Étude et évaluation des politiques d'ordonnancement temps réel multiprocesseur

Maxime Chéramy

▶ **To cite this version:**

Maxime Chéramy. Étude et évaluation des politiques d'ordonnancement temps réel multiprocesseur. Systèmes embarqués. Institut National des Sciences Appliquées de Toulouse (INSA Toulouse), 2014. Français. NNT : . tel-01104953

HAL Id: tel-01104953
https://theses.hal.science/tel-01104953

Submitted on 19 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)*

Présentée et soutenue le *11/12/2014* **par :**

MAXIME CHÉRAMY

Étude et évaluation de politiques d'ordonnancement temps réel multiprocesseur

---

JURY

PASCAL RICHARD	Professeur des Universités, Université de Poitiers	Rapporteur
LILIANA CUCU-GROSJEAN	Chargée de recherche, HDR, INRIA Paris-Rocquencourt	Rapporteur
JEAN-CHARLES FABRE	Professeur des Universités, INP-ENSEEIHT	Examinateur
YVON TRINQUET	Professeur des Universités, Université de Nantes	Examinateur
ANNE-MARIE DÉPLANCHE	Maître de Conférences, Université de Nantes	Directeur
PIERRE-EMMANUEL HLADIK	Maître de Conférences, INSA de Toulouse	Directeur
LAURENT GEORGE	Professeur, ESIEE Paris / LIGM, Université Paris-Est	Invité

École doctorale et spécialité :
EDSYS : Informatique 4200018
Unité de Recherche :
LAAS-CNRS
Directeur(s) de Thèse :
Pierre-Emmanuel Hladik et *Anne-Marie Déplanche*
Rapporteurs :
Pascal Richard et *Liliana Cucu-Grosjean*

Remerciements

Je profite de l'occasion qui m'est donnée pour remercier les personnes qui ont eu un rôle durant l'élaboration de cette thèse. Cette thèse s'est déroulée au sein du LAAS-CNRS dans l'équipe VERTICS entre le 29 août 2011 et le 11 décembre 2014.

Mes premiers remerciements vont pour mes encadrants Pierre-Emmanuel Hladik et Anne-Marie Déplanche qui ont su me guider durant ces trois années sans qu'il n'y ait jamais de tensions entre nous. Ce fut donc pour moi une période très agréable et j'en garderai un bon souvenir. Plus particulièrement je tiens à remercier Pierre-Emmanuel pour m'avoir soufflé l'idée de faire une thèse lorsqu'il était l'un de mes professeurs à l'INSA. Si j'ai accepté c'est aussi parce que j'avais une entière confiance en lui pour m'encadrer sérieusement, me consacrer du temps lorsque c'est nécessaire et travailler dans la bonne humeur. Je ne me suis pas trompé. Je suis également très reconnaissant envers Anne-Marie avec qui j'ai principalement échangé par email et visio-conférence étant donné la distance qui sépare Nantes de Toulouse. Cette distance m'a obligé à mettre par écrit mes idées et ce fut pour moi très bénéfique. Anne-Marie est quelqu'un de très minutieux et qui a toujours des critiques constructives à apporter sur un document. J'ai le sentiment d'avoir beaucoup appris grâce à cela.

Merci aux membres de mon jury pour avoir accepté d'évaluer mes travaux. En commençant par Liliana Cucu-Grosjean et Pascal Richard mes rapporteurs. J'ai conscience que la lecture approfondie de ce manuscrit nécessite un investissement en temps important. Je remercie aussi les autres membres du jury à savoir Jean-Charles Fabre et Yvon Trinquet, mes examinateurs. Enfin, je remercie Laurent George pour l'intérêt porté à mes travaux et de m'avoir fait l'honneur de venir assister à ma soutenance.

Pour terminer, je tiens à remercier les membres de l'équipe VERTICS du LAAS-CNRS pour m'avoir accueilli et de manière plus générale tous les collègues du LAAS que j'ai pu croiser, mais aussi de l'INSA où j'ai effectué mes enseignements. Enfin, je remercie bien sûr mes amis et ma famille qui m'ont accompagné durant ces années.

Table des matières

Introduction générale **1**

Partie I Contexte et Motivations **5**

1 Ordonnancement temps réel multiprocesseur **7**
1.1 Modélisation et vocabulaire . . . 7
1.1.1 Modélisation des applications . . . 7
1.1.2 Modélisation de l'architecture matérielle . . . 10
1.1.3 Ordonnancement . . . 11
1.2 Ordonnancement monoprocesseur . . . 14
1.2.1 Algorithmes d'ordonnancement à priorité fixe au niveau des tâches . 14
1.2.2 Algorithme d'ordonnancement à priorité fixe au niveau des travaux . 15
1.2.3 Algorithmes d'ordonnancement à priorité dynamique au niveau des travaux . . . 15
1.3 Ordonnancement multiprocesseur . . . 17
1.3.1 Ordonnancement par partitionnement . . . 18
1.3.2 Ordonnancement global . . . 20
1.3.3 Ordonnancement semi-partitionné . . . 21
1.3.4 Autres approches . . . 22
1.4 Généralisation des algorithmes monoprocesseurs . . . 23
1.4.1 Algorithmes RM-US[ξ], EDF-US[ξ], EDF$^{(k)}$ et fpEDF . . . 23
1.4.2 Algorithme Global-Fair Lateness . . . 23
1.4.3 Algorithmes d'ordonnancement à laxité nulle (*Zero Laxity*) . . . 24
1.4.4 Algorithme Global-Least Laxity First . . . 25
1.4.5 Algorithme U-EDF . . . 25
1.5 Ordonnancement global dit équitable . . . 26
1.5.1 Ordonnancement PFair . . . 27
1.5.2 Ordonnancement global DP-Fair . . . 29
1.6 Approches hybrides . . . 31
1.6.1 Ordonnancement semi-partitionné . . . 31
1.6.2 Algorithme RUN . . . 33
1.7 Conclusion . . . 35

2 Principe et fonctionnement des mémoires caches **37**
2.1 Fonctionnement d'un processeur . . . 37
2.1.1 Exécution d'un programme . . . 38
2.1.2 Exemple de programme . . . 38
2.1.3 Mémoire cache . . . 39
2.1.4 Types d'architectures . . . 40

2.1.5 Optimisations . . . 41
2.2 Principe d'une mémoire cache . . . 43
2.2.1 Technologies de mémoire . . . 43
2.2.2 Principe de localité . . . 43
2.2.3 Hiérarchie mémoire . . . 44
2.3 Fonctionnement d'un cache . . . 45
2.3.1 Associativité . . . 45
2.3.2 Protocoles de cohérence . . . 48
2.4 Sources de défauts de cache . . . 48
2.4.1 Types de défauts de cache . . . 48
2.4.2 Préemption et « context switch misses » . . . 49
2.4.3 Partage de cache et « cache thrashing » . . . 51
2.4.4 Bilan . . . 52
2.5 Caractérisation du comportement mémoire d'une tâche . . . 53
2.5.1 *Working Set Size* . . . 53
2.5.2 Nombre de cycles par instruction . . . 53
2.5.3 Fréquence d'accès au cache . . . 54
2.5.4 *Stack distance* . . . 54
2.5.5 *Reuse distance* . . . 56
2.5.6 Récapitulatif . . . 56
2.6 Prise en compte des caches dans les systèmes ordonnancés . . . 56
2.6.1 Sauvegarde du cache . . . 57
2.6.2 Partitionnement du cache . . . 57
2.6.3 Verrouillage du cache . . . 58
2.6.4 Co-ordonnancement . . . 59
2.7 Conclusion . . . 59

3 Motivations **61**
3.1 Constats . . . 61
3.1.1 De nombreuses politiques d'ordonnancement... . . . 61
3.1.2 Quelles approches pour l'évaluation de ces politiques ? . . . 62
3.1.3 Des difficultés pour combiner les résultats existants... . . . 64
3.1.4 Et des aspects opérationnels oubliés ou au second plan... . . . 65
3.2 Objectifs . . . 65
3.3 Besoins . . . 68
3.4 Évaluation des outils disponibles . . . 70
3.4.1 MAST . . . 70
3.4.2 Cheddar . . . 71
3.4.3 STORM . . . 71
3.4.4 Yartiss . . . 72
3.4.5 Conclusion sur les simulateurs . . . 73
3.5 Conclusion . . . 73

Partie II Simulation d'ordonnancement **75**

4 SimSo : un outil pour la simulation d'ordonnancement **77**
4.1 Présentation générale . . . 77
4.2 Architecture . . . 78
4.2.1 Simulation à événements discrets . . . 78
4.2.2 Structures de données pour les paramètres de simulation . . . 79
4.2.3 Classes pour la simulation . . . 81

4.2.4 Modèle de temps d'exécution . . . 84
4.3 Ordonnanceurs . . . 86
4.3.1 Fonctionnement et interface d'un ordonnanceur . . . 86
4.3.2 Difficultés liées à la mise en œuvre d'ordonnanceurs . . . 88
4.3.3 Ordonnanceurs disponibles . . . 88
4.3.4 Exemple . . . 90
4.4 Génération de tâches . . . 92
4.4.1 Choix des taux d'utilisation . . . 92
4.4.2 Choix des périodes . . . 94
4.4.3 Tâches sporadiques . . . 94
4.5 Valeurs mesurées . . . 95
4.5.1 Préemptions et migrations . . . 95
4.5.2 Dépassements d'échéance . . . 96
4.5.3 Temps de réponse et laxité normalisée . . . 97
4.5.4 Appels à l'ordonnanceur . . . 97
4.5.5 Récapitulatif des mesures disponibles . . . 97
4.6 Utilisation de Simso . . . 98
4.6.1 Interface graphique . . . 99
4.6.2 Mode script . . . 100
4.7 Conclusion . . . 105

5 Évaluation des politiques d'ordonnancement 107
5.1 Mise en place de l'évaluation . . . 107
5.1.1 Comparaison des méthodes de génération . . . 108
5.1.2 Génération des systèmes et simulations . . . 109
5.1.3 Valeurs mesurées . . . 111
5.2 Évaluation des politiques de type G-EDF . . . 112
5.2.1 Test d'ordonnançabilité GFB . . . 112
5.2.2 Comparaison des algorithmes de type G-EDF . . . 114
5.3 Ordonnancement partitionné . . . 117
5.3.1 Systèmes partitionnables . . . 118
5.3.2 Impact du placement des tâches . . . 120
5.3.3 Comparaison G-EDF et P-EDF . . . 122
5.4 Comparaison des politiques DP-Fair . . . 123
5.4.1 Comparaison de LRE-TL et LLREF . . . 124
5.4.2 DP-WRAP . . . 127
5.5 Comparaison des politiques U-EDF, RUN et EKG . . . 128
5.6 Usage du WCET . . . 131
5.6.1 Simulation d'ordonnancement avec durées aléatoires des travaux . . . 132
5.6.2 Évaluation basée sur le WCET . . . 132
5.6.3 Ordonnancement d'un système en surcharge . . . 134
5.7 Avantages liés aux politiques conservatives . . . 135
5.7.1 Ordonnancement ER-Fair . . . 136
5.7.2 Algorithme d'ordonnancement NVNLF . . . 137
5.7.3 Combiner G-EDF à des politiques non conservatives . . . 140
5.8 Conclusion . . . 143

Partie III Simulation avec impact des caches 145

6 Modèles de cache 147
6.1 Contexte . . . 147

6.1.1 Besoin pour la simulation . . . 148
6.1.2 Entrées disponibles . . . 148
6.1.3 Hypothèses . . . 149
6.2 Présentation des modèles . . . 149
6.2.1 Évaluation du CPI . . . 149
6.2.2 Défauts de cache pour une tâche isolée . . . 151
6.2.3 Défauts de cache pour une tâche avec un cache partagé . . . 151
6.2.4 Estimation des coûts de préemption . . . 157
6.2.5 Estimation des défauts de cache suite à une migration . . . 162
6.3 Évaluation des modèles . . . 162
6.3.1 Choix des outils et benchmarks . . . 163
6.3.2 Caractérisation du comportement mémoire des programmes . . . 166
6.3.3 Caches N-Way et fully-associative . . . 170
6.3.4 Évaluation du comportement des tâches en isolation . . . 172
6.3.5 Partage de cache . . . 175
6.3.6 Coûts des préemptions . . . 177
6.3.7 Conclusion . . . 184

7 Prise en compte des surcoûts temporels dans SimSo 185
7.1 Prise en compte des aspects opérationnels dans SimSo . . . 185
7.1.1 Intégration des pénalités temporelles . . . 185
7.1.2 Intégration de modèles de cache dans SimSo . . . 187
7.2 Expérimentations . . . 191
7.2.1 Comparaison de G-EDF et P-EDF avec pénalités temporelles . . . 191
7.2.2 Variation des *offsets* avec prise en compte des caches . . . 193
7.2.3 Impact du placement des tâches . . . 195
7.2.4 Pénalités de préemption et migration . . . 197
7.3 Conclusion . . . 198

Conclusion 201

A Boite à outils pour l'évaluation 205
A.1 Simulation d'ordonnancement . . . 205
A.1.1 STRESS et PERTS . . . 205
A.1.2 GHOST et RTSIM . . . 206
A.1.3 FORTISSIMO . . . 206
A.1.4 FORTAS . . . 206
A.1.5 RealtssMP . . . 206
A.2 Exécutions réelles . . . 207
A.2.1 Linux . . . 207
A.2.2 LITMUS^{RT} . . . 207
A.2.3 RESCH . . . 207
A.2.4 ChronOS Linux . . . 208
A.3 Simulateurs d'architecture . . . 208
A.3.1 Gem5 . . . 208
A.3.2 Simics . . . 209
A.3.3 SESC et ESESC . . . 209
A.3.4 MARSSx86 et PTLSim . . . 209
A.3.5 Autres . . . 210
A.4 Mesure de SDP et analyse des caches . . . 210
A.4.1 MICA . . . 210
A.4.2 Cachegrind . . . 211

A.4.3 Stressmark Workload 211
A.4.4 StatStack 211
A.5 Benchmarks 211
A.5.1 Benchmarks orientés temps réel 212
A.5.2 Benchmarks orientés embarqués 212
A.5.3 Benchmarks orientés WCET 213
A.5.4 Conclusion sur les benchmarks 214

Bibliographie **215**

Introduction générale

Contexte

Un système temps réel est un système informatique ayant pour particularité de devoir respecter des contraintes temporelles en plus de produire des résultats logiques corrects [Sta88]. Afin de garantir le respect de ces contraintes, il est nécessaire de construire une séquence d'exécution des différents programmes qui composent le système. Cette séquence est appelée un *ordonnancement*. Elle peut être réalisée de manière statique en fixant les instants d'exécution de chaque programme, ou bien de manière dynamique en laissant un composant du système, nommé *ordonnanceur*, décider pendant l'exécution du système quel programme doit s'exécuter à chaque instant. Cette dernière approche offre plus de souplesse au système pour s'adapter aux variations dues à l'exécution des programmes, mais rend plus difficile la preuve du respect des contraintes temporelles pour le système. Cela est d'autant plus vrai que les supports d'exécution deviennent complexe.

Pendant de nombreuses années, les constructeurs de processeurs ont privilégié l'augmentation de la fréquence d'horloge et l'optimisation de l'architecture afin d'accroître les capacités de calcul des plateformes d'exécution, ce qui n'avait qu'un impact sur les durées d'exécution des programmes et donc assez peu d'influence sur les problématiques d'ordonnancement. Cependant, l'augmentation de la fréquence d'horloge entraîne une augmentation de la puissance consommée ainsi que de la température dissipée par effet Joule. De même, l'optimisation des architectures nécessite une augmentation du nombre de transistors et une réduction de leur taille, ce qui a tendance à accentuer les phénomènes de fuite de courant, limitant ainsi les fréquences exploitables. En conséquence, la fréquence des processeurs n'augmente plus depuis le début des années 2000 et les constructeurs ont basculé sur des solutions constituées de plusieurs cœurs de processeur afin de proposer des architectures toujours plus puissantes en terme de capacités de calcul.

Ce passage d'architecture monoprocesseur à des architectures multicœurs [1] ne va pas sans poser de nombreux problèmes : parallélisation des programmes, limite exposée par la loi d'Amdhal, etc. Cela est d'autant plus vrai pour l'ordonnancement temps réel. En effet, le problème du choix des programmes à exécuter pour respecter les contraintes temporelles est augmenté d'une nouvelle dimension : en plus de choisir les programmes à exécuter, l'ordonnanceur doit aussi choisir leur répartition spatiale sur les processeurs. Cette différence rend le problème bien plus complexe et fait apparaître de nouveaux comportements contre-intuitifs, comportements couramment appelés anomalies d'ordonnancement. Par exemple, la réduction de la quantité de travail à réaliser par le système suite à l'augmentation de

1. Une architecture multicœur est un cas particulier d'architecture multiprocesseur où tous les processeurs sont regroupés sur une même puce. Ainsi, par la suite, nous utiliserons le terme « multiprocesseur » pour désigner ce type d'architecture.

la durée entre deux exécutions d'un même programme peut entraîner le non respect des contraintes temporelles.

De nombreuses approches ont été proposées pour répondre aux besoins de l'ordonnancement temps réel multiprocesseur. Dans un premier temps les politiques d'ordonnancement monoprocesseur ont été adaptées aux architectures multiprocesseurs. Deux grandes catégories d'algorithmes ont ainsi été étudiées. La première consiste à partitionner les programmes entre les cœurs de manière à retrouver plusieurs problèmes monoprocesseurs indépendants. La seconde approche s'appuie sur une vision globale du système et propose des algorithmes décidant en ligne quels programmes doivent être exécutés en fonction des ressources disponibles. Au milieu des années 1990, ces algorithmes ont évolués avec des techniques à base d'équité (*fairness*), de semi-partitionnement, de clustering, etc. Ce regain d'intérêt pour cette problématique ne faiblit pas et l'on peut recenser plus d'une cinquantaine de politiques différentes dont plus de la moitié ont été publiées durant ces dix dernières années.

Face à une telle foison d'algorithmes aux approches diverses, il est difficile de déterminer leurs points forts et leurs faiblesses, et les évaluations proposées dans la littérature ne sont pas toujours clairement décrites et complètes. En effet, ces évaluations sont souvent proposées par les auteurs des algorithmes dans le but d'en montrer les qualités en les comparant à quelques politiques similaires. Malheureusement, ces études sont trop souvent limitées dans les configurations testées et les données collectées. De plus, il est très compliqué d'unifier les différents résultats existants car les expérimentations ne sont pas réalisées dans les mêmes conditions (entrées différentes, métriques différentes, implémentations des algorithmes différentes, etc).

Par ailleurs, la plupart des efforts de recherche se sont concentrés sur les aspects algorithmiques et théoriques. Plus particulièrement, une grande attention a été portée à la définition de tests d'ordonnançabilité. Ces tests reposent généralement sur des modèles théoriques qui ne prennent pas (ou partiellement) en compte les architectures matérielles et logicielles. Or les architectures multiprocesseurs apportent plus de complexité avec notamment des caches partagés, de nouveaux bus de communication, des interruptions interprocesseurs, etc. De même, au niveau du système, de nouvelles questions se posent telles que le choix du cœur qui doit exécuter l'ordonnanceur, ou encore quelles données doivent être verrouillées pour éviter les accès concurrents. Tous ces éléments sont la source de surcoûts temporels que nous ne pouvons donc pas ignorer au niveau de la définition et de l'évaluation d'un ordonnanceur.

