which we called *Segment Stretch Transformation* (SST). The basic idea of TST stayed the same, by trying to avoid the fork-join model by stretching the master string, but now it will be filled only with complete parallel segments with no migration.

Algorithm 3.1 represents the algorithm of segment stretch transformation, which is written based on the algorithm of task stretch transformation with the necessary information. The algorithm takes a parallel task of fork-join model τ_i as input with k_i segments and running on n_i , the output are a master string τ_i^{master} which will be assigned to a processor exclusively, and a group of constrained-deadline tasks τ_i^{cd} ,

which are the parallel segments that will be scheduled by the scheduling algorithm FBB-FFD.

As shown in Algorithm 3.1, segment stretch transformation algorithm is performed by the following steps :

- if the maximum execution length of the parallel task C_i 3.4 is less or equal to its deadline, then all the parallel segments of the task can fit in the slack of the master string S_i , and the parallel task can be fully stretched into a sequential one. In this case, we can apply the same transformation of task stretch.
- If not, we calculate the value of the slack time of the master string of the parallel task τ_i :

$$S_i = D_i - \sum_{j=1}^{\frac{k}{2}} (\tau_i^{2j-1} + \tau_i^{2j,1}) \quad (3.7)$$

- As in task stretch transformation, we calculate the number of parallel segments from each parallel region to be executed within the master string 3.5.
- For the remaining of the slack time S'_i , it can be calculated by the following equation :

$$S'_i = S_i - (\lfloor f_i \rfloor * \sum_{j=1}^{\frac{k-1}{2}} \tau_i^{2j,1}) \quad (3.8)$$

Instead of filling this space with equally-divided parts of parallel segments from each parallel region, which is the cause of migration, we will verify all the remaining parallel segments if any can fit inside this space.

- Check if the master string is fully stretched, according to the following condition :

$$\begin{aligned} C_i^{stretched} &= \sum_{j=1}^{\frac{k}{2}} (\tau_i^{2j-1} + \tau_i^{2j,1}) \\ C_i^{stretched} &== D_i \end{aligned} \quad (3.9)$$

If the condition is true, then we assign a processor exclusively to the master string, if not, then we add the master string to the group of the remaining parallel segments and use FBB-FFD to assign processors to all the tasks in the taskset.

Example :

We will apply the segment stretch transformation on the same previous taskset 3.3. In Figure 3.4 we show the result of applying task stretch transformation on τ_1 . We notice that segment $P_1^{2,3}$ and $P_1^{4,3}$ have to be executed on 2 processors. But in segment stretch transformation and as shown in Figure 3.5, the master string is only filled by complete parallel segments. Even though the master string is not fully stretched (there still 1 unit of time not used before the deadline), the parallel segment $P_1^{4,3}$ will not be used.

In segment stretch transformation we keep using the same equation to calculate the number of parallel segments to fill the master string $\lfloor f_i \rfloor$ (equation 3.5), but in task stretch transformation, the value $f_i - \lfloor f_i \rfloor$ which represents the rest of the slack time, it will be filled by parts of parallel segments from each parallel region, which causes job migrations. But in SST, and since we don't allow migration, our algorithm will fill this value of time with complete parallel segments, as shown in Figure 3.5, where $\tau_1^{2,3}$ is scheduled within the master string leaving only $\tau_1^{4,3}$ as a parallel segment and will be scheduled using FBB-FDD.

By using the segment stretch transformation, the taskset remained schedulable, the laxity of the master string is increased, as well as of the laxity of the parallel segments, and we eliminated the job migrations in the system as well.

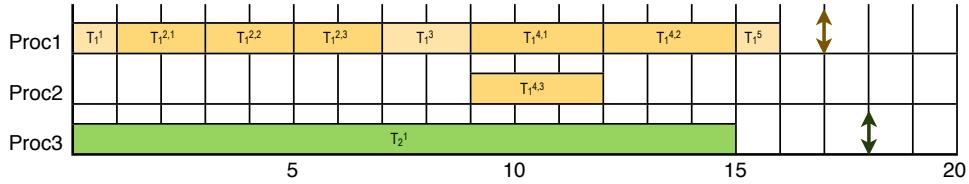


FIGURE 3.5 – Segment stretch transformation example.

This transformation has changed the model type from being rigid into moldable, since the number of processors is assigned by the scheduler after creating the master string. And from a practical implementation point of view, the segment stretch transformation can be fully implemented on a standard Linux RT kernel with no special extensions or patches added, and by only using an ordinary function like *sched_set_affinity()*, each segment of the parallel task can be assigned to a specific processor according to the scheduling results of any partitioning algorithm (*e.g.* FBB-FFD).

The Segment stretch transformation algorithm with the results and the analysis was presented in the 23rd Euromicro Conference on Real-Time Systems 2011 "ECRTS'11" - Work-in-Progress session, which was in Porto, Portugal between the 6th-8th of July/2011. And the paper will be published in the review of the conference [9].

3.3 Analysis and results

In this section, we provide a practical analysis for the performance of the algorithm Segment Stretch Transformation, in comparison with the task stretch transformation, by running an extensive simulation of dataset of almost 400,000 tasksets, each of 16 parallel real-time tasks of fork-join model, over an eight-core machine. This simulation helps us to show the results of scheduling algorithms of parallel real-time tasks of fork-join model using FBB-FFD without any transformation, and then by applying both transformations.

3.3.1 Simulator : *rtmsim*

rtmsim simulator is a real-time multiprocessor simulator, developed by the Algorithms team of the *Université Paris-Est Marne-la-Vallée, France*, and mainly by M.Frédéric Faubertea, who is a PhD student at *LIGM* under the supervision of M.Serge Midonnet. This simulation software helps analysing the performance of real-time scheduling algorithms by choosing a pre-coded approach to run in an extensive simulation.

The simulator is an open source software written in Java programming language, the reason for choosing java is that it is a popular programming language, well-known and used by many programmers and researchers, that will increase the chances of using the simulator, since the researchers will be able to implement their own scheduling algorithms and integrate them into the simulator. The source code and documentation for *rtmsim* are available at this address [<http://igm.univ-mlv.fr/AlgoTR/rtmsim/>].