Contributions

Dans ce contexte, le travail présenté dans ce mémoire s'inscrit de manière générale dans la problématique d'évaluation des algorithmes d'ordonnancement en garantissant des conditions expérimentales maîtrisées et en prenant en considération les aspects opérationnels du système (logiciel et matériel). Le but est d'identifier les caractéristiques des systèmes ayant un impact sur les performances de chaque politique afin d'en obtenir une meilleure connaissance et maîtrise.

Pour réaliser ces évaluations, nous avons fait le choix de la simulation d'ordonnancement. Ce type d'approche se base sur une modélisation du système qui ne prend en compte que les comportements pouvant avoir un impact sur l'ordonnancement. Ainsi, les programmes ne sont représentés que par leur durée d'exécution et seuls les événements d'ordonnancement

sont considérés. Cela a pour effet d'abstraire les détails de l'architecture réelle ainsi que le comportement des programmes.

Cependant, certains éléments du système ont une importance sur l'exécution des programmes, ainsi, et sans aller jusqu'à faire une simulation précise d'une architecture réelle, nous avons affiné les modèles utilisés par la simulation. Parmi les éléments matériels pouvant avoir une telle influence, nous avons fait le choix de nous concentrer sur les caches car l'utilisation efficace des caches dépend directement des choix de l'ordonnanceur (préemptions, migrations, ou encore partage de cache entre plusieurs cœurs). De plus, une utilisation efficace des caches permet de réduire les durées d'exécution ce qui a un impact sur les temps de réponse, la fiabilité des systèmes temps réel mais aussi la consommation énergétique des systèmes. Nous avons également ajouté à la simulation la prise en compte des surcoûts temporels liés à l'exécution du code de l'ordonnanceur.

Les travaux menés conduisent ainsi à deux contributions majeures. La première est la mise à disposition et l'exploitation d'un outil de simulation, *SimSo*, dédié à l'évaluation de politiques d'ordonnancement pour des architectures multiprocesseurs [CHD14]. Sa conception a été guidée par la volonté de pouvoir modifier facilement les modèles utilisés pour la simulation dans le but de s'adapter aux besoins de l'utilisateur (modèles plus précis ou tâches plus complexes par exemple). Plus de vingt-cinq algorithmes d'ordonnancement ont été mis en œuvre dans SimSo ce qui démontre sa capacité à simuler des ordonnancements aux approches variées (global, partitionné, semi-partitionnné, hybride, etc). SimSo permet également d'automatiser de larges campagnes d'évaluation. Grâce à cela, une vingtaine d'algorithmes d'ordonnancement ont pu faire l'objet d'une évaluation présentée dans cette thèse.

La seconde contribution est l'intégration de modèles statistiques de cache dans la simulation. Pour cela, nous avons étudié des modèles développés par d'autres communautés et qui servent à évaluer l'usage des caches par des programmes. À partir de cette étude, nous avons retenu un ensemble de modèles que nous avons mis en œuvre dans SimSo. Ceci permet de simuler un système doté d'un ou plusieurs niveaux de caches, partagés ou non. Les effets des caches sur les durées d'exécution sont pris en compte ce qui permet d'affiner les évaluations des algorithmes d'ordonnancement, en particulier pour ceux qui visent explicitement à utiliser au mieux les caches.

Plan du manuscrit

Le travail qui est présenté s'articule autour de trois grandes parties.

La première partie de cette thèse introduit les notions de base utiles à sa compréhension tout en établissant un ensemble de constats qui ont motivé ce travail. Le chapitre 1 présente les approches existantes pour l'ordonnancement de tâches temps réel sur des architectures multiprocesseurs. L'architecture matérielle ne devrait pas être négligée dans l'évaluation des algorithmes. Ainsi, le fonctionnement d'un processeur et plus particulièrement des caches est présenté dans le chapitre 2. Cette première partie se termine par le chapitre 3 exposant ce qui motive notre travail, quels sont nos objectifs, nos besoins et quelle est notre approche.

Au cours de la seconde partie, nous présentons SimSo, notre simulateur conçu pour faciliter l'étude des politiques d'ordonnancement temps réel pour architectures multiprocesseurs. Ses fonctionnalités, son architecture et son utilisation y sont notamment détaillées dans

le chapitre 4. Nous montrons ensuite, dans le chapitre 5, que SimSo constitue un outil intéressant pour l'étude de politiques d'ordonnancement à travers les expérimentations que nous avons menées pour évaluer comparativement les principaux algorithmes. Au total, une vingtaine d'ordonnanceurs ont été évalués sur plus de 10 000 systèmes générés aléatoirement et identiques pour chaque ordonnanceur.

La troisième partie traite de modèles statistiques pour évaluer l'usage des caches et par conséquent la vitesse d'exécution des programmes. Nous nous sommes demandé dans quelle mesure ces modèles pourraient être utilisés au cœur de la simulation d'ordonnancement pour prendre en compte de manière réaliste (et non pire cas) l'effet des caches sur le système. Dans un premier temps, au chapitre 6, les modèles ayant retenu le plus notre attention sont présentés puis évalués. Ensuite, au chapitre 7, nous décrivons comment ces modèles ont pu être intégrés au sein de SimSo puis des expérimentations sont finalement présentées afin d'illustrer la prise en compte des caches dans la simulation.

Cadre de ces recherches

Cette thèse a été financée par le projet RESPECTED (*Real-time Executive Support with scheduling Policies for thErmally-Constrained mulTicore EmbeddeD systems*), projet ANR du programme ARPEGE qui s'est déroulé du 1er décembre 2010 au 31 mai 2014 et auquel ont pris part les laboratoires IRCCyN (Nantes), LAAS-CNRS (Toulouse), LEAT (Nice) et la société See4sys (Nantes). Ce projet de recherche industrielle a porté sur l'étude d'algorithmes d'ordonnancement temps réel, prenant en compte les aspects thermiques et le partage de ressources dans un contexte multicœur.

Première partie

Contexte et Motivations

CHAPITRE 1

Ordonnancement temps réel multiprocesseur

Ce chapitre propose un panorama des techniques d'ordonnancement en-ligne de tâches temps réel pour des architectures multiprocesseurs. Une cinquantaine de politiques d'ordonnancement ont été recensées au cours de cet état de l'art. Dans ce chapitre, on se contentera de présenter les caractéristiques essentielles des politiques qui nous semblent majeures. Les nombreuses références qui jalonnent cette présentation permettront au lecteur d'approfondir la compréhension d'une politique en particulier.

Afin de pouvoir présenter dans un cadre unifié les différentes approches pour l'ordonnancement de tâches indépendantes sur une architecture multiprocesseur, ce chapitre débute par une présentation des modèles de tâches et de quelques définitions importantes. Historiquement mais aussi fonctionnellement parlant, le contexte multiprocesseur étant une généralisation du cas monoprocesseur, il nous est apparu nécessaire de présenter en premier lieu, les principaux algorithmes d'ordonnancement destinés à des systèmes monoprocesseurs.

1.1 Modélisation et vocabulaire

Dans cette première partie, le modèle de tâches temps réel le plus fréquemment employé dans le domaine est présenté. Il permettra ensuite d'asseoir l'exposé des politiques introduites dans la suite de ce chapitre. Quelques définitions relatives à l'ordonnancement sont formulées dans un second temps. Ceci permet également d'introduire le vocabulaire lié au domaine.

1.1.1 Modélisation des applications

Une application temps réel est constituée d'un ensemble de tâches (*tasks*). Une tâche contrôle le flot d'exécution d'un programme. Les instructions exécutées forment ce que l'on appelle un travail (*job*). Ainsi, une tâche, si elle est récurrente, donne lieu à une succession de travaux, parfois appelés les instances de la tâche. Les travaux d'une application temps réel doivent respecter un certain nombre de contraintes temporelles qui seront présentées par la suite.

Le modèle de tâches décrit ici a été initialement formulé par Liu et Layland [LL73]. Ce modèle permet de traiter le cas de tâches périodiques mais il a ensuite été généralisé aux tâches sporadiques [Mok83, LW82].

Notons cependant l'existence d'autres modèles tels que le modèle *multiframe* [MC96] et sa généralisation [BCGM99] qui tentent de prendre en considération les variations de durées d'exécution des tâches afin d'affiner les analyses d'ordonnançabilité, les *transactions* [Tin94] qui introduisent des dépendances temporelles entre les tâches, le modèle de *tâches récurrentes* [Bar03] qui modélise explicitement les branchements conditionnels ou encore le modèle *digraph* [SEGY11] où le comportement des travaux est modélisé par un graphe acyclique. Nous ne considérerons pas ces modèles dans la suite car ils sont très peu traités par les algorithmes multiprocesseurs, **nous nous focaliserons uniquement sur le modèle initial de Liu et Layland**.

a) Modélisation des travaux

Soit un travail activé à l'instant a et devant se terminer avant l'instant d. Ce travail s'exécute pendant e unités de temps sur l'intervalle $[a, d]$. Ces grandeurs, représentées à travers un exemple par la figure 1.1, permettent de caractériser un travail par le triplet (a, e, d) :

- a l'instant d'activation (*release time*) ;
- e le temps d'exécution (*computation time*) ;
- d l'échéance absolue (*absolute deadline*).

On dit d'un travail qu'il est actif à l'instant t si les conditions suivantes sont réunies :

- le travail est arrivé ($a \leq t$) ;
- l'échéance n'est pas dépassée ($t < d$) ;
- le travail n'a pas fini son exécution.

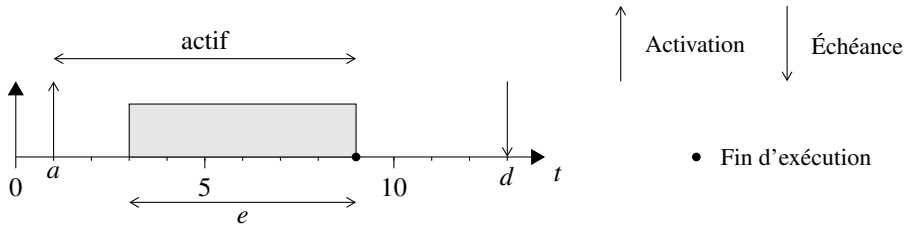


FIGURE 1.1 – Schéma représentatif d'un travail et de ses paramètres.

Certains modèles permettent à des travaux de s'exécuter en parallèle sur plusieurs processeurs dans le cas de programmation par *multithreading*. **Nous prendrons ici pour hypothèse qu'un travail ne peut s'exécuter que sur un seul processeur à la fois.**

b) Modélisation des tâches

L'activation d'une tâche engendre la création d'un travail dans l'état actif. La manière dont les travaux sont générés par une tâche permet de faire la distinction entre trois natures de tâches :

- les tâches périodiques sont activées régulièrement à période fixe ;
- les tâches sporadiques sont activées de manière irrégulière mais avec toutefois au moins une propriété sur la durée minimum entre l'arrivée de deux travaux consécutifs ;
- les tâches apériodiques sont activées de manière irrégulière.

Un système temps réel τ est constitué d'une ensemble fini de tâches : $\tau = \{\tau_1, ..., \tau_n\}$. Chaque tâche τ_i étant constituée d'une suite infinie de travaux, on note : $\tau_i = \{\tau_{i,1}, \tau_{i,2}, ...\}$ où $\tau_{i,j}$ est le j^{ème} travail de la tâche τ_i.

Une tâche τ_i *périodique* ou *sporadique* est caractérisée par le quadruplet (O_i, C_i, T_i, D_i) :

- O_i, la date d'activation du premier travail de la tâche τ_i (0 si non précisé) ;
- la durée d'exécution C_i dans le pire cas (*Worst-Case Execution Time*, WCET) de chaque travail de la tâche τ_i ;
- sa période d'activation T_i. Dans le cas de tâches sporadiques, c'est la durée minimale entre ses activations successives ;
- son échéance relative ou délai critique D_i. C'est la durée entre l'arrivée d'un travail et son échéance (un travail qui arrive à l'instant t doit se terminer avant l'instant $t+D_i$).

Le *taux d'utilisation* d'une tâche correspond à la fraction de temps que la tâche consomme sur un processeur pour s'exécuter. Cette grandeur est très utilisée dans les tests d'ordonnançabilité, ainsi que la *somme totale des taux d'utilisation* ou encore le *plus grand taux d'utilisation* du système. Une tâche dont le taux d'utilisation est grand sera qualifiée de tâche *lourde*, et au contraire une tâche dont le taux d'utilisation est faible sera qualifiée de tâche *légère*. Le seuil à partir duquel une tâche est dite lourde ou légère dépend du contexte (généralement 0.5).

- Taux d'utilisation d'une tâche τ_i :

$$u_i \stackrel{\text{def}}{=} \frac{C_i}{T_i} \tag{1.1}$$

- Utilisation totale du système :

$$u_{sum} \stackrel{\text{def}}{=} \sum_{i=1}^{n} u_i \tag{1.2}$$

- Taux d'utilisation moyen d'un processeur :

$$\frac{u_{sum}}{m}, \text{ avec } m \text{ le nombre de processeurs} \tag{1.3}$$

- Plus grand taux d'utilisation du système :

$$u_{max} \stackrel{\text{def}}{=} \max\{u_1, ..., u_n\} \tag{1.4}$$

Une distinction importante entre les tâches est faite sur leur type d'échéance :

- si $D_i = T_i$, on parle de tâche à échéance implicite ou échéance sur requête (*Implicit-deadline*) ;
- si $D_i < T_i$, on parle de tâche à échéance contrainte (*Constrained-deadline*) ;
- sinon, on parle de tâche à échéance arbitraire (*Arbitrary-deadline*).

Enfin, si les tâches démarrent au même instant ($\forall i, O_i = 0$), on parle de système de tâches à départ simultané ou tâches synchrones (*synchronous task system*).

c) Représentation graphique et synthèse des notations

Les notions les plus importantes sont représentées par la figure 1.2 à travers un exemple d'exécution d'une tâche périodique à échéance contrainte τ_i avec ses premiers travaux. La période de la tâche τ_i (T_i) est de 16 et son échéance (D_i) est de 14. Le premier travail de la tâche τ_i s'active à $t = 3$ ($a_{i,1} = O_i$), mais ne commence à s'exécuter qu'à $t = 4$ ($s_{i,1}$). Ce travail est suspendu en $t = 7$ et reprend son exécution en $t = 10$ pour finalement se terminer en $t = 15$ ($f_{i,1}$). L'échéance de la tâche est bien respectée puisque $f_{i,1} \leq d_{i,1} = a_{i,1} + D_i$. Le temps de réponse ($R_{i,1}$) correspond à la durée entre l'activation du travail et la fin de son exécution : $R_{i,1} = f_{i,1} - a_{i,1} = 12$.

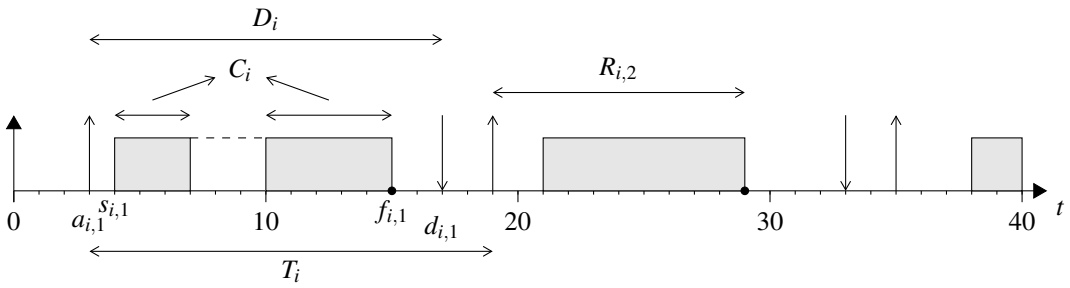


FIGURE 1.2 – Schéma représentatif d'une tâche τ_i. $s_{i,j}$ est la date de début d'exécution du travail $\tau_{i,j}$, $f_{i,j}$ sa date de fin d'exécution, $R_{i,j}$ est le temps de réponse du travail.

Les grandeurs présentées dans cette partie sont synthétisées dans le tableau 1.1.

1.1.2 Modélisation de l'architecture matérielle

Un système temps réel est composé d'une application (ensemble de tâches) et d'une architecture matérielle sur laquelle les travaux pourront s'exécuter. Dans le cadre d'architectures multiprocesseurs, on distingue trois types d'architectures :

- Processeurs identiques (ou homogènes). Tous les processeurs qui constituent la plateforme matérielle présentent des caractéristiques identiques et sont donc parfaitement interchangeables.
- Processeurs uniformes. Chaque processeur est caractérisé par sa capacité de calcul : lorsqu'un travail s'exécute sur un processeur de capacité de calcul s pendant t unités de temps, il réalise $s \times t$ unités de travail.

τ	Ensemble de tâches
τ_i	i-ème tâche de l'ensemble τ
T_i	Période de la tâche τ_i
D_i	Échéance relative de la tâche τ_i
C_i	Durée d'exécution (WCET) de la tâche τ_i
O_i	Date d'activation du premier travail de la tâche τ_i
u_i	Taux d'utilisation de la tâche τ_i
u_{sum}	Taux d'utilisation total du système
u_{max}	Plus grand taux d'utilisation du système
$\tau_{i,j}$	j-ème travail de la tâche τ_i
$a_{i,j}$	Date d'arrivée du travail $\tau_{i,j}$
$e_{i,j}$	Durée d'exécution du travail $\tau_{i,j}$ ($e_{i,j} \leq C_i$)
$d_{i,j}$	Échéance absolue du travail $\tau_{i,j}$
$s_{i,j}$	Date de début d'exécution du travail $\tau_{i,j}$
$f_{i,j}$	Date de fin d'exécution du travail $\tau_{i,j}$
$R_{i,j}$	Temps de réponse du travail $\tau_{i,j}$

Tableau 1.1 – Tableau récapitulatif des notations.

- Processeurs indépendants (ou hétérogènes). Une capacité de calcul $c_{i,k}$ est associée à chaque couple tâche-processeur (τ_i, P_k). L'interprétation est la suivante : les travaux de la tâche τ_i réalisent $c_{i,k} \times t$ unités de travail lorsqu'ils s'exécutent sur le processeur P_k pendant t unités de temps. Ce modèle permet aussi de gérer les processeurs spécialisés qui ne peuvent traiter que certains travaux : il suffit de fixer $c_{i,k}$ à 0 pour exprimer que le processeur P_k ne peut pas prendre en charge la tâche τ_i.

1.1.3 Ordonnancement

L'ordonnancement d'un système consiste à définir une allocation du ou des processeurs aux travaux de sorte à ce que toutes les contraintes qui peuvent accompagner leur exécution soient satisfaites au mieux. Ce problème peut être considéré comme un cas particulier de planification et un vaste choix de méthodes est disponible pour planifier à l'avance l'ordonnancement des tâches. Ce type d'ordonnancement est appelé hors-ligne car il se fait avant l'exécution du système. **Les travaux présentés ici concernent des approches *en-ligne* qui décident de l'ordonnancement pendant l'exécution du système.** Une exécution *en-ligne* est plus souple et permet de tenir compte de paramètres variables au cours de l'exécution (travail qui se termine plus tôt, activation arbitraire d'une nouvelle tâche, etc).

a) Ordonnançabilité

Les études d'ordonnançabilité visent à déterminer si un système est ordonnançable par une politique d'ordonnancement sur une architecture matérielle donnée.

Définition 1.1. *Un système S est fiablement ordonnancé par un algorithme d'ordonnancement A, si et seulement si la séquence produite par A est valide.*

Définition 1.2. *Un système S est ordonnançable s'il existe un algorithme qui l'ordonnance fiablement.*

Un ensemble de conditions suffisantes et/ou nécessaires permettent de déterminer, dans certains cas, si un système est ordonnançable (condition suffisante satisfaite), ou s'il ne l'est pas (condition nécessaire non satisfaite).