The first version of the simulator *rtmsim* was already launched when I started working on the problematic of parallelism, and till now it is under active development. The simulator can generate datasets of tasks with dynamic properties that determine the model of the tasks(either dependent, parallel, ...), the number of processors of the system, certain constraints on the model (implicit, constraint or arbitrary deadlines), and other properties. The generated tasks are saved into .xml files, which gives the possibility to use the same dataset by many algorithms for comparison reasons.

In order to compare the performance of the transformation algorithms (task stretch and segment stretch), new classes had to be added to the simulator. And since there is several types of real-time tasks, like periodic, sporadic, sequential and parallel, We created a common interface called *schedulable*, from which all task types extends, and by that, several types can share same properties and general fields. Figure 3.6 shows the different classes of real-time tasks in the simulator and their relations, for example,

Algorithm 3.1 Algorithm : SegmentStretch

Input : $\tau_i : (\tau_i^{1,1}, (\tau_i^{2,1}, \tau_i^{2,2}, \dots, \tau_i^{2,n_i}), \dots, \tau_i^{k_i,1})$
Output : $(\tau_i^{master}, \{\tau_i^{cd}\})$

$\tau_i^{master} \leftarrow ()$
 $\{\tau_i^{cd}\} \leftarrow \{ \}$

if $C_i \leq D_i$ **then**

Convert the task into sequential, create a fully stretched master string
Apply the Stretch transformation

else

$f_i = \frac{S_i}{\sum_{s=1}^{\frac{k-1}{2}} C_i^{2s}}$
 $q_i \leftarrow (n_i - \lfloor f_i \rfloor)$
for $s = 1 \rightarrow \frac{k-1}{2}$ **do**
 $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2k-1,1}$
 for $p = 1 \rightarrow n_i$ **do**
 if $p = 1 \parallel p > q_i$ **then**
 Add segment to the master string
 $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2k,p}$
 else
 if $C_i^{2k,p} \leq S_i$ **then**
 Execute segment on the master string if it fits in the slack
 $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2k,p}$
 $S_i = C_i^{2k,p}$
 else
 Create a new parallel thread
 $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i^{2k,p}$
 end if
 end if
 end for
end for
 $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{k_i,1}$

if $C_i^{master} \neq D_i$ **then**

Not full master string, should be scheduled using FBB-FFD
 $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i^{master}$
 $\tau_i^{master} \leftarrow ()$

end if

end if

return $\tau_i^{master}, \{\tau_i^{cd}\}$

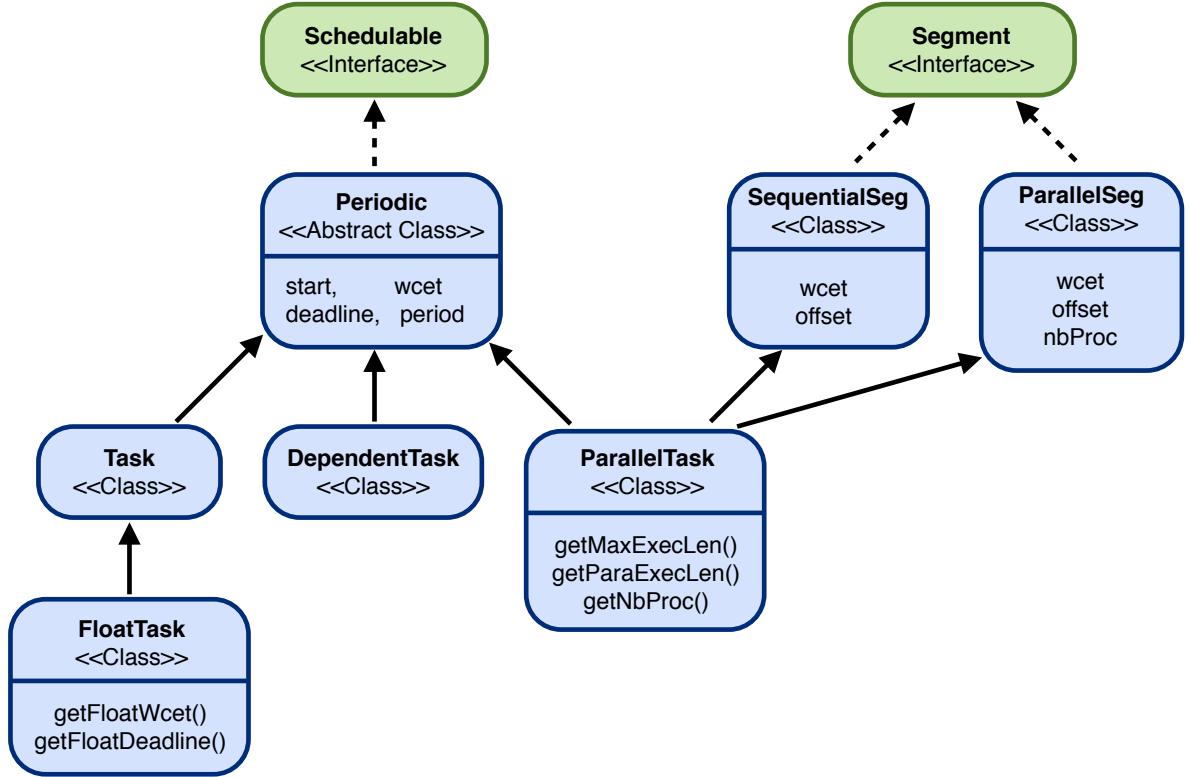


FIGURE 3.6 – Class Model for Task structure in *rtmsim*

parallel, float and dependent (sequential) tasks all share a common class "periodic", from which they all get the periodic real-time properties. Appendix A contains the source codes of the classes written during the internship.

The class of parallel task of fork-join model has a special structure, because it has certain model which includes segments, both parallel and sequential, and it has certain functions needed for the implementation of different algorithms, as shown in the figure. Float tasks are created specially to be used in the task stretch transformation algorithm, because the parallel segments will be partitioned so as to fill the master string when migration happens, and according to that, the resulting segments will have execution times and local deadlines in the float format, because of the used calculations of segments partitioning, and as a result, tasks with float execution times and deadlines are generated.

Another important feature in the *rtmsim* is that it generates datasets of tasks, using a task generator class, in which the arguments can be specified automatically by the user, and the output tasksets are saved in xml files, in order to be used later, and the same taskset will be used for all the algorithms to be compared. The output of the compared algorithms are stored as well in .xml files.

3.3.2 Analysis

Based on a simulation protocol of multiprocessor systems proposed by Davis *et al.* [10], which uses UUniFast as task-generation algorithm [11], we randomly generated 10,000 tasksets per processor's utilization varying from 0.025 to 0.975, each taskset of 16 parallel tasks, which makes it a total of 390,000 tasksets. FBB-FFD is used as scheduling algorithm.

We started the analysis by creating a dataset of periodic parallel tasks of the fork-join model on a system of 4 homogeneous processors, and by using FBB-FFD directly to schedule this dataset, we obtained the results shown in Figure 3.7(a) (the curve with the triangular points). Obviously, FBB-FFD failed to schedule the dataset beyond the processors' utilization of 0.1. This can be explained by knowing that

FBB-FFD is using a specific condition described earlier in 3.6, which says if the task to be scheduled has both an execution time and a deadline of the same value, then the condition will definitely fail if the processor contains other tasks to be executed on. And since parallel segments in the fork-join model always have execution times equal to their local deadlines (Figure 3.3), then each parallel segment will need to be executed exclusively on a single processor, causing the taskset to be non-scheduled.

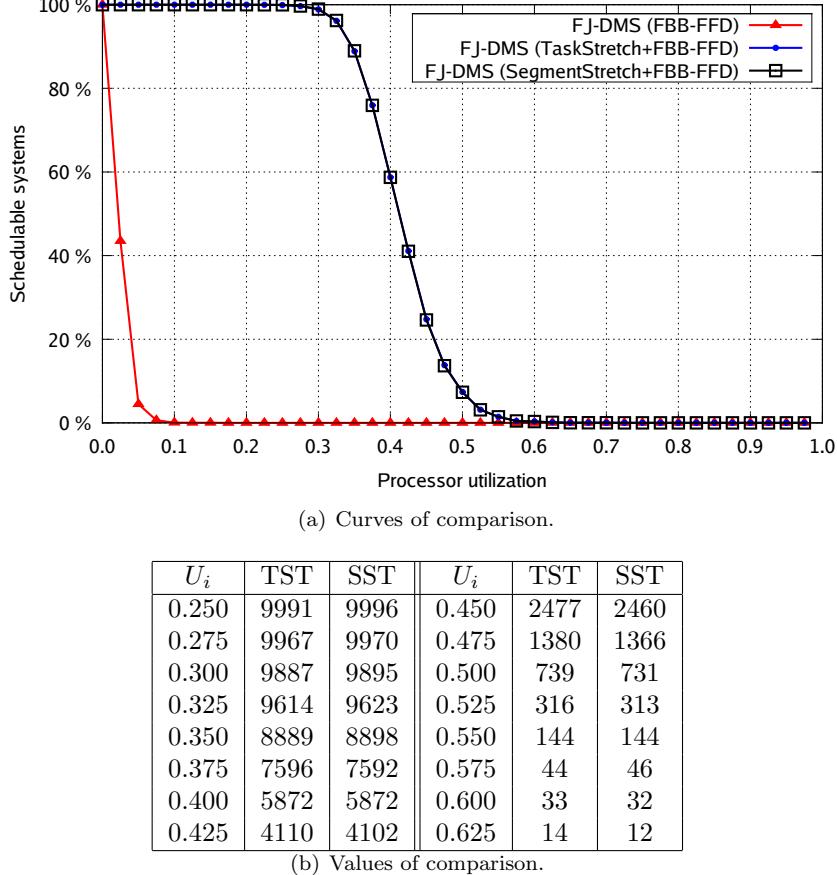


FIGURE 3.7 – Simulation results.

Therefore and by looking at the characteristics of the parallel segments, we notice that they have offsets (since the fork event does not happen at the activation instant of the parallel task, but after the execution of previous segments), which means that they will not all be scheduled at the same time. By using a suitable type of scheduling algorithm that can handle offsets we might be able to enhance the results of the simulation. FDD-RTA (First Fit Decreasing-Response Time Analysis) could be a good choice.

A second analysis was performed to compare task and segment stretch transformation algorithms, by using the same model of extensive simulation described previously. The result of the simulation is shown in Figure 3.7(a), where task stretch transformation is the curve with the round points, and segment stretch is the curve with square points. As we can see, there is no noticeable difference. There is a slight difference between these 2 algorithms as represented in Figure 3.7(b). This table shows the number of tasksets each algorithm succeeded to schedule for each utilization of the processor. Another remark to be noticed from these numbers, is that both algorithms task and segment stretch transformation are incomparable, from the results we can see that the algorithm SST has more schedulable taskset than task stretch transformation in the period [0.250, 0.350], and after that the results are inverted, and task stretch now has more schedulable tasks than segment stretch transformation. This means that the modifications we made on the algorithm task stretch transformation not only kept the original algorithm's performance, but also added a practical implementation advantage.

U_i	TST	SST	U_i	TST	SST
0.225	9996	9997	0.450	2434	2460
0.250	9993	9996	0.475	1352	1366
0.275	9969	9970	0.500	711	731
0.300	9892	9895	0.525	302	313
0.325	9616	9623	0.550	139	144
0.350	8873	8898	0.575	42	46
0.375	7543	7592	0.600	30	32
0.400	5830	5872	0.625	13	12
0.425	4071	4102	0.650	2	2

FIGURE 3.8 – Simulation data with migration cost.

The main difference between the 2 transformation algorithms, task stretch and segment stretch, is that we eliminated the use of migration in segment stretch, while in task stretch if master string is not fully stretched, then one segment from each parallel region will migrate to the processor of the master string, which will add migration cost by the number of parallel regions in each task. In order to analyse this remark, we included the migration cost to the WCET of each parallel segment before being scheduled by FBB-FFD whenever migration happens, and for demonstration purposes we chose a migration cost of 5 units of time (where the maximum period of parallel task is 1000 units of time). The results of the simulation is provided by the table of data 3.8, which represents the number of successfully scheduled tasksets over both transformation, and over a period of processor’s utilization from 0.025 to 0.975 with steps of 0.025. In the table, only values from 0.225 to 0.650 are shown in the table because the data outside this region is the same in both algorithms, and there is nothing to compare. As we can see in the table, and after applying the migration costs, segment stretch transformation is always slightly better than task stretch transformation.