Définition 1.3. *Lorsqu'un système respecte une condition suffisante de l'algorithme d'ordonnancement utilisé, alors il est ordonnançable par cet algorithme.*

Définition 1.4. *Lorsqu'un système ne respecte pas une condition nécessaire de l'algorithme d'ordonnancement utilisé, alors il n'est pas ordonnançable par cet algorithme.*

De nombreux travaux sur l'ordonnancement visent à fournir des algorithmes optimaux (Définition 1.5), mais cette optimalité n'est atteinte que dans un cadre précis (restriction sur le type de tâches et sur l'architecture matérielle) et sous certaines hypothèses (exemple : coût des préemptions ou des prises de décision nulles).

Définition 1.5. *Un algorithme d'ordonnancement A est dit optimal pour une classe de systèmes et selon les hypothèses posées, parmi une classe de politiques d'ordonnancement, si et seulement si, il peut ordonnancer fiablement tout système ordonnançable par une politique de cette classe.*

Un algorithme A domine un algorithme B si tout système ordonnançable par B l'est par A et s'il existe au moins un système ordonnançable par A et qui ne l'est pas par B. S'il existe des systèmes ordonnançables par l'un et non par l'autre et inversement, on dit que ces politiques ne sont pas comparables (Définition 1.6).

Définition 1.6. *Deux algorithmes A et B sont dits non comparables si et seulement si :*

- *il existe un système ordonnançable par A et non ordonnançable par B ;*
- *et il existe un système ordonnançable par B et non ordonnançable par A.*

b) Ordonnancement conduit par le temps ou les évènements

Deux grandes façons d'implémenter un ordonnanceur existent. La première est conduite par le temps (*tick-driven*) et consiste à appeler l'ordonnanceur périodiquement. L'ordonnanceur prend une décision d'ordonnancement qui sera alors appliquée par le système d'exploitation. L'autre méthode est conduite par les évènements (*event-driven*) et consiste à réagir aux évènements tels que des activations ou terminaisons de tâche. L'utilisation de l'une ou l'autre de ces méthodes est fortement liée à l'architecture du système d'exploitation.

c) Ordonnancement avec préemptions

Si un système permet d'interrompre un travail, d'en mémoriser l'état, et de le relancer plus tard, on dit que le système est préemptif (Définition 1.7). Cette interruption s'appelle une préemption et provoque un changement de contexte si une autre tâche s'exécute à la place.

Définition 1.7. *Si un ordonnanceur peut interrompre l'exécution d'un travail pour en exécuter un autre, alors le système est préemptif.*

La possibilité de provoquer des préemptions permet d'ordonnancer correctement un nombre d'ensembles de tâches bien plus grand. Cependant, pour certains systèmes et avec une même politique d'ordonnancement, un système ordonnançable en non-préemptif peut ne plus l'être en autorisant des préemptions.

Dans la suite de ce chapitre, seuls des systèmes préemptifs sont considérés. De plus, la plupart des travaux qui seront présentés supposent une durée de préemption nulle ou déjà intégrée dans le calcul du pire temps d'exécution (WCET).

d) Ordonnancement conservatif

Un ordonnanceur conservatif, ou *work-conserving*, est un ordonnanceur qui ne laisse pas des tâches prêtes en attente si un processeur est disponible pour l'exécuter (Définition 1.8). Dans la partie consacrée à la présentation de politiques d'ordonnancement, nous verrons qu'un certain nombre de politiques ne sont pas conservatives. Généralement, une politique *work-conserving* permet de réduire les temps de réponse et s'adapte mieux à des durées d'exécution effectives inférieures au WCET.

Définition 1.8. *Un ordonnanceur conservatif a comme particularité de ne jamais laisser un processeur dans un état oisif (idle) s'il reste des travaux prêts à s'exécuter. Au contraire, un ordonnanceur non-conservatif peut introduire des laps de temps pendant lesquels aucun travail actif ne s'exécute.*

e) Anomalies et prédictabilité

Définition 1.9. *Une anomalie d'ordonnancement apparaît lorsqu'un changement dans les paramètres du système engendre des effets contre-intuitifs [DB11].*

Ces anomalies peuvent apparaître dans le cas de systèmes non-préemptifs ou dans le cas de systèmes multiprocesseurs. Un exemple est l'augmentation de la période d'une tâche, soit une baisse du taux d'utilisation, qui devrait intuitivement améliorer l'ordonnançabilité, mais qui en pratique peut rendre un ensemble de tâches non ordonnançable. Andersson étudie en détail cette problématique dans le chapitre 5 de sa thèse [And03].

Définition 1.10. *Un algorithme d'ordonnancement est dit prédictible si les temps de réponse des travaux ne peuvent pas être augmentés par une réduction de leur durée d'exécution, avec tous les autres paramètres constants [HL94].*

Cette propriété est importante car la durée d'exécution des travaux est seulement modélisée par une durée maximale (WCET). Ainsi l'ordonnançabilité d'un système prédictible est garantie, si elle est vérifiée sur le modèle avec WCET. Nous verrons cependant que se limiter à une durée fixe peut conduire à des évaluations biaisées du comportement des algorithmes d'ordonnancement (voir chapitres 3 et 4).

Définition 1.11. *Un algorithme d'ordonnancement est dit viable (sustainable) pour un modèle de système, si et seulement si l'ordonnançabilité d'un ensemble de tâches conforme à ce modèle reste inchangée après une réduction des durées d'exécution, une augmentation des périodes ou des dates d'inter-arrivée, ou une augmentation des échéances.*

1.2 Ordonnancement monoprocesseur

Les politiques d'ordonnancement temps réel les plus connues sont très certainement Rate Monotonic et Earliest Deadline First. Il en existe cependant d'autres, dont l'utilité dépend du type de système. Classiquement, on considère que l'ordonnanceur choisit d'exécuter les travaux les plus prioritaires à tout moment. La gestion de ces priorités peut être distinguée de trois manières différentes [CFH+04] :

- Les algorithmes *à priorité fixe au niveau des tâches*, pour lesquels les tâches et les travaux partagent la même priorité. Exemple : Rate Monotonic.
- Les algorithmes *à priorité fixe au niveau des travaux* qui attribuent aux travaux des priorités fixes pendant l'exécution. Ainsi, les travaux d'une même tâche peuvent avoir des priorités différentes, mais cette priorité ne varie pas pendant l'exécution d'un travail. Exemple : Earliest Deadline First.
- Les algorithmes *à priorité dynamique au niveau des travaux* qui définissent à chaque instant une priorité pour chaque travail. Exemple : Least Laxity First.

Il est cependant courant que les deux dernières catégories soient regroupées sous le terme d'algorithme à priorité dynamique. Cette classification demeure valide dans le cas multiprocesseur.

Enfin, notons que pour toutes les politiques d'ordonnancement monoprocesseur, la condition nécessaire d'ordonnançabilité suivante doit être satisfaite :

$$\sum_{i=1}^{n} u_i \leq 1 \tag{1.5}$$

1.2.1 Algorithmes d'ordonnancement à priorité fixe au niveau des tâches

Dans le cas des politiques à priorité fixe au niveau des tâches, chaque tâche possède une priorité, qui est définie avant l'exécution du système et qui ne varie pas.

a) Rate Monotonic

Avec l'ordonnanceur *Rate Monotonic* (RM) est un ordonnanceur à priorité fixe. La priorité des tâches est fonction de leur période : plus elle est petite et plus la tâche est prioritaire. Cette affectation de priorité est optimale pour des tâches périodiques, indépendantes, synchrones et à échéances implicites.

Dans [LL73], les auteurs déterminent une condition suffisante d'ordonnançabilité pour des systèmes de n tâches à échéance implicite, sporadiques et synchrones (équation 1.6). Une autre condition suffisante, appelée condition hyperbolique, a été introduite par [BB01]. Celle-ci offre une meilleure limite d'utilisation pour un temps de calcul similaire.

$$\sum_{i=1}^{n} u_i \leq n(2^{\frac{1}{n}} - 1) \tag{1.6}$$

Il est aussi possible de décider de manière exacte de l'ordonnançabilité d'un système par RM, les travaux de [LSD89, ABR+93] vont dans ce sens. La complexité de ces tests étant pseudo-polynomiale, il est impossible de faire ces tests en ligne. Pour cela, des approximations sont fournies par d'autres tests [BB02].

b) Deadline Monotonic

L'ordonnanceur *Deadline Monotonic* (DM) est un ordonnanceur tel que les priorités sont affectées en fonction de l'échéance relative des tâches : plus l'échéance est petite et plus la tâche est prioritaire [LW82]. L'affectation des priorités est optimale pour des systèmes de tâches périodiques, indépendantes, synchrones et à échéance contrainte. Dans le cas des systèmes à échéances implicites, Deadline Monotonic et Rate Monotonic se confondent.

c) Optimal Priority Assignment

L'algorithme OPA, proposé par Audsley [Aud91], permet d'affecter les priorités des différentes tâches de façon optimale pour des systèmes de tâches asynchrones avec des échéances arbitraires. Par optimal, nous entendons que s'il existe une solution alors l'algorithme produit toujours une affectation des priorités de telle manière à ce que le système soit ordonnançable (selon un test d'ordonnançabilité donné).

1.2.2 Algorithme d'ordonnancement à priorité fixe au niveau des travaux

Un algorithme à priorité fixe au niveau des travaux est un algorithme pour lequel la priorité des tâches peut varier d'un travail à un autre mais la priorité d'un travail ne change pas. Ces algorithmes sont cependant parfois appelés algorithmes à priorité dynamique dans le sens où la priorité d'une tâche peut varier.

a) Earliest Deadline First

Earliest Deadline First (EDF) est un ordonnanceur tel que la priorité d'un travail est plus forte lorsque son échéance absolue est plus proche. Généralement, EDF est utilisé avec préemption : si une nouvelle tâche arrive et que sa date d'échéance absolue est plus rapprochée, alors la tâche en cours d'exécution est préemptée et la nouvelle tâche est exécutée.

Pour des tâches périodiques, indépendantes et à échéances contraintes, on retiendra que EDF est un algorithme d'ordonnancement optimal [BG03b]. EDF offre de meilleures performances que RM en terme d'ordonnançabilité [But05]. La condition nécessaire et suffisante dans ce cas est : $\sum_{i=1}^{n} u_i \leq 1$

1.2.3 Algorithmes d'ordonnancement à priorité dynamique au niveau des travaux

Un algorithme à priorité dynamique au niveau des travaux est un algorithme qui peut modifier le niveau de priorité d'un travail à tout moment.

a) Least Laxity First

La laxité dynamique d'un travail à un instant donné est le temps maximum pendant lequel son exécution peut être retardée sans manquer son échéance. La laxité dynamique est définie par : $d - (t_{cur} + c_{ret})$ avec d la date d'échéance absolue, t_{cur} la date actuelle et c_{ret} le temps d'exécution restant du travail (voir Figure 1.3).

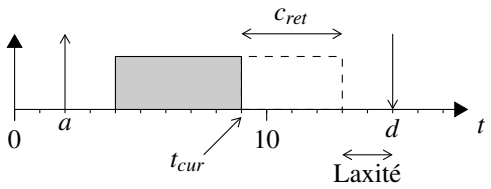


FIGURE 1.3 – Représentation visuelle de la notion de laxité dynamique.

L'algorithme Least Laxity First (LLF) rend prioritaire les travaux dont la laxité dynamique est la plus faible. C'est donc un algorithme avec priorité dynamique au niveau des travaux car la priorité d'un travail évolue dans le temps (la priorité de chaque travail non exécuté augmente au fil du temps tandis qu'elle reste constante pour les autres).

Contrairement à un algorithme à priorité fixe pour les travaux tel que EDF, il est nécessaire pour LLF de mettre à jour fréquemment les priorités des travaux alors que pour EDF ce n'est fait qu'à l'activation de la tâche.

LLF est optimal pour l'ordonnancement de tâches préemptives périodiques ou sporadiques, mais perd son optimalité pour l'ordonnancement de tâches non-préemptives [Mok83].

Cet algorithme peut cependant engendrer de nombreux changements de contexte. C'est en particulier le cas lorsque plusieurs travaux ont la même laxité. Dans ce cas, l'algorithme LLF provoque une situation dite de « task thrashing ». En effet, soit le cas de deux travaux (notons a et b) avec la même laxité (*laxity-tie*) : si on choisit d'exécuter a alors la priorité de b augmente et dépasse donc la priorité de a, donc b préempte a et la situation inverse se produit alors. La figure 1.4 montre ce phénomène avec deux tâches quelconques et une mise à jour des priorités toutes les unités de temps. Bien évidemment, plus le nombre de tâches est important, plus ce problème risque de survenir.

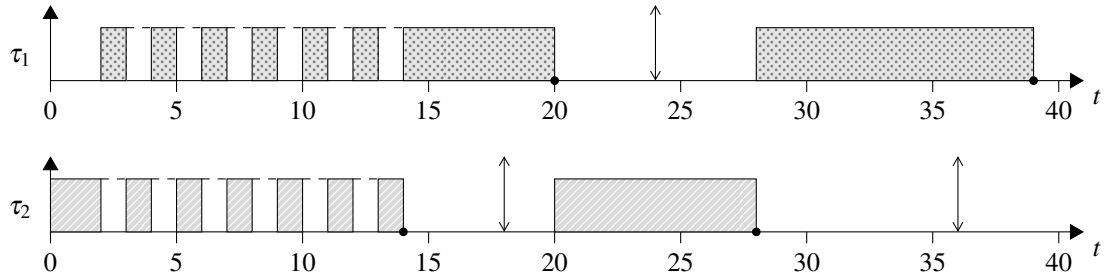


FIGURE 1.4 – Exemple du phénomène de « task thrashing » entre deux tâches engendré par LLF avec un quantum de 1. Tâches : $\tau_1(C_1 = 12, D_1 = T_1 = 24), \tau_2(C_2 = 8, D_2 = T_2 = 18)$.

L'algorithme Modified-Least-Laxity-First (MLLF) tente de résoudre ce problème tout en gardant son optimalité [OY98]. L'idée principale est d'exécuter consécutivement les travaux dont la laxité est identique. Contrairement à LLF qui exécute systématiquement le travail ayant la plus faible laxité, MLLF permet ce qu'on appelle une inversion de laxité.

Soit L_a la laxité la plus faible et D_{min} l'échéance relative du travail ayant la seconde plus faible laxité. L'algorithme MLLF provoque un réordonnancement lorsque l'une des conditions suivante est satisfaite :

- Lors de l'activation ou la terminaison d'un travail.
- Après $L_a - D_{min}$ unités de temps depuis la dernière décision d'ordonnancement.

Les auteurs de cette variante annoncent une réduction par deux du nombre de préemptions lorsque la charge devient maximale, d'après leurs simulations.

L'algorithme Enhanced Least-Laxity-First (ELLF), publié l'année suivante, tente aussi de résoudre le problème de *trashing* [HGT99]. Pour cela, ELLF détecte les cas de *laxity-tie* et exclut toutes les tâches à l'exception de celle ayant l'échéance la plus proche pour la laisser s'exécuter sans problème de *thrashing*. Cette tâche peut cependant se faire préempter par une tâche ayant une laxité inférieure aux tâches exclues. En cas de préemption ou lors de la fin de l'exécution de la tâche, l'algorithme reprend normalement. En pratique, les auteurs de ELLF indiquent que les résultats sont similaires à ceux de MLLF.

1.3 Ordonnancement multiprocesseur

Cette partie s'intéresse aux politiques d'ordonnancement capables d'ordonnancer les tâches sur plusieurs processeurs (ou cœurs). **Les politiques présentées et les résultats associés se limitent au cas des processeurs identiques pour des raisons de simplicité**, cependant, certains algorithmes sont capables de tirer profit d'architectures avec des processeurs différents [Fun04, YA14].

L'ordonnancement monoprocesseur vise à résoudre le problème d'allocation temporelle du processeur aux tâches. L'ordonnancement multiprocesseur rajoute en plus un problème d'allocation spatiale, c'est-à-dire quel processeur utiliser.

La notion de migration, qui consiste en un changement de processeur exécutant la tâche, apparaît alors. On distingue alors les politiques d'ordonnancement suivant trois degrés de migration possibles [CFH+04] :

1. Aucune migration possible.
2. Migration restreinte aux frontières des travaux (ou migration de tâche).
3. Migration libre : les travaux peuvent migrer pendant leur exécution (migration de travail).

Les premiers travaux sur l'ordonnancement multiprocesseur classaient les algorithmes en deux catégories :

- Ordonnancement par partitionnement : répartition a priori des n tâches sur les m processeurs. L'ordonnancement est réalisé localement sur chaque processeur et les tâches ne peuvent pas migrer d'un processeur à l'autre.
- Ordonnancement global : l'ordonnancement s'applique sur l'ensemble des tâches et processeurs. Par conséquent, la migration d'une tâche d'un processeur à un autre devient possible.

L'intérêt des chercheurs s'est porté au début sur l'ordonnancement par partitionnement car il est simple et permet de réutiliser les résultats des travaux sur l'ordonnancement monoprocesseur. En effet, l'ordonnancement par partitionnement revient à réduire le problème d'ordonnancement sur m processeurs à m problèmes d'ordonnancement monoprocesseur.

L'ordonnancement global a longtemps été mis de côté à cause des faibles taux d'utilisation affichés dans les pires cas ou en raison des difficultés pour prouver l'ordonnançabilité d'un système. Par exemple, Dhall *et al.* ont montré qu'un ensemble de tâches au facteur d'utilisation total proche de 1 peut ne pas être ordonnançable en appliquant globalement les politiques RM ou EDF [DL78], indépendamment du nombre de processeurs disponibles. En effet, soit un ensemble de n tâches synchrones à échéance implicite composé de $n-1$ tâches pour lesquelles $C_i = 2\epsilon$ et $T_i = 1$, et d'une tâche pour laquelle $C = 1$ et $T = 1 + \epsilon$. Le taux total d'utilisation de ce système est égal à $2(n-1)\epsilon + \frac{1}{1+\epsilon}$ ce qui tend vers 1 lorsque ϵ tend vers 0. Les politiques RM et EDF donnent dans cet exemple la priorité la plus faible à la tâche la plus lourde ce qui provoque son dépassement d'échéance si le nombre de processeurs est strictement inférieur au nombre de tâches.

Depuis, de nombreuses politiques d'ordonnancement globales dédiées au cas multiprocesseur ont vu le jour et certaines permettent d'atteindre l'optimalité vis-à-vis de la limite d'utilisation (équation 1.7).

$$u_{sum} \leq m \text{ et } u_{max} \leq 1 \tag{1.7}$$

Il est aussi important de noter que dans la plupart des cas, il n'est pas possible de comparer directement ces politiques [LW82]. Leung *et al.* ont montré par exemple que pour des priorités fixes au niveau des tâches, les approches par partitionnement et globales sont non comparables (Définition 1.6).

Pour pallier les défauts des politiques d'ordonnancement utilisant des approches partitionnées ou globales, il existe des alternatives intermédiaires. La première est l'ordonnancement semi-partitionné qui se base sur une approche partitionnée, mais qui permet à certaines tâches de migrer lorsque le système n'est pas partitionnable. Le clustering et l'ordonnancement hiérarchique sont d'autres approches qui consistent à appliquer un ordonnancement global de sous-ensembles de tâches sur des sous-ensembles de processeurs. Enfin, il existe des approches récentes qu'il est difficile de catégoriser telles que RUN [RLM+11] ou QPS [MLR+14].

1.3.1 Ordonnancement par partitionnement

Le principe de l'ordonnancement par partitionnement est relativement simple dans le cadre de processeurs identiques. Il est aussi nécessaire de connaître, dès la conception du système, l'ensemble des tâches. L'ordonnancement par partitionnement consiste à partitionner (au sens mathématique) les tâches pour affecter chaque sous-ensemble de tâches à un unique processeur. Ensuite, les tâches associées à un même processeur sont ordonnancées selon un ordonnanceur monoprocesseur (Figure 1.5). Généralement, le partitionnement est effectué hors-ligne, en se basant sur les conditions d'ordonnançabilité et les caractéristiques des tâches, en particulier leur facteur d'utilisation.

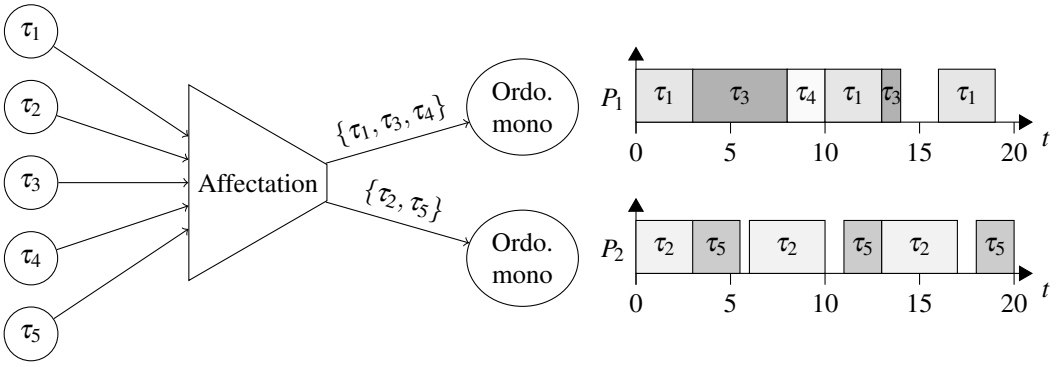


FIGURE 1.5 – Représentation graphique de la stratégie d'ordonnancement par partitionnement

a) Performance dans le pire cas

Les algorithmes de type partitionné souffrent d'un très mauvais taux d'utilisation du système dans le pire des cas. Cette limite d'utilisation du système est : $u_{sum} < \frac{m+1}{2}$. En effet, soit un système de m processeurs avec $m+1$ tâches à ordonnancer. Si chaque tâche a un taux d'utilisation de $0.5+\epsilon$ ($\epsilon > 0$), alors une seule tâche pourra être affectée par processeur et une tâche ne pourra donc pas être affectée à un processeur. L'utilisation totale de ce système est de $(m+1)\cdot(0.5+\epsilon)$, ce qui nous donne bien $lim_{\epsilon \to 0}(m+1)\cdot(0.5+\epsilon) = \frac{m+1}{2}$.

Pour $m > 4$ processeurs, le taux d'utilisation moyen par processeur est en dessous de 60% et il tend ensuite vers 50%.

b) Affectation des tâches

Ce problème d'affectation de tâches à des processeurs peut se ramener à un problème de « bin packing ». Le problème classique de « bin packing » consiste à ranger un ensemble d'éléments caractérisés par leur taille dans un nombre fini de boîtes à la capacité limitée tout en minimisant le nombre de boîtes nécessaires. Dans notre cas, les éléments à ranger sont les tâches et les boîtes sont les processeurs qui disposent d'une capacité limitée au delà de laquelle un ordonnancement correct ne peut pas être assuré. Les conditions suffisantes d'ordonnançabilité sont utilisées pour vérifier que l'algorithme choisi sera en mesure d'ordonnancer correctement les tâches sur chaque processeur.

Le problème d'affectation des tâches se limite à trouver une solution telle que le nombre de boîtes soit inférieur ou égal au nombre de processeurs. Le nombre de partitions de n éléments en k ensembles non vides est donné par le nombre de Stirling de seconde espèce,

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n. \tag{1.8}$$

Le problème d'affectation des tâches autorise à laisser des processeurs non utilisés ce qui nous donne $S'(n,m)$ le nombre de solutions possibles

$$S'(n,m) = \sum_{k=1}^{m} S(n,k) = \sum_{k=1}^{m}\frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n. \tag{1.9}$$

Ce nombre est tel qu'il devient rapidement impossible d'énumérer et vérifier l'ordonnançabilité de toutes les possibilités. Cependant, il est possible de profiter des nombreux travaux

existants pour résoudre le problème de bin packing. Des chercheurs ont montré qu'il était possible de déterminer le nombre minimum de boîtes nécessaires pour une centaine d'éléments en l'espace de quelques secondes [Kor03, Sha04].

Ces méthodes, dites exactes, sont pertinentes pour un nombre de tâches relativement faible (inférieur à la centaine). Au delà, il convient plutôt d'utiliser une heuristique, bien plus rapide, mais qui pourrait éventuellement ne pas trouver de partitionnement valide alors qu'il en existe un. De plus, les heuristiques présentées ci-dessous permettent d'accueillir une nouvelle tâche dans un système en cours d'exécution sans remettre en cause le partitionnement actuel.

c) Heuristiques

L'usage d'une heuristique permet d'affecter un grand ensemble de tâches aux processeurs. Les heuristiques les plus classiques sont « First-Fit », « Best-Fit », « Next-Fit » et « Worst-Fit ». Ces heuristiques sont généralement couplées à un tri des tâches par ordre décroissant selon leur taux d'utilisation. La complexité de ces algorithmes est de l'ordre de $\mathcal{O}(n \times m)$, le tri des tâches est de l'ordre de $\mathcal{O}(n \cdot log(n))$ et ne change donc pas vraiment la complexité de l'heuristique. Ces heuristiques permettent généralement d'obtenir des résultats proches de la solution optimale [LL85, OB98]. Coffman *et al.* ont étudié en détail les performances des heuristiques pour le bin packing [CGJ97]. D'autres heuristiques existent mais ne seront pas présentées ici.

First-Fit L'idée principale de l'algorithme First-Fit est, comme son nom l'indique, d'affecter chaque tâche au premier processeur trouvé tel que l'ordonnanceur local peut l'ordonnancer avec les tâches déjà affectées. Pour chaque tâche, on reprend la recherche depuis le début de la liste des processeurs.

Next-Fit L'algorithme Next-Fit est similaire au First-Fit à la différence près que l'on ne reprend pas la recherche depuis le début à chaque tâche. La recherche d'un processeur capable d'accueillir la tâche reprend à partir du processeur sur lequel on a affecté la tâche précédente.

Best-Fit L'algorithme Best-Fit choisit le processeur disposant de la plus petite capacité disponible et capable d'accepter la tâche tout en restant ordonnançable. En pratique, c'est équivalent au First-Fit, mais en triant les processeurs par capacité. Si la liste des processeurs est maintenue dans un tas, la complexité supplémentaire pour maintenir la liste triée est de l'ordre de $\mathcal{O}(m \cdot log(m))$ ce qui ne change pas significativement la complexité totale de l'heuristique.

Worst-Fit Exactement identique au Best-Fit, mais en choisissant le processeur disposant de la plus grande capacité disponible et capable d'accepter la tâche tout en restant ordonnançable. Cet algorithme peut aussi être vu comme un algorithme d'équilibrage de charge.

1.3.2 Ordonnancement global

La stratégie visant à partitionner les tâches pour les affecter statiquement à des processeurs montre rapidement certaines limites dues à l'impossibilité de migrer pour les tâches. Par exemple, un système composé de trois tâches identiques avec $C = 2$, $T = 3$, $D = T$ et deux processeurs, n'est pas ordonnançable par partitionnement alors qu'en autorisant l'une des

tâches à s'exécuter sur un processeur puis l'autre, on voit que le système est ordonnançable (voir Figure 1.6).

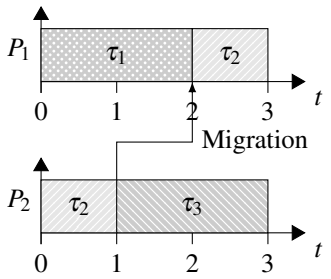


FIGURE 1.6 – Exemple de système de tâches non ordonnançable par une politique par partitionnement, mais ordonnançable avec migration (ici, politique EDZL).

La stratégie d'ordonnancement global n'emploie qu'un seul ordonnanceur et les tâches ne sont pas destinées à un processeur en particulier (voir Figure 1.7), les migrations étant autorisées. Puisque l'ordonnancement global permet de supprimer la contrainte de non-migration, il semble logique que la classe de problèmes ordonnançables soit plus grande. Malheureusement, il a été montré qu'appliquer RM ou EDF de manière globale, ne permet pas d'obtenir des conditions suffisantes d'ordonnançabilité plus avantageuses qu'un ordonnancement par partitionnement [DL78]. C'est pour cette raison que de nouveaux algorithmes dédiés à l'ordonnancement multiprocesseur ont vu le jour (voir parties 1.4 et 1.5).

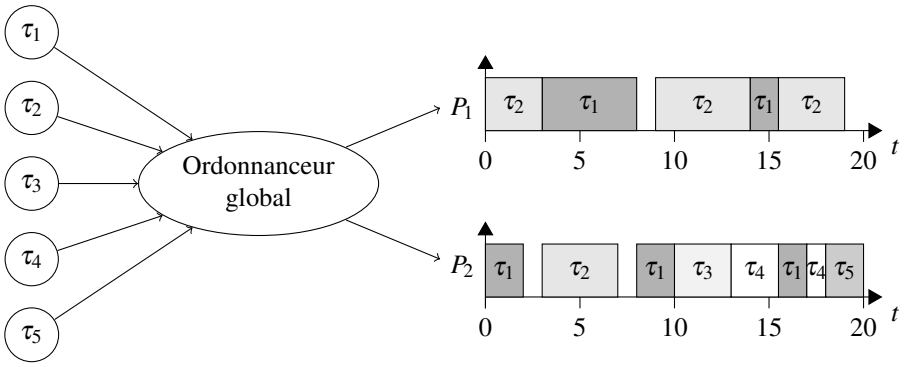


FIGURE 1.7 – Représentation graphique de la stratégie d'ordonnancement global

1.3.3 Ordonnancement semi-partitionné

L'ordonnancement par partitionnement ne permet pas d'ordonnancer certains systèmes à cause de l'impossibilité d'effectuer une migration de tâches d'un processeur à un autre. À l'opposé, l'ordonnancement global offre une totale liberté dans la migration des tâches, et c'est cela qui lui permet d'atteindre l'optimalité. Cependant, cette flexibilité peut avoir un coût à l'exécution et il est donc souhaitable de limiter le nombre de migrations pour les mêmes raisons qui ont poussé les chercheurs à limiter le nombre de préemptions [DA05].

Les algorithmes semi-partitionnés se situent entre ces deux extrêmes et proposent d'autoriser une migration contrôlée de certaines tâches. Pour cela, ils s'inspirent d'algorithmes de partitionnement et permettent à des tâches de migrer successivement sur plusieurs processeurs. Ils peuvent donc être vus comme une amélioration des algorithmes partitionnés en ajoutant plus de flexibilité. La frontière entre ordonnancement global et ordonnancement semi-partitionné est parfois mince comme nous le verrons dans la partie 1.6.1.

Les principaux algorithmes semi-partitionnés ont été mis en œuvre dans LITMUS^{RT} [CLB+06] et les résultats indiquent que les politiques semi-partitionnées sont intéressantes en pratique et offrent, en outre, un bon compromis entre nombre de préemptions-migrations et ordonnançabilité [BBA11] lorsqu'on les compare aux autres politiques.

1.3.4 Autres approches

L'ordonnancement par « clusters », ou « clustering », est une autre forme d'approche intermédiaire entre l'ordonnancement partitionné et l'ordonnancement global. Les processeurs physiques du système sont regroupés en clusters. Un cluster est donc un regroupement de processeurs. Le clustering consiste à affecter les tâches aux clusters et à appliquer un ordonnancement global sur chaque cluster. De plus, l'approche par clustering peut être divisée en deux catégories : le clustering physique et le clustering virtuel.

Dans le cas d'un clustering physique, les processeurs sont associés aux clusters de façon définitive [CAB07] (voir Figure 1.8). Ainsi, seul un ordonnancement des tâches au sein de chaque cluster est nécessaire. L'intérêt de cette approche est multiple. Cela permet de limiter la migration de certaines tâches sur certains groupes de processeurs (ex : sur des cœurs partageant des caches), mais aussi d'augmenter en pratique le nombre de systèmes ordonnançables par rapport à une approche partitionnée tout en limitant le nombre de migrations.

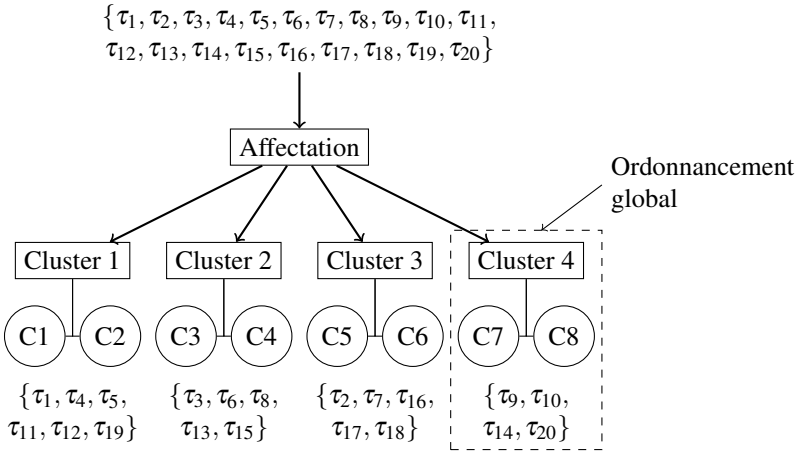


FIGURE 1.8 – Exemple de clustering physique sur une architecture à 8 cœurs pour l'ordonnancement de 20 tâches.

Le clustering virtuel offre une approche plus générale en permettant de changer dynamiquement l'association des processeurs aux clusters [SEL08]. En contrepartie, cela nécessite un algorithme d'affectation des clusters sur les processeurs.

Le clustering ne sera pas étudié ici, en revanche, certaines politiques utilisent des notions que l'on peut rattacher au clustering. C'est par exemple le cas de EKG qui peut effectuer des regroupements de processeurs tels qu'aucune tâche ne peut migrer entre deux groupes différents de processeurs. Les politiques RUN et NPS-F empruntent elles aussi des notions similaires. Ainsi, il n'est pas toujours aisé d'étiqueter certaines politiques qui mélangent des principes issus de stratégies très différentes.

1.4 Généralisation des algorithmes monoprocesseurs

Dans cette partie, nous nous intéressons à l'ordonnancement multiprocesseur global, par la généralisation des algorithmes monoprocesseurs. La généralisation des techniques d'ordonnancement monoprocesseur telles que RM et EDF au cas multiprocesseur consiste à exécuter les m (au plus) travaux ayant la plus grande priorité sur les m processeurs. Cependant, la résolution des compétitions entre travaux de même priorité peut alors poser problème. Il a été montré que la technique de résolution a de l'influence sur l'ordonnançabilité [GFB02]. Par la suite, les politiques qui sont généralisées au cas multiprocesseur par une approche globale seront notées « G- » suivi du nom de la politique (exemple : G-RM et G-EDF).

Les tests disponibles pour ces politiques ne permettent pas d'utiliser pleinement les processeurs tout en garantissant l'ordonnançabilité. Ces tests d'ordonnançabilité [ABJ01, BG03a, BCL05b, SB02, GFB03] dépendent principalement des grandeurs u_{sum} et u_{max}. En outre, dans la plupart des tests, la présence d'une tâche lourde empêche de conclure sur l'ordonnançabilité. Ce résultat va permettre la création de nouvelles politiques, présentées ci-dessous et qui visent à lever l'effet Dhall.

1.4.1 Algorithmes RM-US[ξ], EDF-US[ξ], EDF$^{(k)}$ et fpEDF

Les politiques RM-US[ξ] [ABJ01] et EDF-US[ξ] [SB02] dérivent respectivement des algorithmes G-RM et G-EDF. Partant de l'analyse des critères d'ordonnançabilité des politiques RM et EDF, ces nouvelles politiques proposent d'ajouter un seuil sur le taux d'utilisation au delà duquel les tâches sont considérées à priorité maximale (avec résolution arbitraire des conflits). Ceci permet d'obtenir de meilleures conditions suffisantes d'ordonnançabilité en visant en particulier la limitation liée à u_{max}.

Autrement dit, ces politiques classent les tâches en deux catégories :

- si $u_i > \xi$, alors τ_i est une tâche lourde et sa priorité ainsi que celle de ses travaux est maximale ;
- sinon, τ_i est une tâche légère et le calcul de la priorité reste inchangé par rapport à G-RM ou G-EDF.

L'algorithme EDF$^{(k)}$ a une approche similaire et propose de fixer une priorité maximale aux $k-1$ tâches ayant les plus grands taux d'utilisation du système [GFB03]. Cette approche permet donc théoriquement de choisir la valeur k telle que EDF$^{(k)}$ se comporte comme EDF-US[ξ]. L'algorithme PriD complète EDF$^{(k)}$ en déterminant automatiquement la valeur du paramètre k.

Enfin, l'algorithme fpEDF donne une priorité maximale aux tâches ayant un taux d'utilisation supérieur à $\frac{1}{2}$ en se limitant aux $m-1$ premières tâches par ordre décroissant de taux d'utilisation [Bar04]. On rappelle que m représente le nombre de processeurs.

1.4.2 Algorithme Global-Fair Lateness

L'algorithme *Global-Fair Lateness* (G-FL), développé pour des systèmes temps réel souples, propose une modification mineure de G-EDF afin de réduire le retard maximal des travaux [EA12]. Le retard d'un travail est défini par la différence entre la date de fin du

travail et l'échéance. En pratique, cette modification améliore aussi le nombre de systèmes correctement ordonnancés.

La priorité d'un travail selon la politique EDF correspond à son échéance absolue ($d_{i,j}$) alors que G-FL utilise la priorité suivante :

$$d_{i,j} - \frac{m-1}{m} C_i \tag{1.10}$$

Les auteurs affirment que pour des systèmes temps réel souples, l'algorithme G-FL devrait remplacer G-EDF.

1.4.3 Algorithmes d'ordonnancement à laxité nulle (*Zero Laxity*)

Les politiques RMZL, FPZL et EDZL sont des algorithmes à priorité dynamique, basés sur les algorithmes respectifs RM, FP et EDF, et intégrant la notion de laxité. Pour rappel, la laxité est la marge temporelle d'un travail pendant lequel il peut ne pas être exécuté sans manquer son échéance (voir Figure 1.3). Lorsqu'un travail devient à laxité nulle, cela signifie que s'il n'est pas immédiatement exécuté et ce sans interruption, alors il ne pourra pas respecter son échéance.

Le principe de ces algorithmes « Zero Laxity » est d'attribuer une priorité maximale aux travaux de laxité nulle afin d'éviter un non respect d'échéance et d'ordonnancer selon RM, FP ou EDF sinon.

Lorsqu'un système est ordonnançable par G-EDF, alors G-EDF et EDZL se comportent de manière identique si les travaux des tâches utilisent leur WCET. De plus, il existe des exemples de cas où un système de tâches est ordonnançable par EDZL et ne l'est pas par G-EDF (voir figure 1.9). Ainsi, EDZL domine G-EDF [PHK+05].