Chapitre 4

Work in Progress

4.1 Predictability of fork-join model

Real-time systems are used usually for critical applications, where respecting time constraints is as important as executing the tasks correctly. Therefore reliability and performance are critical characteristics in real-time systems. Usually, Real-time tasks are represented by the worst case execution time among other temporal constraints, which it is the maximum time the task will need in order to finish its execution correctly, but that does not mean the task has to reach this value in normal cases of execution, because of that a real-time system has to be predictable.

Definition :

A scheduling system is said to be predictable, if tasks in the taskset stay schedulable when they execute for a time less than the worst case execution time (WCET), and it is sufficient to provide a non-predictable example of taskset scheduled under a certain scheduling algorithm to say this model of tasks are not predictable under this scheduling algorithm. We studied the predictability of parallel real-time tasks of fork-join model under Scheduling algorithms like deadline monotonic DM, rate monotonic RM and earliest deadline first EDF, unfortunately, parallel tasks of fork-join model have been proved to be non-predictable systems.

Proof :

Figure 4.1 shows periodic constrained-deadline parallel fork-join taskset, executes on 2 processors :

$$\begin{aligned}\tau_1 &= ((2, 1, 1), 2, 6, 6) \\ \tau_2 &= ((2), 1, 2, 3) \\ \text{Priority}(\tau_1) &> \text{Priority}(\tau_2)\end{aligned}$$

As shown in Figure 4.1(a), the taskset is schedulable on the 2 processors, and all the segments of both tasks τ_1 and τ_2 are executing up to their worst-case execution times WCETs successfully, and without missing any deadline. But when the segment τ_1^1 finishes its execution 1 unit of time earlier instead of 2, the parallel segments $\tau_1^{2,1}$ and $\tau_1^{2,2}$ are activated earlier at $t = 1$, pre-empting and suspending the execution of τ_2^1 for 1 unit of time, which is enough for this segment to miss its deadline, as shown in Figure 4.1(b). As a result, this parallel taskset is not schedulable nor predictable.

Usually, non-predictability happens in tasksets when tasks with higher priorities have no pre-emption effects on lower-priority tasks in worst case execution scenarios, but in normal scenarios, when this higher task executes for less than its WCET, it has a blocking effect on the other tasks and pre-empt them.

Parallel real-time tasks of fork-join model are not predictable using fixed-task priority assignment like deadline monotonic "DM", and neither by using fixed-job priority assignment like Earliest deadline first "EDF", as the following example shows 4.2, the tasks τ_1 and τ_2 are not predictable when they have fixed job priority, and since $D_2 < D_1$ then $P_2 > P_1$ according to EDF (and also according to DM), causing it

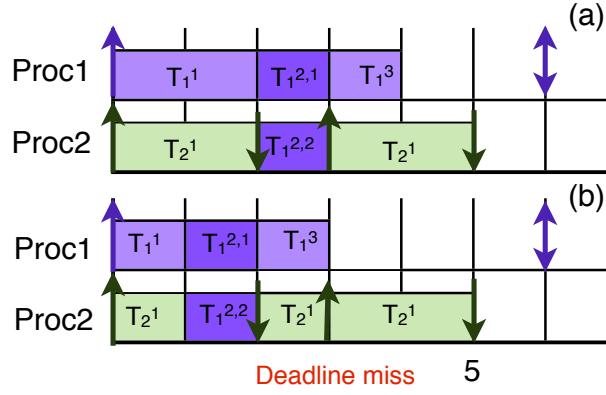


FIGURE 4.1 – Non-predictable parallel taskset of Fork-join model.

to pre-empt the execution of segment $\tau_1^{2,2}$ when segment τ_2^1 executes for less than its worst-case execution time. And since we provided an example showing the unpredictability of EDF parallel tasks of fork-join model, then this priority assignment is not predictable for this type of parallel tasks.

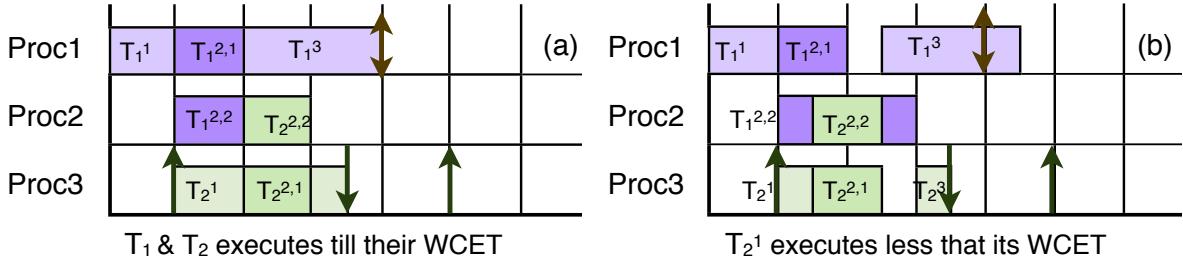


FIGURE 4.2 – Non-predictable parallel taskset of Fork-join model under EDF priority assignment.

As a perspective to this internship, we are working on providing a predictable schedule algorithm for parallel fork-join model which is predictable, which includes proposing an adapted priority assignment for this model of parallelism.

4.2 Response time analysis

Another perspective to this research internship is to work on a specific analysis of response time of the parallel real-time tasks of fork-join model. In any system, response time can be defined as the time this system takes to react to a given input, but in real-time task, the response time is defined as the time a task takes from the activation instant (the task is ready to start execution), till it finishes its execution, which can be more than the WCET, because WCET is the maximum execution time of a task without interference.

For sequential real-time tasks executing on uniprocessor systems, the analysis of response time is widely used and studied, and scheduling algorithms based on response time analysis aim to reduce its values in the tasks which is reflected on the performance of the tasks in general, since it is an indication to the speed of the execution.

This analysis can be quite useful for the scheduling theories of parallel real-time tasks, and as far as we know, it has not been studied before.

Chapitre 5

Conclusion

Applications based on real-time systems are usually described as critical applications, in which deadlines must be met regardless the load of the system. The importance of these applications gave real-time systems the necessary importance to be studied and analysed over the years, and the researchers in this domain are racing to propose new solutions and theories in order to keep up with the development of systems and technologies either it is hardware or software.