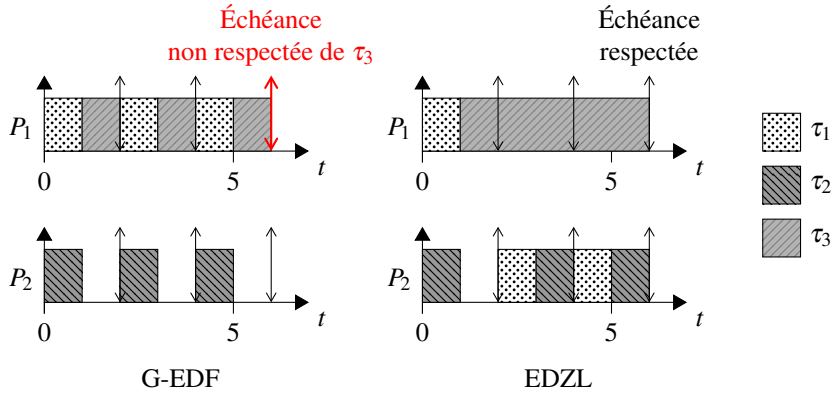


FIGURE 1.9 – Exemple où EDZL permet un ordonnancement alors que G-EDF est en échec. $\tau_1 : (C_1 = 1, T_1 = D_1 = 2), \tau_2 : (C_2 = 1, T_2 = D_2 = 2), \tau_3 : (C_3 = 5, T_3 = D_3 = 6)$

Park *et al.* montrent par simulation que EDZL permet aussi d'ordonnancer un nombre plus important de systèmes que les variantes de G-EDF, EDF-US [SB02] et fpEDF [Bar04]. De plus, le nombre de préemptions reste similaire à G-EDF.

L'algorithme EDCL (pour *Earliest Deadline Critical Laxity*) est une variante de EDZL qui ne change la priorité des travaux ayant atteint une certaine laxité qu'au moment de l'activation et la terminaison de travaux [KY08a]. Ceci permet de réduire le nombre de

préemptions en l'absence de nouveaux évènements. Le nombre de systèmes ordonnançables est plus faible qu'avec EDZL mais les auteurs annoncent que les résultats sont bons.

Davis *et al.* ont montré que les ensembles de tâches ordonnançables selon les tests pour FPZL sont plus nombreux que ceux ordonnançables selon les tests pour EDZL [DK12].

1.4.4 Algorithme Global-Least Laxity First

L'algorithme Least Laxity First, ainsi que sa variante MLLF [1], s'adaptent facilement à une architecture multiprocesseur. Pour rappel, LLF rend prioritaire les travaux ayant la laxité la plus faible, soit les travaux ayant le moins de flexibilité. Tout comme EDZL, G-LLF rend prioritaire les tâches dont la laxité devient nulle ce qui lui permet d'éviter un certain nombre d'échecs à l'ordonnancement.

Cette stratégie semble donner de bons résultats et des travaux récents ont montré que les tests d'ordonnançabilité pour G-LLF donnent un plus grand ensemble de systèmes ordonnançables que ceux de EDZL [LES10]. Cependant, la principale critique émise concerne le nombre important de préemptions et migrations engendré par G-LLF qui pourrait compromettre sa mise en pratique.

1.4.5 Algorithme U-EDF

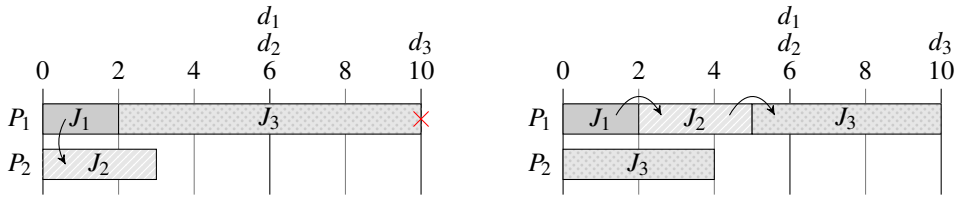
L'algorithme U-EDF est un ordonnanceur multiprocesseur optimal pour des tâches sporadiques à échéances implicites [NBN+12, NFG12]. Les auteurs indiquent que U-EDF n'est pas basé sur la notion d'équité (*fairness*) (voir partie 1.5), cependant, du temps est réservé sur les processeurs en fonction des taux d'utilisation des travaux. Des évènements qui correspondent à des fins de budgets virtuels sont introduits pour permettre d'atteindre l'optimalité. Notons que l'apparition de cet algorithme s'est faite après les travaux sur les algorithmes équitables et les approches hybrides dans la perspective de réduire le nombre de préemptions et migrations par rapport à ces derniers.

Les auteurs présentent U-EDF comme étant une généralisation de EDF « horizontale » contrairement à G-EDF qu'ils qualifient de « verticale ». Pour illustrer cette notion, les auteurs proposent l'exemple d'un système composé de deux processeurs et trois travaux $J_1 = (2, 6)$, $J_2 = (3, 6)$ et $J_3 = (9, 10)$ avec (c, d) qui représente une exécution de c unités de temps et une échéance à l'instant d. La figure 1.10a illustre la séquence obtenue pour G-EDF où l'on peut constater que l'échéance du travail J_3 n'est pas respectée. Cet échec est causé par le fait que G-EDF favorise l'exécution au plus tôt des travaux sans anticiper l'impact sur les futurs exécutions.

La figure 1.10b montre une possible généralisation « horizontale » de EDF. Elle consiste à ordonnancer les travaux avec les plus grandes priorités (c'est-à-dire avec l'échéance la plus proche) de manière séquentielle sur un premier processeur et à ne commencer l'allocation sur un autre processeur que si le premier processeur ne peut pas assurer l'ordonnancement de tous les travaux. Dans l'exemple présenté, le travail J_3 sera exécuté sur le premier et le deuxième processeur car le premier processeur ne permet pas de traiter l'ensemble des travaux. Les auteurs qualifient cette approche d'« horizontale » car l'algorithme va d'abord

1. MLLF est capable de gérer une architecture constituée de m processeurs en considérant L_a comme la laxité la plus importante parmi les m laxités les plus faibles et D_{min} l'échéance relative du travail avec la $m + 1$-ème laxité la plus faible.

tenter de « remplir » un processeur au maximum avant d'allouer un travail à un nouveau processeur. Malheureusement, la construction de l'ordonnancement telle qui vient d'être présentée est faite hors-ligne. L'algorithme U-EDF propose une solution pour construire un tel ordonnancement de manière en-ligne.



(a) Généralisation « verticale ». (b) Généralisation « horizontale ».

FIGURE 1.10 – Généralisations de EDF verticale et horizontale [NBN+12].

L'algorithme U-EDF se divise en deux phases :

1. Lorsqu'un nouveau travail devient actif, les travaux actifs sont assignés aux processeurs en utilisant l'approche horizontale telle que décrite ci-dessus. Pendant cette phase, du temps est provisionné sur chaque processeur en fonction des taux d'utilisation pour les tâches inactives afin de garantir leur exécution avant leur échéance.
2. La seconde phase consiste à ordonnancer les travaux selon l'algorithme EDF-D en fonction des budgets calculés dans la première phase.

L'algorithme EDF-D est une variante de EDF qui ordonnance les tâches *éligibles* sur les processeurs. Une tâche est dite éligible sur un processeur si elle a du temps réservé sur ce processeur et si elle n'a pas été choisie pour s'exécuter sur un autre processeur d'indice inférieur (les processeurs étant homogènes, les processeurs sont simplement ordonnés par un indice). Ensuite EDF-D sélectionne pour chaque processeur la tâche éligible avec la plus petite échéance comme étant celle devant s'exécuter.

La difficulté de cette approche réside dans la détermination des budgets virtuels à réserver sur les processeurs lors de l'arrivée d'un nouveau travail. L'algorithme ne pouvant simplement être résumé, un lecteur intéressé par les détails est invité à lire l'article [NBN+12].

1.5 Ordonnancement global dit équitable

Dans leur paragraphe « Why Greedy Schedulers Fail », Levin *et al.* expliquent pourquoi les approches qui généralisent simplement les ordonnanceurs monoprocesseurs ne permettent pas d'atteindre l'optimalité dans le cas multiprocesseur [LFS+10]. En effet, contrairement au cas monoprocesseur, il peut être nécessaire de prendre des décisions sur l'ordonnancement alors qu'il n'y a pas d'événement « habituel » donnant lieu à un réordonnancement, c'est-à-dire, autre qu'une fin de tâche, une activation d'une nouvelle tâche ou une laxité nulle.

L'algorithme U-EDF est capable d'ordonnancer de manière optimale des systèmes composés de tâches à échéance implicite par l'introduction d'événements déterminés à partir des durées d'exécution et périodes des tâches. La création de cet algorithme a en réalité été motivé par l'existence des algorithmes équitables et qui sont présentés dans cette section.

1.5.1 Ordonnancement PFair

Les algorithmes de la famille PFair (*Proportionate Fair*) ont une approche différente de l'ordonnancement et sont destinés aux architectures multiprocesseurs [BCPV96]. La différence avec les algorithmes classiques d'ordonnancement est que les algorithmes PFair imposent explicitement l'exécution des tâches à un taux régulier.

L'objectif d'un algorithme PFair est de se rapprocher d'un ordonnancement idéal. Un ordonnancement est dit idéal, ou équitable, lorsque chaque tâche reçoit exactement $u_i \cdot t$ unités de temps processeur dans l'intervalle $[0, t[$. Bien sûr, un tel ordonnancement est impossible dans le cas discret, et un algorithme PFair va simplement essayer de s'en rapprocher.

a) Modélisation de l'ordonnancement PFair

Discrétisation du temps Les décisions d'ordonnancement sont réalisées à des valeurs de temps entières qui commencent à 0. Le temps est discrétisé en intervalles uniformes appelés des slots de temps où l'intervalle de temps $[t, t+1[$, avec $t \in \mathbb{N}$, correspond au slot t.

Contrainte PFair De façon à modéliser une séquence d'ordonnancement, nous définissons la fonction binaire $S : \tau \times \mathbb{N} \to \{0, 1\}$ qui renvoie 1 lorsqu'une tâche τ_i est ordonnancée sur le slot t et 0 sinon.

Grâce à cette fonction, il est possible d'introduire la notion de décalage en temps (*lag*), qui mesure l'écart entre l'exécution idéale et la séquence construite (Définition 1.12).

Définition 1.12. *Le décalage d'une tâche τ_i à l'instant t se note :*

$$lag(\tau_i, t) = u_i \cdot t - \sum_{l=0}^{t-1} S(\tau_i, l) \tag{1.11}$$

Grâce à cette notion de décalage, il nous est possible de définir la contrainte PFair (Définition 1.13). La figure 1.11 montre l'ordonnancement d'une tâche qui respecte cette contrainte.

Définition 1.13. *Un algorithme est dit PFair lorsque, $\forall t \in \mathbb{N}*, \forall \tau_i \in \tau, |lag(\tau_i, t)| < 1$. Autrement dit, à chaque instant, l'erreur d'exécution est strictement inférieure à un quantum de temps, ou encore que $\sum_{l=0}^{t-1} S(\tau_i, l) = \lfloor u_i \cdot t \rfloor$ ou $\lceil u_i \cdot t \rceil$.*

Pseudo-réveil et pseudo-échéance : La construction d'un ordonnancement PFair passe par la division de chaque tâche en sous-tâches de durée d'exécution un quantum. Une tâche τ_i est ainsi divisée en une séquence infinie de sous-tâches τ_i^j avec τ_i^1 la première sous-tâche. Chaque sous-tâche possède une fenêtre d'exécution possible pour que son exécution soit « PFair » : $[r(\tau_i^j), d(\tau_i^j)]$. On appelle, $r(\tau_i^j)$ la date de pseudo-réveil et $d(\tau_i^j)$ la date de pseudo-échéance. La fenêtre d'exécution de τ_i^j se note $w(\tau_i^j)$ et sa longueur est définie par : $|w(\tau_i^j)| = d(\tau_i^j) - r(\tau_i^j)$.

La contrainte PFair implique [BCPV96] :

$$r(\tau_i^j) = \left\lfloor \frac{j-1}{u_i} \right\rfloor \text{ et } d(\tau_i^j) = \left\lceil \frac{j}{u_i} \right\rceil \tag{1.12}$$

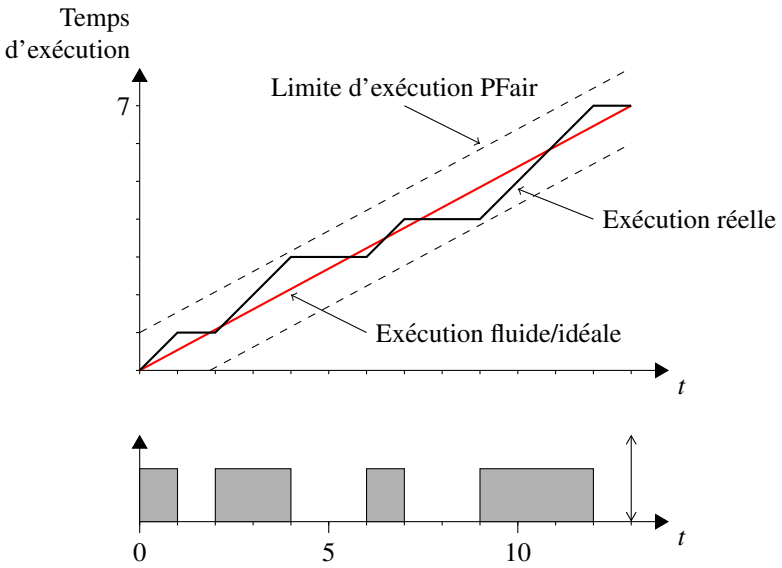


FIGURE 1.11 – Ordonnancement PFair d'une tâche τ_i ($C_i = 7$, $T_i = 13$).

La figure 1.12 montre les fenêtres d'exécution pour l'exemple précédent ($C_i = 7$, $T_i = 13$). On peut constater qu'en effet, chaque sous-tâche s'est bien exécutée à l'intérieur de sa fenêtre d'exécution.

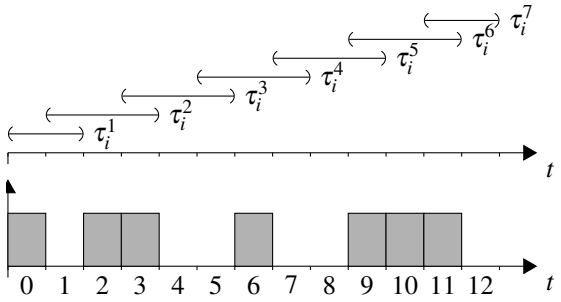


FIGURE 1.12 – Fenêtres d'exécution des sous-tâches de τ_i.

Dans le cadre de l'ordonnancement PFair, une tâche est qualifiée de lourde si et seulement si son facteur d'utilisation est supérieur ou égal à 1/2, sinon il s'agit d'une tâche légère. Il est intéressant de remarquer alors que seule une tâche lourde possède des fenêtres d'exécution de taille 2 [AS99].

b) Optimalité de l'ordonnancement PFair

Supposons un système composé de n tâches périodiques, indépendantes, synchrones et à échéances implicites. Supposons également que les durées d'exécution et les périodes sont des multiples du quantum de temps choisi. Soit une échéance d'une tâche quelconque τ_i qui se produit à l'instant $k \cdot T_i$ ($k \in \mathbb{N}*$). Pour que le système soit ordonnançable, la durée allouée pour la tâche depuis le début de l'exécution du système doit être de $k \cdot C_i$.

D'après la définition 1.13, la tâche τ_i a été exécutée pendant une durée de $\lfloor u_i \cdot k \cdot T_i \rfloor = \lfloor k \cdot C_i \rfloor$ ou $\lceil u_i \cdot k \cdot T_i \rceil = \lceil k \cdot C_i \rceil$. Or $k \cdot C_i$ est entier donc la durée d'exécution de la tâche τ_i est exactement égale à $k \cdot C_i$. Le système est donc ordonnançable.

Un ordonnancement PFair de n tâches périodiques, synchrones et à échéances implicites, sur m processeurs identiques existe si et seulement si la condition 1.13 est respectée [BCPV96].

$$\sum_{i=1}^{n} u_i \leq m \tag{1.13}$$

Bien qu'en théorie un tel ordonnancement est optimal, il convient de nuancer ce résultat en pratique. En effet, les surcoûts d'exécution tels que les prises de décision à chaque quantum, ou encore les nombreuses préemptions et possibles migrations, ne sont absolument pas prises en compte. Et enfin, les processeurs doivent être parfaitement synchronisés entre eux pour respecter les précédences entre sous-tâches d'une même tâche (des travaux visent à adapter PFair pour tenir compte de ce problème de synchronisation [HA05]).

Il existe différents algorithmes de type PFair. Citons d'abord EPDF (*Earliest Pseudo-Deadline First*) qui n'est optimal que pour deux processeurs au maximum. Puis citons les algorithmes PF, PD [BCPV96] et PD^2 [AS99] qui complètent EPDF par l'évaluation et la comparaison de critères supplémentaires.

c) Variantes conservatives

Un ordonnanceur PFair n'est pas conservatif dans la mesure où il peut être amené à ne pas exécuter de tâches pour éviter de prendre trop d'avance. Les politiques dites ER-Fair [AS00a] proposent d'alléger la contrainte (Définition 1.13) sur le décalage par simplement : $lag(\tau_i, t) < 1$. Les avantages sont des temps de réponse plus courts et une mise en œuvre simplifiée.

La politique ER-PD^2 est une adaptation ER-Fair de l'algorithme PD^2. Plus récemment, Kim et Cho ont proposé la politique PL, elle aussi de type ER-Fair, et qui propose d'intégrer la laxité comme critère pour départager les sous-tâches [KC11].

1.5.2 Ordonnancement global DP-Fair

Les algorithmes de la famille de PFair atteignent l'optimalité grâce à de nombreux découpages et à l'ajout d'une contrainte forte sur la fluidité de l'exécution des tâches. Ces nombreux découpages engendrent beaucoup de préemptions et prises de décisions. Dans une optique de réduction de ces désagréments, Zhu *et al.* montrent qu'il n'est pas nécessaire de se rapprocher autant de l'exécution fluide des travaux [ZMM03]. Afin d'assurer l'optimalité, l'important est qu'à chaque échéance, aucune tâche ne soit en retard par rapport à son exécution fluide. La technique BFair, pour « Boundary Fair », est très similaire à la technique DP-Fair mais se base sur des valeurs entières [ZQMM11].

a) Principe

Les techniques DP-Fair, pour « Deadline Partitioning Fair » reposent sur le concept de *Deadline Partitioning* qui consiste à découper le temps en intervalles définis par l'ensemble des échéances des tâches [LFS$^+$10]. À la fin de chaque intervalle, DP-Fair impose aux tâches de ne pas avoir de retard par rapport à leur exécution fluide.

La première étape de l'algorithme est le partitionnement du temps en intervalles définis par les dates d'échéance de l'ensemble des tâches. Plus formellement, et en s'inspirant de la notation utilisée dans [LFS+10], on note $t_0 = 0$ et on pose t_1, t_2, etc., les dates des différentes échéances par ordre chronologique ($t_j < t_{j+1}$). Le $j^{\text{ème}}$ intervalle, noté σ_j, correspond donc à l'intervalle $[t_{j-1}, t_j[$, et on note sa longueur $L_j = t_j - t_{j-1}$. Par la suite, i correspond à l'indice de la tâche τ_i et j à l'indice de l'intervalle σ_j.

Pour chaque intervalle σ_j, l'ensemble des m processeurs met à disposition $m \cdot L_j$ unités temporelles d'exécution. Il convient alors de définir pour chaque tâche, le nombre d'unités temporelles d'exécution qui seront à exécuter dans cet intervalle et de définir l'ordonnancement au sein de l'intervalle.