Parallel programming model is developed in order to be used on multiprocessor systems which have been widely manufactures nowadays, and the idea of integrating parallelism in real-time systems is a practical method to get the most advantage of multiprocessor systems and increase the performance of real-time calculations, although a lot of analysis have to be proposed for this specific model of parallelism.

In this work, we studied the fork-join model, which is one of many parallel programming models, and a specific transformation we found in literature. We also proposed our own transformation in order to improve some aspects in the scheduling theory. The performance and the comparison of both algorithms of transformation are provided in the analysis part with the aid of a real-time simulator developed by the real-time team in the research laboratory. Finally we described in details the advantages and disadvantages of each transformation, and we showed the possible perspectives of this specific subject of research.

Studying a specific model of parallel real-time tasks which is the fork-join model is an example of the real-time analysis that can be done on parallel tasks, and from these results, the research and the analysis can be expanded to study other models of parallel real-time tasks, and work on a general model of parallelism in the future.

This work is a starting point of a PhD subject, in which i will continue working on it for more 3 years at the same laboratory of research in the university Paris-est Marne-la-vallée under the supervision of M.Serge MIDONNET.

Bibliographie

- [1] “Ecrtts11.” [Online]. Available : <http://www.cister.isep.ipp.pt/ecrtts11/>
- [2] J. Goossens and V. Berten, “Gang ftp scheduling of periodic and parallel rigid real-time tasks,” in *Proc. of RTNS*, 2010, pp. 189–196.
- [3] S. Kato and Y. Ishikawa, “Gang edf scheduling of parallel task systems,” in *Proc. of RTSS*, 2009, pp. 459–468.
- [4] S. Collette, L. Cucu, and J. Goossens, “Integrating job parallelism in real-time scheduling theory,” *Information Processing Letters*, vol. 106, pp. 180–187, 2008.
- [5] K. Lakshmanan, S. Kato, and R. (Raj) Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in *Proc. of RTSS*, 2010, pp. 259–268.
- [6] “Openmp.” [Online]. Available : <http://www.openmp.org>
- [7] N. Fisher, S. Baruah, and T. P. Baker, “The partitioned scheduling of sporadic tasks according to static-priorities,” in *Proc. of ECRTS*, 2006, pp. 118–127.
- [8] S. Oikawa and R. Rajkumar, “Portable rk : A portable resource kernel for guaranteed and enforced timing behavior,” in *Proc. of RTAS*, 1999, p. 111.
- [9] F. Fauberteau, S. Midonnet, and M. Qamhieh, “Partitioned scheduling of parallel real-time tasks on multiprocessor systems,” in *23rd Euromicro Conference on Real-Time Systems (ECRTS’11). WIP*, 2011.
- [10] R. I. Davis and A. Burns, “Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, 2010.
- [11] E. Bini and G. C. Buttazzo, “Biassing effects in schedulability measures,” in *Proceedings of the 16th Euromicro Conference on Real-time Systems (ECRTS)*. IEEE Computer Society, 2004, pp. 196–203.

Annexe A

Simulator code

A.1 ParallelTask.java

```
1 // Copyright (c) 2010, Frederic Fauberteau
2 // All rights reserved.
3
4 package fr.upr.rtmsim.task;
5
6 import java.util.ArrayList;
7 import java.util.Iterator;
8 import java.util.List;
9
10 import org.w3c.dom.Document;
11 import org.w3c.dom.Element;
12
13 /**
14 * @author Frederic Fauberteau
15 * @author Manar Qamhieh
16 */
17 public class ParallelTask extends Periodic {
18
19     private ArrayList<Segment> segments = new ArrayList<Segment>();
20
21     public ParallelTask(long start, List<Long> wcets, long deadline, long period,
22                         int nbProc) {
23         super(start, wcetSum(wcets), deadline, period);
24         if (wcets.size() % 2 != 1) {
25             throw new IllegalArgumentException(
26                 "There must have an odd number of segments.");
27         }
28         long offset = 0;
29         for (int i = 0; i < wcets.size(); i++) {
30             long wcet = wcets.get(i);
31             if (wcet == 0) {
32                 throw new IllegalArgumentException(
33                     "Length of a segment must be at least 1.");
34             }
35             if (i % 2 == 0) {
36                 segments.add(new SequentialSegment(wcets.get(i), offset));
37             } else {
38                 segments.add(new ParallelSegment(wcets.get(i), nbProc, offset));
39             }
40             offset += wcets.get(i);
41         }
42     }
43
44     private static long wcetSum(List<Long> wcets) {
45         long sum = 0;
46
47         for (long wcet : wcets) {
48             sum += wcet;
49         }
50
51         return sum;
52     }
53 }
```

```

51     }
52
53     public long getMaximumExecutionLength() {
54         long sum = 0;
55         for (Segment segment : segments) {
56             sum += segment.getMaximumExecutionLength();
57         }
58         return sum;
59     }
60
61     public long getParallelExecutionLength() {
62         long sum = 0;
63         for (Segment segment : segments) {
64             sum += segment.getParallelExecutionLength();
65         }
66         return sum;
67     }
68
69     public int getNbProcessor() {
70         if (segments.size() == 1) {
71             return segments.get(0).getNbProcessor();
72         }
73         return segments.get(1).getNbProcessor();
74     }
75
76     public Iterable<ParallelSegment> getParallelSegments() {
77         return new Iterable<ParallelTask.ParallelSegment>() {
78
79             @Override
80             public Iterator<ParallelSegment> iterator() {
81                 return new Iterator<ParallelSegment>() {
82
83                     private int index = 1;
84
85                     @Override
86                     public boolean hasNext() {
87                         if (index < segments.size()) {
88                             return true;
89                         }
90                         return false;
91                     }
92
93                     @Override
94                     public ParallelSegment next() {
95                         int tmp = index;
96                         index += 2;
97                         return (ParallelSegment) segments.get(tmp);
98                     }
99
100                    @Override
101                    public void remove() {
102                        throw new UnsupportedOperationException("Operation not supported");
103                    }
104                };
105            };
106        };
107    }
108
109    public Schedulable growUp(long executionTime) {
110        ArrayList<Long> wcets = new ArrayList<Long>();
111        for (int i = 0; i < segments.size() - 1; i++) {
112            wcets.add(segments.get(i).getWcet());
113        }
114        wcets.add(segments.get(segments.size() - 1).getWcet() + executionTime);
115        return new ParallelTask(getStart(), wcets, getDeadline(), getPeriod(),
116                               getNbProcessor());
117    }
118
119    public Element toXML(Document document) {
120        Element elmt = document.createElement("task");
121

```

```

122     elmt.setAttribute("type", "parallel");
123     elmt.setAttribute("nbproc", Integer.toString(getNbProcessor()));
124     Element attr = document.createElement("start");
125     attr.setAttribute("value", Long.toString(getStart()));
126     elmt.appendChild(attr);
127     attr = document.createElement("wcet");
128     attr.setAttribute("nbsegment", Integer.toString(segments.size()));
129     for (Segment seg : segments) {
130         attr.appendChild(seg.toXML(document));
131     }
132     elmt.appendChild(attr);
133     attr = document.createElement("deadline");
134     attr.setAttribute("value", Long.toString(getDeadline()));
135     elmt.appendChild(attr);
136     attr = document.createElement("period");
137     attr.setAttribute("value", Long.toString(getPeriod()));
138     elmt.appendChild(attr);
139     return elmt;
140 }
141
142 @Override
143 public String toString() {
144     StringBuilder sb = new StringBuilder();
145     sb.append('(').append(getStart()).append(",(");
146     for (Segment segment : segments) {
147         sb.append(segment).append(',');
148     }
149     sb.setLength(sb.length() - 1);
150     sb.append("),").append(getDeadline()).append(',').append(getPeriod())
151         .append(')');
152     return sb.toString();
153 }
154
155 public interface Segment {
156
157     public long getWcet();
158
159     public long getMaximumExecutionLength();
160
161     public long getParallelExecutionLength();
162
163     public int getNbProcessor();
164
165     public Element toXML(Document document);
166
167     public long getOffset();
168 }
169
170 public class ParallelSegment implements Segment {
171
172     private final long wcet;
173
174     private final int nbProc;
175
176     private final long offset;
177
178     ParallelSegment(long wcet, int nbProc, long offset) {
179         this.wcet = wcet;
180         this.nbProc = nbProc;
181         this.offset = offset;
182     }
183
184     public long getWcet() {
185         return wcet;
186     }
187
188     public long getMaximumExecutionLength() {
189         return getNbProcessor() * getWcet();
190     }
191
192 }
```

```

193     public long getParallelExecutionLength() {
194         return getWcet();
195     }
196
197     public int getNbProcessor() {
198         return nbProc;
199     }
200
201     public Element toXML(Document document) {
202         Element elmt = document.createElement("segment");
203         elmt.setAttribute("type", "parallel");
204         elmt.setAttribute("length", Long.toString(getWcet()));
205         return elmt;
206     }
207
208     @Override
209     public long getOffset() {
210         return offset;
211     }
212
213     @Override
214     public String toString() {
215         return "{" + getWcet() + ": " + getNbProcessor() + ":" + getOffset() + '}';
216     }
217 }
218
219 public class SequentialSegment implements Segment {
220
221     private final long wcet;
222
223     private final long offset;
224
225     SequentialSegment(long wcet, long offset) {
226         this.wcet = wcet;
227         this.offset = offset;
228     }
229
230     public long getWcet() {
231         return wcet;
232     }
233
234     public long getMaximumExecutionLength() {
235         return getWcet();
236     }
237
238     public long getParallelExecutionLength() {
239         return 0;
240     }
241
242     public int getNbProcessor() {
243         return 1;
244     }
245
246     public Element toXML(Document document) {
247         Element elmt = document.createElement("segment");
248         elmt.setAttribute("type", "sequential");
249         elmt.setAttribute("length", Long.toString(getWcet()));
250         return elmt;
251     }
252
253     @Override
254     public long getOffset() {
255         return offset;
256     }
257
258     @Override
259     public String toString() {
260         return "{" + getWcet() + ": " + getOffset() + "}";
261     }
262 }
263

```

264
265 }

A.2 FloatTask.java

```
1 // Copyright (c) 2010, Frederic Fauberteau
2 // All rights reserved.
3
4 package fr.upesrtmsim.task;
5
6 /**
7 * @author Frederic Fauberteau
8 * @author Manar Qamhieh
9 */
10 public class FloatTask extends Task {
11
12     //private double start;
13
14     private double wcet;
15
16     private double deadline;
17
18     //private double period;
19
20     public FloatTask(double start, double wcet, double deadline, double period) {
21         super((long)start, (long)wcet, (long)deadline, (long)period);
22         //this.start = start;
23         this.wcet = wcet;
24         this.deadline = deadline;
25         //this.period = period;
26     }
27
28     @Override
29     public double getFloatDeadline(){
30         return deadline;
31     }
32
33     @Override
34     public double getFloatWcet(){
35         return wcet;
36     }
37
38     @Override
39     public String toString() {
40         return "(" + getStart() + "," + getFloatWcet() + "," + getFloatDeadline() + ",
41             " + getPeriod() + ")";
42     }
43
44 }
```

A.3 BasicFjDms.java

```

1 // Copyright (c) 2010, Frederic Fauberteau
2 // All rights reserved.
3
4 package fr.upc.rtmsim.partitioning;
5
6 import fr.upc.rtmsim.task.ParallelTask;
7 import fr.upc.rtmsim.task.Task;
8 import fr.upc.rtmsim.task.Taskset;
9 import fr.upc.rtmsim.task.ParallelTask.ParallelSegment;
10
11 /**
12 * <p>
13 * FjDmsSegmentStretch transforms parallel tasks of fork-join structure into a
14 * set of tasks accepted by the partitioning algorithm FbbFfd.
15 * </p>
16 *
17 * @author Manar Qamhieh
18 */
19 public class BasicFjDms implements Partitioning<ParallelTask, Task>{
20
21     @Override
22     public Partition<Task> partition(Taskset<ParallelTask> taskset,
23             int nbProc) {
24         Partition<Task> partition = new Partition<Task>(nbProc);
25         Taskset<Task> Fdd = new Taskset<Task>();
26         for (ParallelTask parallelTask : taskset) {
27             Taskset<Task> segTaskset = getSegments(parallelTask);
28             for (Task task : segTaskset) {
29                 Fdd.add(task);
30             }
31         }
32
33         if (Fdd.size() != 0) {
34
35             FbbFfd<Task> fbbFfd = new FbbFfd<Task>();
36             partition = fbbFfd.partition(Fdd, nbProc);
37         }
38         return partition;
39     }
40
41     private Taskset<Task> getSegments(ParallelTask parallelTask) {
42
43         Taskset<Task> outputTaskset = new Taskset<Task>();
44         long deadline = parallelTask.getDeadline();
45         // main thread contains both sequential and parallel segments
46         Task main = new Task(parallelTask.getStart(), parallelTask.getWcet(),
47             deadline, parallelTask.getPeriod());
48         outputTaskset.add(main);
49
50         // parallel threads
51         for (ParallelSegment segment : parallelTask.getParallelSegments()) {
52             for(int i=1; i<segment.getNbProcessor(); i++){
53                 outputTaskset.add(new Task(segment.getOffset(), segment.getWcet(),
54                     segment
55                     .getWcet(), deadline));
56             }
57         }
58     }
59 }
```