Deux étapes sont nécessaires pour ce type d'algorithme :

- La dotation temporelle est l'étape où on définit pour chaque tâche τ_i sa durée d'exécution locale pour l'intervalle σ_j.
- La distribution temporelle est la façon de répartir, pour l'ensemble des tâches, les durées d'exécution locales sur les m processeurs sur l'intervalle considéré.

b) Politiques DP-Fair

Il existe plusieurs politiques de type DP-Fair, les différences se jouent principalement sur la façon dont la dotation temporelle et la distribution temporelle sont décidées. Citons en particulier BF [ZMM03] et LLREF [CR06], les premières politiques utilisant cette stratégie.

LLREF affecte une dotation temporelle à chaque tâche τ_i sur l'intervalle σ_j égale à $L_j \cdot u_i$. La distribution temporelle des dotations sur chaque intervalle se base sur la notion de TL-plane. Des évènements de type B et C seront alors générés lors d'évènements de laxité nulle sur l'intervalle ou en cas de fin d'utilisation de la dotation locale.

La figure 1.13 illustre un TL-plane pour l'exécution de 3 tâches sur 2 processeurs dans l'intervalle j. Les travaux disposant de la plus grande dotation locale restante (l_i^j) sont choisis. Lorsque la laxité de la tâche τ_1 devient nulle, un évènement de type C est généré et l'algorithme choisit à nouveau les deux tâches ayant la plus grande dotation locale restante. Lorsque la tâche τ_2 se termine, un évènement de type B est généré et l'algorithme choisit les deux tâches restantes.

La politique LRE-TL se base sur LLREF et réduit de manière significative le nombre de préemptions et migrations en ne choisissant pas les tâches ayant la plus grande dotation locale restante. De plus, LRE-TL supporte également les tâches sporadiques [FN09].

Levin *et al.* ont présenté l'algorithme DP-WRAP qui a pour particularité d'utiliser l'algorithme de McNaughton pour effectuer la distribution temporelle. Cet algorithme est également capable de traiter des tâches sporadiques [LFS+10].

Enfin, NVNLF est une variante *work-conserving* de LLREF qui permet de réduire de façon importante le nombre de préemptions et migrations tout en réduisant aussi les temps de réponse [FKY08]. L'algorithme NVNLF se base sur la notion de TL-plane mais étendue afin de prendre en compte les ressources processeurs non utilisées. NVNLF alloue dans un premier temps le même budget que LLREF, puis, distribue le temps processeur restant aux tâches (non entièrement déjà dotées). Une tâche pourra donc continuer son exécution au delà de son exécution fluide sans pénaliser les autres tâches.

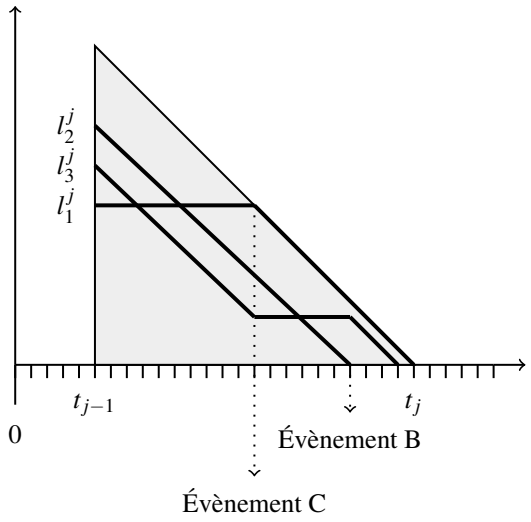


FIGURE 1.13 – Exécution au sein d'un TL-plane sur l'intervalle σ_j des tâches τ_1, τ_2 et τ_3 sur 2 processeurs.

1.6 Approches hybrides

1.6.1 Ordonnancement semi-partitionné

Les politiques d'ordonnancement semi-partitionné visent à améliorer les limites d'utilisation des algorithmes basés sur le partitionnement de tâches dans le cas où le système n'est pas partitionnable. Les premières politiques de ce genre sont EDF-fm publié en 2005 par Anderson et al [ABD05] et EKG, publié en 2006 par Andersson et al [AT06]. Dans le cas de ces deux ordonnanceurs, les tâches sont affectées aux processeurs à l'aide d'une variante de l'algorithme Next-Fit. Les tâches sont affectées au processeur courant et lorsque celui-ci n'a pas assez de place, la tâche est découpée en deux parties. Ainsi, on obtient un partitionnement avec au maximum $m-1$ tâches qui seront amenées à migrer d'un processeur à l'autre.

Afin de réduire le nombre de tâches migrantes, EKG propose de faire des regroupements de processeurs de taille K et d'interdire les migrations en dehors de ces groupes. Les tâches dont le taux d'utilisation dépasse $\frac{k}{k+1}$ sont exécutés sur des processeurs dédiés. Un exemple est donné par la figure 1.14.

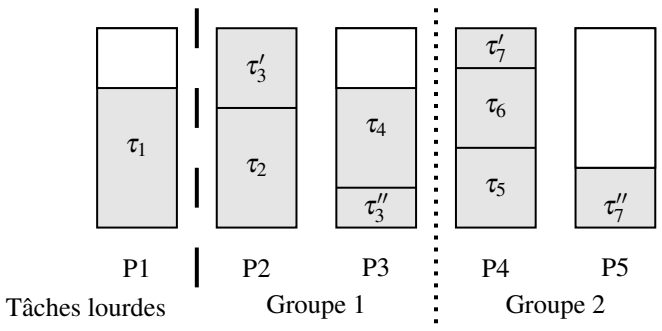


FIGURE 1.14 – Affectation de 7 tâches sur une architecture à 5 processeurs selon EKG avec $K = 2$. $u_1 = 0.7, u_2 = 0.6, u_3 = 0.6, u_4 = 0.5, u_5 = 0.4, u_6 = 0.4, u_7 = 0.5$

Les tâches migrantes s'exécutent selon le principe d'équité au début et en fin d'intervalles

de temps définis par les dates de réveil et d'échéance des tâches d'un même groupe. Les autres tâches sont exécutées avec la politique EDF. EKG n'est optimal que pour des tâches périodiques et pour K égal au nombre de processeurs. Cependant, le nombre de préemptions et migrations augmente fortement dans ce cas, il convient donc de chercher la valeur de K la plus faible permettant d'ordonnancer le système de tâches considéré. EKG a été entendu aux tâches sporadiques [AB08] et aux tâches à échéances arbitraires [ABB08].

L'algorithme EKG assigne les tâches migrantes et non-migrantes en même temps mais d'autres politiques procèdent en deux étapes distinctes : dans un premier temps, les tâches sont affectées selon un algorithme de partitionnement, puis dans un second temps, les tâches qui n'ont pas été affectées sont réparties sur plusieurs processeurs, suivant un second algorithme.

L'algorithme EDHS [KY08d] utilise cette seconde approche. La figure 1.15 montre la technique de partitionnement de EDHS. Au cours de la seconde étape, chaque processeur ne peut accueillir qu'une seule tâche migrante. Il est intéressant de noter que tout système de tâches ordonnançable par partitionnement, l'est aussi par EDHS. Corollairement, EDHS ne provoque de migrations que pour des systèmes qui ne peuvent pas être ordonnancés par partitionnement.

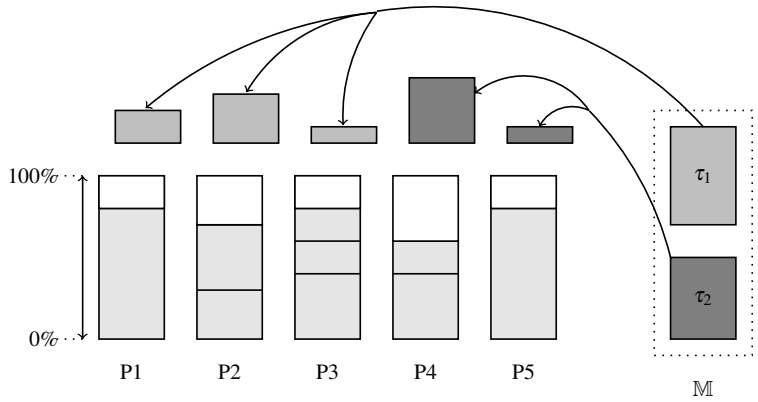


FIGURE 1.15 – Semi-partitionnement selon EDHS.

Les tâches migrantes s'exécutent successivement sur les différents processeurs pour une durée déterminée lors de la phase d'affectation des tâches. Les autres tâches s'exécutent sur leur processeur suivant l'ordonnancement EDF mais en laissant la priorité aux tâches migrantes. Lorsqu'une tâche migrante a épuisé son temps d'exécution sur un processeur, elle continue son exécution immédiatement sur le processeur suivant en préemptant si nécessaire un travail en cours d'exécution.

La politique EDF-WM [KYI09] améliore EDHS en apportant une souplesse supplémentaire sur l'exécution des tâches migrantes. En effet, comme le montre la figure 1.16, il existe des cas où EDHS provoque des erreurs qui auraient pu être évitées par plus de flexibilité. Ainsi, pour chaque portion de tâche migrante, on définit une fenêtre d'exécution possible sur chaque processeur. On obtient alors pour chaque portion de tâche une date d'arrivée et une date d'échéance. Ces « sous-tâches » sont ordonnancées comme les autres tâches selon EDF mais en se basant sur les dates virtuelles d'activation et d'échéance.

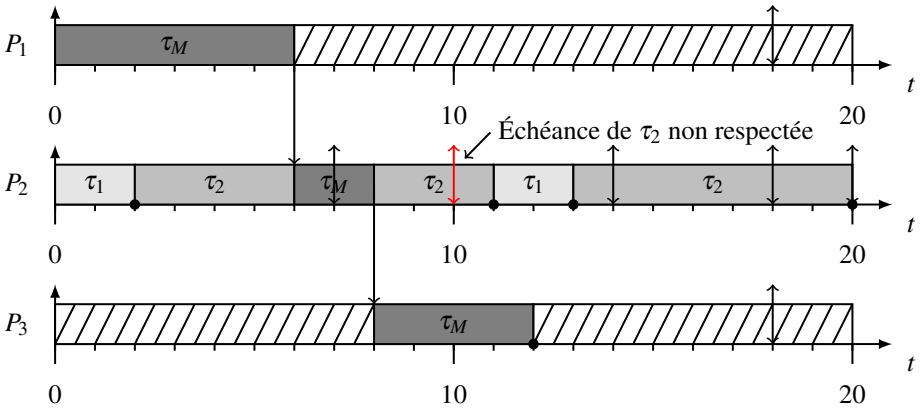


FIGURE 1.16 – Ordonnancement EDHS.

1.6.2 Algorithme RUN

L'algorithme RUN permet d'ordonnancer de manière optimale un ensemble de tâches périodiques indépendantes, synchrones et à échéances implicites sur une architecture multiprocesseur [RLM+11]. L'optimalité est obtenue à l'aide d'une version d'équité proportionnelle dite faible qui a pour avantage de provoquer moins de préemptions et de migrations. L'algorithme SPRINT est une modification de RUN pour l'ordonnancement de tâches sporadiques avec échéances implicites [BNVT14].

La politique RUN peut se diviser en deux grandes étapes. La première étape, effectuée hors-ligne, permet de construire un arbre constitué des tâches à ordonnancer au niveau des feuilles et de *serveurs* au niveau des nœuds. La seconde étape, effectuée en-ligne, consiste à exécuter l'ordonnanceur à partir de cet arbre en suivant un ensemble de règles basées sur EDF et la notion de dualité.

a) Partie hors-ligne

La construction de l'arbre se fait à partir de tâches à ordonnancer et de deux opérations : *pack* et *dual*. On suppose que le taux d'utilisation du système est égal au nombre de processeurs. Il est en effet simple d'ajouter des tâches *idle* pendant la construction de l'arbre. Avant tout, il convient d'introduire la notion de tâche *dual* (définition 1.14).

Définition 1.14. *La tâche dual d'une tâche τ est notée τ^*. La tâche dual τ^* ne s'exécute que pendant les temps d'inactivité de la tâche τ et réciproquement. Celle-ci partage les mêmes échéances que τ mais une durée d'exécution complémentaire vis-à-vis de sa période.*

Soit un ensemble de $m + k$ tâches et dont la somme des taux d'utilisation est égale à m. Les auteurs montrent alors que l'ordonnancement de cet ensemble de tâches sur m processeurs est possible si et seulement s'il est possible d'ordonnancer les tâches duals sur k processeurs. La figure 1.17 illustre ceci à travers un exemple simple.

L'opération pack consiste à regrouper des tâches ou serveurs avec un algorithme tel que Worst-Fit. Les tâches et serveurs sont regroupés de telle sorte que l'utilisation totale ne dépasse pas un.

Les opérations de dual et pack sont enchainées jusqu'à convergence au nœud racine. Les nœuds sont appelés serveurs. Les nœuds formés par un regroupement de serveurs sont

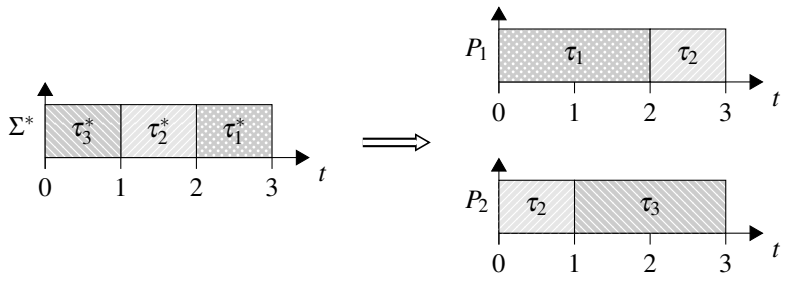


FIGURE 1.17 – Équivalence entre l'ordonnancement des tâches τ_1, τ_2 et τ_3 sur 2 processeurs et leurs tâches duals τ_1^*, τ_2^* et τ_3^* sur 1 processeur. Les tâches ont pour période 3, échéance 3 et durée d'exécution 2.

appelés « Serveur EDF » et les nœuds obtenus par l'opération dual sont appelés « Serveur dual ».

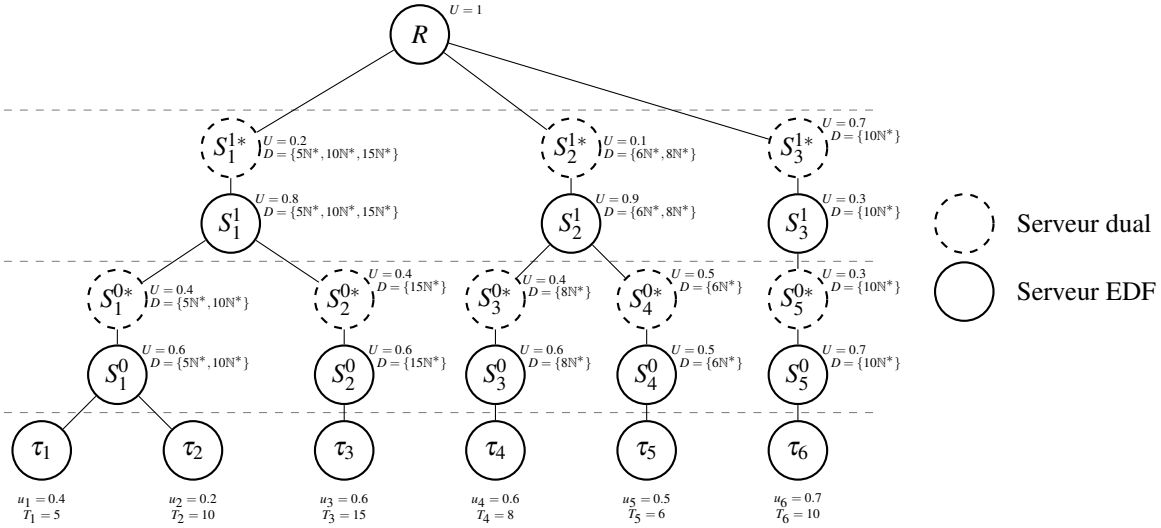


FIGURE 1.18 – Arbre de réduction.

Les échéances des nœuds sont déterminés au moment de la création de l'arbre, en fonction du type de nœud :

- Les échéances d'un « Serveur dual » sont identiques à celles de son nœud fils.
- Les échéances d'un « Serveur EDF » sont l'union des échéances des nœuds fils.

Le taux d'utilisation des nœuds dépend également du type :

- Le taux d'utilisation d'un « Serveur dual » est égal à 1 moins l'utilisation de son nœud fils.
- Le taux d'utilisation d'un « Serveur EDF » est égal à la somme des taux d'utilisation des nœuds fils.

b) Partie en-ligne

Un budget (proportionnel à son facteur d'utilisation) est affecté aux serveurs lors de l'activation d'une de leurs tâches clientes. À chaque activation, fin d'exécution ou lorsqu'un

budget est écoulé, alors il est nécessaire de réordonnancer le système. Afin de déterminer l'ensemble des tâches à exécuter, l'arbre est parcouru depuis la racine jusqu'aux feuilles en appliquant les deux règles suivantes :

Règle 1. *Si un serveur EDF s'exécute, alors le nœud fils ayant l'échéance la plus proche s'exécute. Si un serveur EDF ne s'exécute pas, alors aucun de ses fils ne s'exécute.*

Règle 2. *Si un serveur dual s'exécute, alors son fils ne s'exécute pas et réciproquement.*

1.7 Conclusion

Ce chapitre a débuté par une présentation du modèle de tâches temps réel considéré et des politiques d'ordonnancement monoprocesseurs. Le tableau 1.2 recense les principales politiques monoprocesseurs disponibles. Nous avons ensuite présenté les problématiques liées à l'ordonnancement multiprocesseur et les différentes stratégies mises en œuvre.

Nom	Type	Abréviation
Fixed Priority [LL73]	Priorité fixe tâches	FP
Rate Monotonic [LL73]	Priorité fixe tâches	RM
Deadline Monotonic [LW82]	Priorité fixe tâches	DM
Optimal Priority Assignment [Aud91]	Priorité fixe tâches	OPA
Earliest Deadline First [LL73]	Priorité fixe travaux	EDF
Least Laxity First [Mok83]	Priorité dynamique	LLF
Modified Least Laxity First [OY98]	Priorité dynamique	MLLF
Enhanced Least Laxity First [HGT99]	Priorité dynamique	ELLF

Tableau 1.2 – Principaux ordonnanceurs monoprocesseurs.

Au cours de ces vingts dernières années, de très nombreux algorithmes d'ordonnancement multiprocesseur ont été proposés. Il en résulte un nombre considérable d'ordonnanceurs tel qu'il devient difficile de tous les répertorier. Nous avons recensé près de cinquante politiques multiprocesseurs (voir tableau 1.3).

Aucune politique n'a jusqu'à présent réussi à véritablement s'imposer face aux autres. Les raisons sont multiples et sont liées à des limites d'utilisation trop faibles, une mise en œuvre trop complexe, des hypothèses pas assez réalistes ou encore des surcoûts à l'exécution trop importants. Cependant, au cours de ces dernières années, les politiques RUN et U-EDF ont toutes deux montré qu'il était possible de corriger en partie les défauts des politiques à base de *fairness* ou de semi-partitionnement tout en étant optimal.

Il ne faut pas oublier pour autant les politiques moins complexes telles que les variantes de EDF qui permettent d'ordonnancer de nombreux systèmes et ont l'avantage d'être plus simples à mettre en œuvre tout en pouvant être combinées avec des techniques de clustering lorsque le nombre de processeurs devient trop grand. Enfin, les ordonnanceurs à priorité fixe ne sont pas à exclure car leur fonctionnement plus simple permet d'étudier des systèmes plus complexes avec par exemple des dépendances entre les tâches.

Nom	Type	Abréviation
Partioned FP (+ algorithme d'affectation)	Partitionné	P-FP
Partioned EDF (+ algorithme d'affectation)	Partitionné	P-EDF
Physical Clustering (+ une politique globale)	Clustering	C-politique
VC-IDT [ESL09]	Virtual Clustering	VC-IDT
Global-EDF	Global	G-EDF
Global-RM	Global	G-RM
Earliest Deadline Zero Laxity [Lee94]	Global	EDZL
Earliest Deadline Critical Laxity [KY08a]	Global	EDCL
FPZL, FPCL, FPSL [DK12]	Global	FPZL, FPCL, FPSL
Least Laxity First [Mok83]	Global	G-LLF
Modified Least Laxity First [OY98]	Global	G-MLLF
Fixed Priority with adaptiveTkC [AJ00]	Global	adaptiveTkC
RM-US [ABJ01]	Global	RM-US
DM with Density Separation [BCL05b]	Global	DM-DS
Slack Monotonic-US [And08]	Global	SM-US
EDF-US [SB02]	Global	EDF-US
Priority-Driven / EDF$^{(k)}$ [GFB03]	Global	PriD
Global-Fair Lateness [EA12]	Global	G-FL
U-EDF [NBN$^+$12]	Global	U-EDF
Earliest Pseudo-Deadline First [AS00b]	PFair	EPDF
PF [BCPV96]	PFair	PF
SA [KS97]	PFair	SA
PD [BGP95]	PFair	PD
PD^2 [AS99]	PFair	PD^2
ER-PD^2 [AS00a]	ER-Fair	ER-PD^2
PL (Pseudo-Laxity) [KC11]	ER-Fair	PL
BF [ZMM03, ZQMM11]	BFair	BF
BF2 [NSG$^+$14]	BFair	BF2
LLREF (ou LNREF) [CR06]	DP-Fair	LLREF
NVNLF [FKY08]	DP-Fair	NVNLF
LRE-TL [FN09]	DP-Fair	LRE-TL
DP-WRAP [LFS$^+$10]	DP-Fair	DP-WRAP
EDF-fm [ABD05]	Semi-partitionné	EDF-fm
EKG [AT06]	Semi-partitionné	EKG
Ehd2-SIP ou EDDHP [KY07]	Semi-partitionné	Ehd2-SIP
EDDP [KY08b]	Semi-partitionné	EDDP
RMDP [KY08c]	Semi-partitionné	RMDP
DM with Priority Migration [KY09]	Semi-partitionné	DM-PM
EDHS [KY08d]	Semi-partitionné	EDHS
EDF Window constrained Migration [KYI09]	Semi-partitionné	EDF-WM
PDMS_HPTS_DS [LRL09]	Semi-partitionné	PDMS_HPTS_DS
SPA2 [GSYY10]	Semi-partitionné	SPA2
HSP [FQ12]	Semi-partitionné	HSP
EDF with Bandwith Reservation [ML10]	Semi-partitionné	EDF-BR
EDF-os [AEDC14]	Semi-partitionné	EDF-os
NPS-F [BA09]	Semi-partitionné	NPS-F
Carousel-EDF [SSTB13]	Hybride	Carousel-EDF
RUN [RLM$^+$11]	Hybride	RUN
Quasi-Partitioning Scheduler [MLR$^+$14]	Hybride	QPS

Tableau 1.3 – Liste non exhaustive d'ordonnanceurs multiprocesseurs.

CHAPITRE 2

Principe et fonctionnement des mémoires caches

Nous avons vu au chapitre précédent que de très nombreuses politiques d'ordonnancement multiprocesseur ont été proposées dans la littérature afin d'optimiser les performances du point de vue de l'ordonnançabilité. Les algorithmes les plus récents annoncent qu'ils permettent de réduire les artefacts temporels dus au support matériel d'exécution, en particulier en réduisant le nombre de préemptions et de migrations. Qu'en est-il réellement de ces effets et quels en sont les causes ? Pour répondre à ces questions, nous nous intéressons dans ce nouveau chapitre aux mécanismes présents dans un processeur et pouvant avoir un impact sur l'ordonnancement. Parmi ceux-ci, nous avons décidé de nous concentrer sur les mémoires caches pour deux raisons principales. La première est que ce mécanisme a un impact important sur la vitesse d'exécution des programmes informatiques et donc indirectement sur les séquences d'ordonnancement des travaux. La seconde raison est que l'efficacité des caches dépend en partie des décisions d'ordonnancement. Par exemple, les préemptions dues à une décision d'ordonnancement peuvent perturber les caches, tout comme le choix des tâches qui s'exécutent en parallèle dans le cas multiprocesseur a un impact sur les mémoires caches partagées.

La première partie de ce chapitre vise à fournir les bases nécessaires pour comprendre le fonctionnement d'un processeur pour l'exécution d'un programme et quelles sont les sources d'optimisation possibles. Une explication du fonctionnement des mémoires caches et la description de leur influence sur la vitesse d'exécution des programmes permettra notamment de comprendre comment l'ordonnanceur peut essayer d'en tirer profit au maximum. Nous verrons ensuite comment caractériser le comportement mémoire d'une tâche et, dans une dernière partie, nous présenterons quelques approches visant à prendre en considération l'impact des caches.

2.1 Fonctionnement d'un processeur

Le processeur est le composant principal d'un ordinateur dans le sens où c'est lui qui est en charge d'interpréter et exécuter les instructions du programme. C'est pour cette raison qu'il est indispensable que ce dernier soit performant.

Dans cette partie, les principes fondamentaux du fonctionnement d'un processeur sont rappelés.

2.1.1 Exécution d'un programme

Un programme est une séquence d'instructions qui spécifie les opérations que le processeur doit effectuer afin d'obtenir un résultat. Ces instructions, appelées instructions machine, doivent être compréhensibles par le processeur qui exécute le programme. Les instructions machine qu'un processeur peut exécuter correspondent à son jeu d'instructions (*Instruction Set Architecture*, ISA). Les plus classiques sont les accès à la mémoire ou aux registres, les opérations arithmétiques et logiques, ou encore les sauts conditionnels. Le code machine d'un programme est lisible pour un développeur sous la forme de langage assembleur mais le processeur manipule en réalité les instructions sous une forme binaire.

De nos jours, il est de moins en moins fréquent de développer directement en langage assembleur. D'une part, les instructions de ce langage sont très bas niveau et n'offrent donc pas un niveau d'abstraction suffisant pour mettre en œuvre des fonctionnalités complexes. D'autre part, le code assembleur écrit est spécifique à une architecture précise et nécessite d'être revu en profondeur en cas d'évolution matérielle. Cependant, programmer en langage assembleur permet, dans certains cas, d'optimiser une portion de code afin d'utiliser au mieux les spécificités d'une architecture. Généralement, les programmes sont par conséquent écrits dans un langage de plus haut niveau, tel que le langage C par exemple, qui sera ensuite transformé en langage machine par un compilateur (voir figure 2.1).

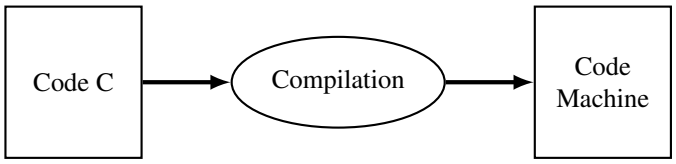


FIGURE 2.1 – Compilation du code source d'un programme vers du code machine.

Pour exécuter les instructions d'un programme, le processeur va réaliser en boucle ce qu'on appelle le cycle *fetch-decode-execute* et qui consiste dans les grandes lignes à :

- charger l'instruction depuis la mémoire où est stocké le code à exécuter et dont le compteur d'instructions donne l'adresse, puis incrémenter le compteur (*fetch*),
- décoder l'instruction (*decode*),
- exécuter l'instruction (*execute*).

Ces différentes étapes peuvent être découpées en sous-étapes plus précises selon la micro-architecture du processeur.

Le processeur doit faire un accès à la mémoire pour lire chaque instruction mais aussi lire et modifier des données de manière fréquente (voir figure 2.2).

2.1.2 Exemple de programme

La figure 2.3 montre le code assembleur d'un programme initialement développé en C et qui calcule les 15 premières valeurs de la suite de Fibonacci. Le code assembleur est destiné pour une architecture x86 (32 bits) et les principales instructions utilisées dans cet exemple sont résumées dans le tableau 2.1.

Dans cet exemple, la fonction *main* contient onze instructions. Or, le nombre total d'instructions exécutées est de 83. Ceci montre donc que certaines instructions (lignes assembleur 9 à 14) ont été appelées de nombreuses fois. Pour des raisons de performance, il

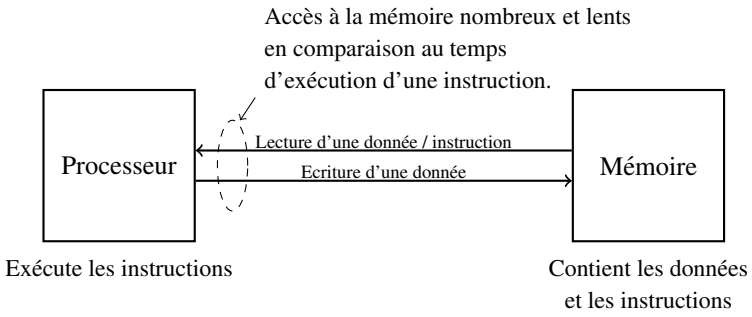


FIGURE 2.2 – Accès mémoire entre le processeur et la mémoire centrale.

Instruction	Description
mov	Copie une donnée ou une valeur vers un emplacement.
add	Additionne deux valeurs et place le résultat dans le premier argument.
cmp	Compare deux valeurs.
jne	Saute à l'adresse indiquée si la comparaison a retourné non égal.

TABLEAU 2.1 – Principales instructions utilisées dans l'exemple donné dans la figure 2.3.

```
1  int tab[15];
2
3  int main() {
4    tab[0] = 0;
5    tab[1] = 1;          } ①
6    int i = 2;
7
8    do {
9      tab[i] = tab[i-1] + tab[i-2]; ②
10     i++;
11   } while (i < 15);   ③
12
13   return 0;           ④
14 }
```

```
1          .text
2          .globl main
3          .type  main, @function
4  main:
5      ①   mov    DWORD PTR tab, 0
6          mov    DWORD PTR tab+4, 1
7          mov    eax, 2
8  .L3:
9      ②   mov    edx, DWORD PTR tab[-4+eax*4]
10         add    edx, DWORD PTR tab[-8+eax*4]
11         mov    DWORD PTR tab[0+eax*4], edx
12         add    eax, 1
13     ③   cmp    eax, 15
14         jne    .L3
15     ④   mov    al, 0
16         ret
17         .size  main, .-main
18         .comm  tab,60,32
```

FIGURE 2.3 – Conversion d'un programme C vers du code assembleur (syntaxe Intel).

est important d'éviter de devoir rechercher ces instructions depuis la mémoire centrale à chaque fois. En ce qui concerne les accès aux données (lignes 9 à 11), on peut voir que les données lues ont été accédées récemment. Là encore, il est important de pouvoir accéder rapidement à ces valeurs sans devoir aller les chercher en mémoire.

En conclusion, cet exemple montre la présence de nombreux accès à la mémoire pour lire les instructions et manipuler des données (voir figure 2.2). De plus, les accès sont susceptibles de fortement réutiliser certaines instructions et données.

2.1.3 Mémoire cache

L'accès à la mémoire centrale est lent en comparaison au temps d'exécution d'une instruction (voir figure 2.2). Il est donc primordial que ces accès soient les plus rapides possible.

Afin de réduire les temps d'accès aux instructions et données, il est possible d'utiliser des mémoires caches qui sont plus rapides d'accès et de fonctionnement.

Un cache est une zone mémoire conçue pour être très rapide mais en contrepartie ces caches ont pour défaut d'être plus petits pour des raisons technologiques et de coût. L'idée est alors de conserver dans cette mémoire les informations qui ont le plus de chance d'être à nouveau demandées lors de l'exécution du programme afin de réduire les temps d'accès. Ces caches sont intercalés entre le processeur et la mémoire comme le montre la figure 2.4.

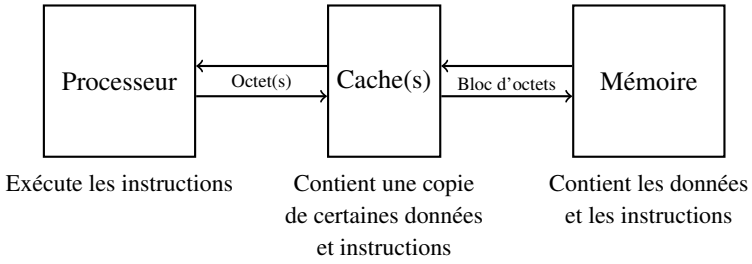


FIGURE 2.4 – Accès mémoire entre le processeur et la mémoire centrale avec un cache.

Lorsque le processeur doit accéder à une information, alors il commence par interroger le cache. Si l'information est présente, alors on parle de succès ou de *cache hit*. Au contraire, lorsque la donnée n'est pas présente, on parle alors de défaut de cache ou de *cache miss*. Dans ce second cas, l'information est recherchée dans un autre cache ou dans la mémoire centrale comme dans l'exemple donné par la figure 2.4.

2.1.4 Types d'architectures

Les principales architectures d'ordinateur sont les architectures *Von Neumann*, *Harvard* et sa variante dite *Harvard modifiée*. Elles diffèrent principalement dans leur manière de structurer les accès mémoire aux données et aux instructions.

L'architecture de *Von Neumann* a pour particularité de disposer d'une unique structure pour stocker les données et instructions. Instructions et données partagent ainsi les mêmes bus et le même espace d'adressage. L'inconvénient de cette architecture est que le partage d'un même bus pour les données et les instructions forme un goulot d'étranglement qui peut nuire aux performances.

L'architecture de type *Harvard*, quant à elle, sépare physiquement la mémoire des données et la mémoire d'instructions. L'accès se fait en utilisant des bus séparés ce qui permet d'accéder en même temps à une instruction et une donnée. Ceci apporte ainsi un gain de performance comparé à une architecture de type *Von Neumann*.

Cette architecture a ensuite été modifiée pour utiliser un même espace d'adressage pour les deux mémoires tout en conservant deux bus distincts. Très souvent, les données et instructions sont stockées au même endroit. Cette architecture permet cependant d'utiliser des caches séparés entre les instructions et les données comme le montre la figure 2.5. La présence de caches séparés peut éviter un goulot d'étranglement au niveau de la mémoire centrale ou d'un unique cache. Cette architecture est appelée *Harvard Modifiée*.

En pratique, la plupart des processeurs actuels sont de type *Harvard Modifié* à l'exception de quelques systèmes. C'est par exemple le cas du ARM7, de type *Von Neumann*, pour lequel les données et instructions sont stockées au même endroit et où la présence de

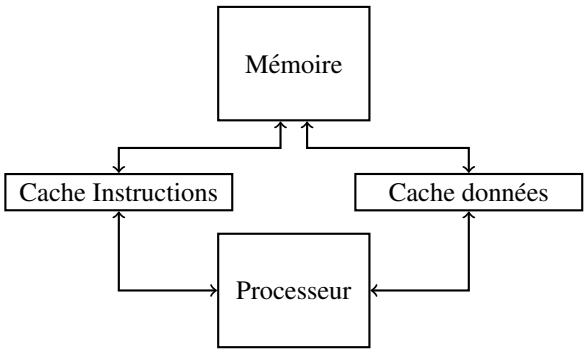


FIGURE 2.5 – Exemple d'architecture de type *Harvard Modifié* montrant les deux bus séparés mais une unique mémoire centrale contenant données et instructions.

caches n'est pas souhaitée pour des raisons de complexité de l'architecture. Au contraire, les microcontrôleurs Intel 8051, ainsi que certains PIC, utilisent un programme enregistré dans une mémoire disponible à l'exécution en lecture seule (exemple : EEPROM) et les données sont enregistrées en RAM ce qui justifie l'utilisation d'une architecture de type *Harvard*.

Notons également que d'un point de vue fonctionnel, une architecture de type *Harvard Modifié* est semblable à une architecture *Von Neumann*. Par conséquent, il est fréquent que des processeurs de type *Harvard Modifié* soient présentés comme étant de type *Von Neumann* (exemple : x86 ou ARM9). Parfois, le terme « Modifié » est également omis lorsqu'il s'agit d'insister sur les bus séparés.

2.1.5 Optimisations

Cette partie présente succinctement (et à titre d'information) les mécanismes les plus classiques qui permettent d'améliorer la vitesse d'exécution des programmes. Ces mécanismes reposent sur la capacité, dont disposent certaines instructions, à pouvoir s'exécuter en même temps sans interférence. C'est ce qu'on appelle le parallélisme au niveau des instructions (*Instruction-Level Parallelism*, ILP). De multiples stratégies permettent de mettre à profit cette propriété sur un processeur disposant pourtant d'un seul cœur.

a) Pipeline

L'exécution des instructions d'un programme peut être divisée en plusieurs étapes successives. En cela, l'exécution d'une instruction est similaire à l'assemblage d'un bien matériel. Prenons par exemple le cas de l'assemblage de voitures. Supposons que les étapes nécessaires soient d'installer le moteur (10mn), les portes (8mn) puis les roues (5mn), dans cet ordre. L'assemblage d'une voiture nécessite donc 23 minutes, soit un peu plus de deux voitures par heure. Or, si on dispose les voitures sur une ligne d'assemblage, il est possible d'installer un moteur sur la voiture suivante pendant la fixation des portes sur la première. Ainsi, dès que la première voiture est assemblée, une nouvelle voiture sort de la chaîne d'assemblage toutes les 10 minutes soit un débit plus de deux fois supérieur.

Un pipeline est une technique d'implémentation qui permet de traiter plusieurs instructions en même temps en réutilisant le principe de fonctionnement d'une ligne de production tel que présenté ci-dessus. Le nombre d'étapes dans un pipeline est un choix d'architecture du

processeur. Le gain en terme de vitesse d'exécution est théoriquement égal au nombre de niveaux de pipeline. Cependant, un certain nombre de cas empêchent l'utilisation optimale du pipeline :

- Le processeur ne permet pas l'exécution simultanée de certains calculs (*structural hazard*) ;
- Une instruction dépend du résultat d'une instruction précédente (*data hazard*) ;
- Le programme présente des branchements (*control hazard*).

Le temps passé par une instruction sur chaque niveau du pipeline correspond généralement à un cycle processeur. Ce dernier est réglé en fonction de la durée d'exécution du niveau de pipeline le plus lent. Il est donc important que la durée d'exécution de chaque niveau soit équilibrée.

b) Prédiction de branchement

Un programme est une suite séquentielle d'instructions. Cependant, il est possible d'effectuer des sauts au cours de l'exécution d'un programme afin de pouvoir effectuer des tests conditionnels et des boucles. La présence d'un branchement ne permet pas d'utiliser de manière efficace le pipeline ce qui peut provoquer un ralentissement du programme.

Une solution naïve pour gérer la présence de sauts dans un pipeline serait d'arrêter de traiter de nouvelles instructions à la détection d'un saut en attendant que la destination soit connue. Cependant, si le pipeline est long, ceci peut coûter de nombreux cycles d'horloge. Une autre solution est alors de faire un choix quant au branchement et si ce choix n'est pas le bon, alors les instructions présentes à tort dans le pipeline sont annulées. Cette approche entraîne un surcoût supplémentaire en cas d'erreur, mais en cas de succès les performances du pipeline ne sont pas dégradées.