A.4 FjDms.java

```

1 // Copyright (c) 2010, Frederic Fauberteau
2 // All rights reserved.
3
4 package fr.upr.rtmsim.partitioning;
5
6 import java.util.Iterator;
7
8 import fr.upr.rtmsim.task.FloatTask;
9 import fr.upr.rtmsim.task.ParallelTask;
10 import fr.upr.rtmsim.task.Taskset;
11 import fr.upr.rtmsim.task.ParallelTask.ParallelSegment;
12
13 /**
14 * <p>
15 * FjDms transforms parallel tasks of fork-join structure into a set of float
16 * tasks by using stretchTask function where migration is allowed (description
17 * can be found in the paper mentioned below), then FbbFfd partitioning
18 * algorithm is used to schedule these tasks.
19 * </p>
20 * <p>
21 * This algorithm has been proposed in:<br/>
22 * <tt>
23 * Lakshmanan, Karthik and Kato, Shinpei and (Raj) Rajkumar, Raghunathan<br/>
24 * Scheduling Parallel Real-Time Tasks on Multi-core Processors<br/>
25 * Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS), 259-268, 2010<br/>
26 * </tt>
27 * </p>
28 *
29 * @author Manar Qamhieh
30 */
31 public class FjDms implements Partitioning<ParallelTask, FloatTask>{
32
33     private final static long MIGRATION_COST = 5;
34
35     @Override
36     public Partition<FloatTask> partition(Taskset<ParallelTask> taskset,
37             int nbProc) {
38         Partition<FloatTask> partition = new Partition<FloatTask>(nbProc);
39         Taskset<FloatTask> stretchedTaskset, Fdd;
40         Taskset<FloatTask> fullMaster = new Taskset<FloatTask>();
41
42         Fdd = new Taskset<FloatTask>();
43         for (ParallelTask parallelTask : taskset) {
44             stretchedTaskset = stretchTask(parallelTask);
45
46             for (FloatTask task : stretchedTaskset) {
47                 if (task.getPeriod() == task.getFloatWcet()) {
48                     fullMaster.add(task);
49                     if (nbProc == 0) {
50                         return null;
51                     }
52                     nbProc--;
53                 } else
54                     Fdd.add(task);
55             }
56         }
57
58         Iterator<Processor> it = partition.iterator();
59         if (Fdd.size() != 0) {
60             FbbFfd<FloatTask> fbbFfd = new FbbFfd<FloatTask>();
61             Partition<FloatTask> partFfd = fbbFfd.partition(Fdd, nbProc);
62             if (partFfd == null) {
63                 return null;
64             }
65             for (Processor processor : partFfd) {
66                 partition.addAll(it.next(), partFfd.get(processor));
67             }
68         }
69     }

```

```

70         for (FloatTask task : fullMaster) {
71             partition.add(it.next(), task);
72         }
73     }
74
75     return partition;
76 }
77
78 private Taskset<FloatTask> stretchTask(ParallelTask task){
79
80     Taskset<FloatTask> outputTasks = new Taskset<FloatTask>();
81     FloatTask masterTask = null;
82
83     double Di = task.getDeadline();
84     double Ti = task.getPeriod();
85     double CiMin = task.getWcet();
86     double CiMax = task.getMaximumExecutionLength();
87     double fi;           //the capacity of the master string
88     int    qi;           //the total number of parallel threads in each parallel
89     segment
90     int    mi = task.getNbProcessor();
91     double wcetSum = CiMin;
92
93
94     if(CiMax <= Ti){
95         //Convert parallel task into sequential
96         wcetSum = CiMax;
97     }else{
98         //Stretch the task to its deadline
99         fi = (Di - CiMin) / task.getParallelExecutionLength();
100        qi = (int) (mi - Math.floor(fi));
101
102        for (ParallelSegment segment : task.getParallelSegments()) {
103            //starting the loop from k = 2, since the first parallel segment is
104            //already included in the master string
105            for(int k = 2; k <= mi; k++){
106                if( k > qi){
107                    //segments will be executed on the master string
108                    wcetSum += segment.getWcet();
109                    //the idea is to use the master string 100%, so at the end
110                    //wcetSum will be equal to the deadline.
111                }else if (k < qi){
112                    //create new parallel thread
113                    double deadline = (1+fi) * segment.getWcet();
114                    outputTasks.add(new FloatTask(segment.getOffset(), segment.
115                        getWcet(), deadline, Ti));
116                }else{
117                    //split among a parallel thread and the master string
118                    double tmp = fi-Math.floor(fi);
119                    wcetSum += tmp * segment.getWcet();
120                    double deadline = 1+Math.floor(fi);
121                    double fwct = ((1-tmp)*segment.getWcet());
122                    // add migration cost.
123                    fwct += MIGRATION_COST;
124                    outputTasks.add(new FloatTask(segment.getOffset(), fwct ,
125                        deadline*segment.getWcet(), Ti));
126                }
127            }
128        }
129    }
130
131    masterTask = new FloatTask(0, wcetSum, Di, Ti);
132    outputTasks.add(masterTask);
133
134    return outputTasks;
135 }
136
137 }
138
139 }
140
141 }
```

A.5 FjDmsSegmentStretch.java

```

1 // Copyright (c) 2010, Frederic Fauberteau
2 // All rights reserved.
3
4 package fr.upr.rtmsim.partitioning;
5
6 import java.util.Iterator;
7
8 import fr.upr.rtmsim.task.ParallelTask;
9 import fr.upr.rtmsim.task.Task;
10 import fr.upr.rtmsim.task.Taskset;
11 import fr.upr.rtmsim.task.ParallelTask.ParallelSegment;
12
13 /**
14 * <p>
15 * FjDmsSegmentStretch transforms parallel tasks of fork-join structure into a set of tasks by using stretchSegment function, it has the same functionality as the class FjDms but migration is not allowed, then FbbFfd partitioning algorithm is used to schedule these tasks.
16 * </p>
17 *
18 * @author Manar Qamhieh
19 */
20
21 public class FjDmsSegmentStretch implements Partitioning<ParallelTask, Task>{
22
23     @Override
24     public Partition<Task> partition(Taskset<ParallelTask> taskset,
25             int nbProc) {
26         Partition<Task> partition = new Partition<Task>(nbProc);
27         Taskset<Task> stretchedTaskset, Fdd;
28         Taskset<Task> fullMaster = new Taskset<Task>();
29
30         Fdd = new Taskset<Task>();
31         for (ParallelTask parallelTask : taskset) {
32             stretchedTaskset = stretchSegment(parallelTask);
33
34             for (Task task : stretchedTaskset) {
35                 if (task.getPeriod() == task.getWcet()) {
36                     fullMaster.add(task);
37                     if(nbProc == 0){
38                         return null;
39                     }
40                     nbProc--;
41                 } else
42                     Fdd.add(task);
43             }
44         }
45
46         Iterator<Processor> it = partition.iterator();
47         if (Fdd.size() != 0) {
48             FbbFfd<Task> fbbFfd = new FbbFfd<Task>();
49             Partition<Task> partFfd = fbbFfd.partition(Fdd, nbProc);
50             if (partFfd == null) {
51                 return null;
52             }
53             for (Processor processor : partFfd) {
54                 partition.addAll(it.next(), partFfd.get(processor));
55             }
56         }
57         for (Task task : fullMaster) {
58             partition.add(it.next(), task);
59         }
60
61         return partition;
62     }
63
64     private Taskset<Task> stretchSegment(ParallelTask task){
65
66         Taskset<Task> outputTasks = new Taskset<Task>();
67
68
69

```

```

70     Task masterTask = null;
71
72     long Di = task.getDeadline();
73     long Ti = task.getPeriod();
74     long CiMin = task.getWcet();
75     long CiMax = task.getMaximumExecutionLength();
76     double fi;           //the capacity of the master string
77     int   qi;            //the total number of parallel threads in each parallel
78     segment
79     int   mi = task.getNbProcessor();
80     long wcetSum = CiMin;
81
82
83     if(CiMax <= Ti){
84         //Convert parallel task into sequential
85         wcetSum = CiMax;
86     }else{
87         //Stretch the task to its deadline
88         fi = (double)(Di - CiMin) / task.getParallelExecutionLength();
89         qi = (int) (mi - Math.floor(fi));
90         //to avoid fractions
91         double remSlack = (Di-CiMin - (Math.floor(fi)*task.
92             getParallelExecutionLength()));
93
94         for (ParallelSegment segment : task.getParallelSegments()) {
95             //starting the loop from k = 2, since the first parallel segment is
96             //already included in the master string
97             int segCount = 0;
98             for(int k = 2; k <= mi; k++){
99                 if( k > qi){
100                     //segments will be executed on the master string
101                     wcetSum += segment.getWcet();
102                     //the idea is to use the master string 100%, so at the end
103                     //wcetSum will be equal to the deadline.
104                 }else {
105                     // check if segment fits in the remaining Slack of the master
106                     // string
107                     if (segment.getWcet() <= remSlack){
108                         //execute segment on master string
109                         wcetSum += segment.getWcet();
110                         remSlack -= segment.getWcet();
111                         segCount++;
112                     }else{
113                         //create new parallel thread
114                         long tmp = (long) (Math.floor(fi) + segCount + 1);
115                         long deadline = tmp * segment.getWcet();
116                         outputTasks.add(new Task(segment.getOffset(), segment.
117                             getWcet(), deadline, Ti));
118                     }
119                 }
120             }
121         }
122     }
123 }
```

A.6 FbbFfd.java

```
1 // Copyright (c) 2010, Frederic Fauberteau
2 // All rights reserved.
3
4 package fr.upc.rtmsim.partitioning;
5
6 import fr.upc.rtmsim.partitioning.binpacking.OptBinPackingAlgorithm;
7 import fr.upc.rtmsim.partitioning.binpacking.DecreasingDeadline;
8 import fr.upc.rtmsim.partitioning.binpacking.FirstFit;
9 import fr.upc.rtmsim.priority.PriorityManagerKey;
10 import fr.upc.rtmsim.scheduling.DmRbf;
11 import fr.upc.rtmsim.task.Task;
12 import fr.upc.rtmsim.task.Taskset;
13
14 /**
15 * This class is the implementation of the algorithm
16 * <em>Fisher Baruah Baker First-Fit Decreasing</em> (FBB-FFD) based on
17 * <em>First-Fit Decreasing</em>. This algorithm is aimed to partition a set of
18 * independent sporadic tasks.
19 *
20 * <p>
21 * This algorithm has been proposed in:<br/>
22 * <tt>
23 * Fisher, Nathan Wayne and Baruah, Sanjoy K. and Baker, Theodore P.<br/>
24 * The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities<br/>
25 * Proceedings of the 18th Euromicro Conference on Real-time Systems (ECRTS), 118-127,
26 * 2006<br/>
27 * </tt>
28 * </p>
29 *
30 * @author Frederic Fauberteau
31 * @author Manar Qamhieh
32 */
33 public class FbbFfd<T extends Task> implements Partitioning<T, T> {
34
35     @Override
36     public Partition<T> partition(Taskset<T> taskset, int nbProc) {
37         OptBinPackingAlgorithm<T> algorithm = new OptBinPackingAlgorithm<T>(
38             new DecreasingDeadline(), new FirstFit(), PriorityManagerKey.DM, new DmRbf<T>());
39         return algorithm.partition(taskset, nbProc);
40     }
41 }
```