Plusieurs stratégies permettent au processeur d'effectuer une bonne prédiction de branchement. Certaines optimisations peuvent se faire au moment de la compilation, tandis que d'autres peuvent se faire à l'exécution en tenant compte de l'historique des sauts précédents.

c) Superscalaire

Certains éléments, tels que des Unité Arithmétiques et Logiques (UAL) par exemple, sont disponibles en plusieurs exemplaires afin de permettre l'exécution simultanée de plusieurs instructions sur un même niveau de pipeline. Bien sûr, certains éléments ne sont pas redondants ce qui peut provoquer dans certains cas des conflits sur les ressources. Dans la plupart des cas, cela permet cependant d'améliorer les performances.

Certains processeurs superscalaires offrent également la possibilité de faire du Simultaneous Multithreading (SMT). Autrement dit, il est possible d'avoir plusieurs fils d'exécution pour un même processeur. Par exemple, Intel propose la technologie HyperThreading qui permet d'exécuter deux fils d'exécution par processeur. Les gains annoncés par Intel sont de l'ordre de 30% mais les résultats dépendent fortement des programmes.

d) Ordonnancement des instructions

Lors de la compilation d'un programme, le compilateur est capable d'agir sur l'ordre des instructions afin d'augmenter au maximum le niveau de parallélisme dans le but d'utiliser au mieux les processeurs superscalaires notamment. Cependant, il n'est pas raisonnable de faire un fichier binaire pour chaque micro-architecture en tenant compte de toutes ses spécificités. Ainsi, certains processeurs modernes sont capables de modifier l'ordre de certaines instructions au cours de l'exécution. On parle alors de processeurs *out-of-order*.

2.2 Principe d'une mémoire cache

Au cours des sections 2.1.1 et 2.1.2, l'importance de pouvoir accéder rapidement aux données et aux instructions a été mise en évidence. La solution proposée est l'utilisation de mémoires cache (voir section 2.1.3).

Cette partie présente les raisons pour lesquelles l'utilisation d'un cache permet en effet de réduire en moyenne la latence des accès. De plus, la raison pour laquelle on retrouve plusieurs niveaux de cache est également expliquée.

2.2.1 Technologies de mémoire

La DRAM (*Dynamic Random-Access Memory*) est un type de mémoire qui est simple, peu coûteuse et peu encombrante. Il est donc possible de fabriquer des composants mémoires de grande taille. Cependant, avec l'augmentation de la vitesse de calcul des processeurs, cette mémoire est devenue une source de ralentissement.

La SRAM (*Static Random-Access Memory*) est une autre technologie mémoire qui est plus rapide mais plus complexe. En effet, le nombre de transistors nécessaires pour une cellule de SRAM est six fois plus grand que pour une cellule de DRAM. Par conséquent, la SRAM est plus chère à fabriquer et plus encombrante.

La mémoire centrale devra contenir les instructions des programmes ainsi que les données que ces programmes manipulent. Ainsi, il est pertinent d'utiliser la DRAM pour sa capacité de stockage. Afin de pallier les problèmes de performance d'une mémoire utilisant de la DRAM, il est possible d'utiliser des caches utilisant de la SRAM, plus rapide mais plus petite.

2.2.2 Principe de localité

Un programme informatique n'accède généralement pas à sa mémoire de manière uniforme. Ainsi, certaines informations ont plus de chance d'être utilisées que d'autres et ce sont ces informations qu'il convient de garder en cache.

En effet, certaines instructions sont fréquemment exécutées dans des boucles, et les variables (données) sont souvent manipulées plusieurs fois dans un espace de temps proche. C'est ce qu'on appelle le principe de localité pour lequel on distingue deux types :

- Localité temporelle : lorsqu'une information a été utilisée récemment, elle a de fortes chances d'être de nouveau utilisée (exemples : variables du programme ou instructions répétées dans une boucle) ;
- Localité spatiale : lorsqu'une information a été utilisée récemment, les informations voisines ont plus de chances d'être utilisées (exemples : cases proches dans un tableau, instructions consécutives d'un programme).

La présence d'un cache permet de tirer profit de ces propriétés afin de réduire les temps d'accès aux données et instructions. La finalité première d'un cache est évidemment de garder en mémoire certaines informations récemment utilisées pour profiter de la localité temporelle. De plus, comme expliqué dans la partie qui suit, un cache manipule les informations sous la forme de blocs (ou lignes) de plusieurs octets consécutifs en mémoire. Autrement dit, lorsqu'une information est accédée, les octets proches sont eux aussi rapatriés dans le cache. Ceci permet de réduire les transferts entre la mémoire et le cache grâce à la localité spatiale.

2.2.3 Hiérarchie mémoire

Une ou plusieurs mémoires caches peuvent être présentes entre le processeur et la mémoire centrale, organisées sous la forme d'une hiérarchie. De nombreuses architectures comportent ainsi plusieurs niveaux de caches. Ces niveaux sont généralement appelés L1, L2, L3 et ainsi de suite, en fonction de leur position dans la hiérarchie. En effet, le processeur tente d'accéder à l'information en commençant par le cache L1, si l'information n'est pas présente, le cache L2 est sollicité, etc. Les processeurs destinés au grand public ont à l'heure actuelle en général trois niveaux de caches.

Il existe plusieurs raisons pour laquelle il est intéressant d'avoir plusieurs niveaux de cache. La première est qu'il est possible de réaliser des caches très rapides, parfois situés directement sur la puce du processeur, mais de petite taille. Un second cache de taille plus importante, mais par conséquent plus lent et encombrant, permet alors de seconder le premier niveau de cache et réduire le nombre d'accès à la mémoire. D'autre part, il est également possible d'avoir un cache, de grande taille, partagé entre plusieurs cœurs. La figure 2.6 illustre l'exemple d'une hiérarchie mémoire constituée de deux niveaux de cache L1 et L2. Dans le cache L1, les données et les instructions sont séparées ce qui permet d'éviter que les instructions évincent toutes les données du cache ou inversement. Au niveau du cache L2, données et instructions sont unifiées et le cache est partagé entre les deux cœurs.

Enfin, les caches pourront être inclusifs ou exclusifs. Dans le cas de caches inclusifs, toutes les informations contenues dans le premier niveau de cache seront présentes dans le second niveau de cache. Alors que dans le cas de caches exclusifs, une information ne peut être présente que dans un cache à la fois.

En pratique, on se réfèrera aux documentations du processeur ou du contrôleur de caches pour connaitre les temps d'accès aux différents composants en nombre de cycles. Par exemple, la documentation du L2C-310 qui est le contrôleur de cache L2 pour les ARM Cortex-A9 donne des temps de 1-2 cycles pour le L1, 8 cycles pour le L2 et 30-100 cycles pour la mémoire centrale ou un L3 [ARM10]. On retrouve le même ordre de grandeur sur un Intel Core i7 avec respectivement 4, 11 et 39 cycles pour les accès aux caches L1, L2 et L3 [Int08].

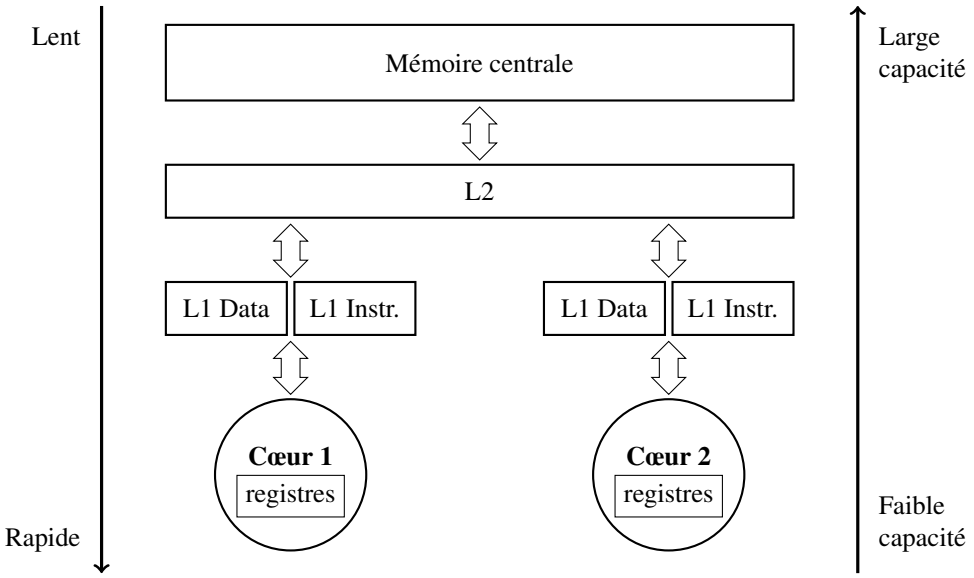


FIGURE 2.6 – Exemple de hiérarchie mémoire avec un cache L1 de données et un cache L1 d'instructions par cœur, et un cache L2 partagé.

2.3 Fonctionnement d'un cache

Une mémoire cache garde en mémoire les informations récemment utilisées car il est fort probable que ces informations soient à nouveau utiles dans un futur proche. Tous les caches ne sont pas identiques et il existe plusieurs paramètres importants pour un cache. L'objectif de cette partie est d'expliquer le fonctionnement interne des caches et de présenter les paramètres qui permettent de les caractériser.

2.3.1 Associativité

Avant toute chose, il est important de préciser qu'un cache ne manipule pas les informations octet par octet mais par groupe de plusieurs octets, appelé *ligne* ou *bloc*, et dont la taille varie d'une architecture à une autre. Ceci permet en particulier de limiter la quantité d'informations nécessaires à la gestion du cache. Un cache est ainsi constitué d'un ensemble de lignes.

Les informations contenues dans la mémoire centrale sont donc considérées sous la forme de lignes et ces lignes seront recopiées dans les emplacements du cache. Pour des raisons de clarté, nous supposerons par la suite que l'adressage se fait à l'octet, ce qui ne change pas les explications. L'adresse mémoire d'un octet est découpée en deux parties : le numéro de bloc qui permet d'identifier la ligne, et le numéro de l'octet qui permet d'identifier l'octet au sein d'une ligne. La figure 2.7 montre l'exemple du découpage d'une adresse 32 bits pour identifier le bloc de 64 octets et l'octet au sein de ce bloc.

L'emplacement mémoire à l'intérieur du cache où sera recopiée la ligne dépend de la stratégie du cache. Nous verrons ci-après le fonctionnement d'un cache direct où chaque ligne mémoire dispose d'un emplacement unique dans le cache, le fonctionnement d'un cache associatif qui décide de l'emplacement à utiliser à l'aide d'un algorithme de remplacement et enfin nous aborderons le cas des caches associatifs par ensembles qui sont un compromis entre ces deux approches.

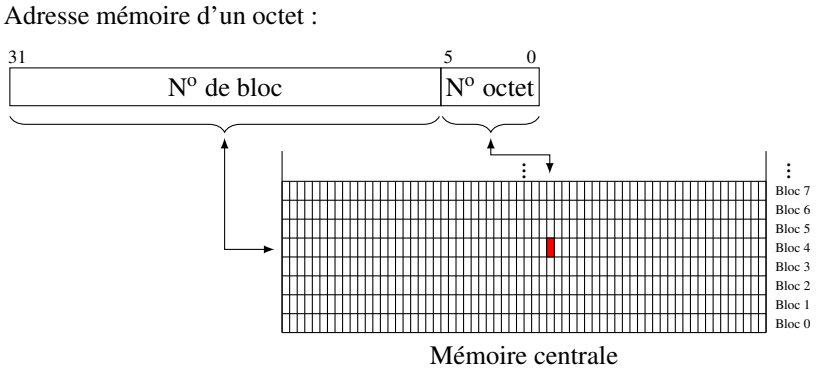


FIGURE 2.7 – Exemple de découpage d'une adresse mémoire pour identifier, dans la mémoire, un octet au sein d'un bloc de 64 octets (2^6).

a) Cache direct (*direct mapping*)

Dans le cas d'un cache utilisant une correspondance directe, chaque bloc de la mémoire est associé à un emplacement unique dans le cache. Cet emplacement est typiquement déterminé par l'indice formé par les bits de poids faible du numéro de bloc. Les bits restants du numéro de bloc (bits de poids fort) sont mémorisés dans le cache afin de conserver l'adresse du bloc actuellement présent dans le cache. Ces bits sont appelés le *tag* de la ligne.

La figure 2.8 illustre ce découpage. Afin de mieux comprendre le fonctionnement d'un cache direct, prenons l'exemple de la lecture d'un octet. Les 13 bits d'indice permettent d'identifier l'emplacement du cache où la ligne accédée devrait être, si elle est présente dans le cache. Ensuite, les bits de poids fort (*Tag*) sont comparés pour vérifier que la ligne présente dans le cache correspond à celle à laquelle on souhaite accéder. Si c'est le cas, l'octet est lu dans le cache en utilisant le numéro de l'octet. Sinon le bloc sera chargé depuis la mémoire et remplacera la ligne qui était présente au même emplacement.

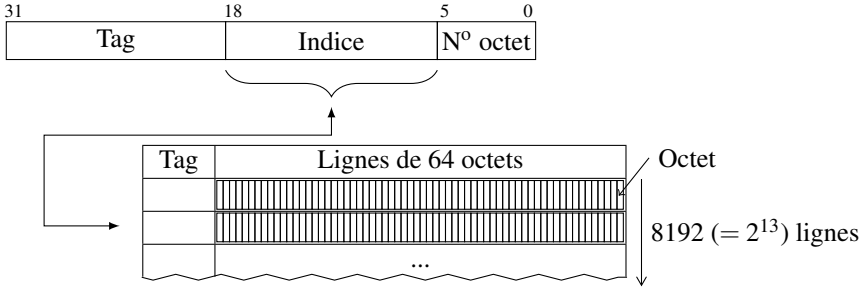


FIGURE 2.8 – Cache direct de 512 Kio (8192 lignes de 64 octets).

Le principal inconvénient de cette méthode est qu'une ligne sera retirée du cache si un accès est réalisé à une autre adresse qui possède le même emplacement dans le cache. Le risque est donc d'avoir un programme qui manipule en boucle deux valeurs qui vont s'exclure alternativement du cache.

L'intérêt de cette méthode est essentiellement la simplicité de sa mise en œuvre et sa vitesse d'exécution. En effet, on sait rapidement si une ligne est présente dans le cache et on sait également quelle ligne remplacer si besoin.

b) Cache associatif (*fully associative*)

De manière alternative, dans un cache associatif, une ligne peut potentiellement utiliser n'importe quel emplacement du cache. C'est un algorithme complémentaire qui va décider, pour chacune d'entre elles, au moment de leur recopie dans le cache, quel emplacement dans le cache elles doivent occuper.

L'inconvénient majeur de cette approche est la nécessité, à chaque accès au cache, de comparer le numéro du bloc demandé avec celui de l'ensemble des lignes présentes dans le cache afin de vérifier si la ligne est présente dans le cache. Au contraire, cette méthode permet de garder dans le cache les lignes qui ont le plus de chance d'être accédées à nouveau et donc de réduire le nombre de défauts de cache.

Lors d'un défaut de cache, l'algorithme de remplacement est appelé afin de déterminer l'emplacement du cache qui doit être utilisé pour accueillir la nouvelle ligne. Il existe tout un ensemble d'algorithmes pour ce faire tels que LRU (*Least Recently Used*), LFU (*Least Frequently Used*), FIFO, aléatoire, etc. La complexité de ces algorithmes influe sur les temps d'accès du cache. Les politiques les plus courantes sont LRU et ses approximations. Un cache LRU conserve les dernières lignes utilisées. Ainsi, lors d'un défaut de cache, la nouvelle ligne remplacera la ligne présente dans le cache dont la date du dernier accès est la plus ancienne. Un cache associatif provoque généralement moins de défauts de cache que la méthode par accès direct, mais elle est aussi plus complexe (algorithmiquement) et donc plus consommatrice en temps. Dès que le nombre de lignes devient trop grand, cette méthode est à écarter.

c) Cache associatif par ensemble à N voies (*N-way set associative*)

La correspondance associative par ensemble à N voies est une solution intermédiaire aux caches directs et associatifs qui se veut être un bon compromis entre ces deux approches. Cette solution consiste à adresser de manière directe un ensemble d'emplacements du cache puis d'utiliser le fonctionnement d'un cache associatif sur ce sous-ensemble du cache. Ainsi, on limite les inconvénients du cache associatif par un nombre réduit de lignes à comparer tout en gardant une partie de ses avantages. C'est donc un compromis entre un cache rapide (direct) et un cache qui fait moins de défauts de cache (associatif).

Un cache associatif par ensemble à N voies est un cache où chaque ensemble contient N emplacements. On dit aussi que son associativité est de N. Un cache direct est un cas particulier où l'associativité est de 1, et un cache entièrement associatif est un cache où l'associativité est égale à la capacité en nombre de lignes du cache.

La figure 2.9 montre le fonctionnement d'un cache associatif d'associativité 8 utilisant la politique de remplacement LRU. L'indice de la ligne sert à identifier l'ensemble du cache mais il est alors nécessaire de comparer le tag des 8 lignes de l'ensemble pour déterminer si la ligne est présente ou non. Si la ligne n'est pas présente, la ligne la plus anciennement utilisée sera remplacée.

Il est fréquent de voir des caches avec une correspondance directe ou avec une faible associativité (2 ou 4 lignes) pour les caches L1 dont on exige une grande vitesse. Alors qu'au contraire, pour les caches de niveau supérieur, l'associativité peut monter entre 2 et 16. Expérimentalement, lorsque l'associativité est supérieure à 8 ou 16, les résultats sont très proches de ce qui peut être obtenu avec un cache entièrement associatif [HS89]. Ainsi, les caches entièrement associatifs sont peu utilisés en pratique.

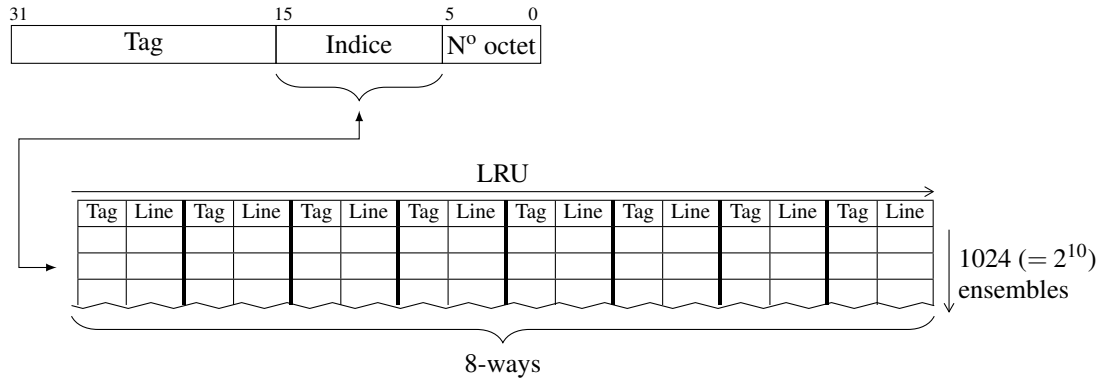


FIGURE 2.9 – Cache associatif 8-ways de 512 Kio (8192 lignes de 64 octets). 10 bits servent à identifier un ensemble parmi les 1024.

2.3.2 Protocoles de cohérence

Dans le cas d'architectures multiprocesseurs, il est nécessaire d'avoir un protocole de cohérence qui évite à un processeur de manipuler des données obsolètes. Un bloc peut posséder de nombreuses copies : potentiellement un par cache en plus de la mémoire centrale. Lorsqu'un bloc est modifié, toutes ses copies doivent être actualisées. De plus, l'ordre temporel des modifications doit être le même pour tous les processeurs.

Pour cela, il existe de nombreux protocoles dont les plus répandus semblent être MESI et MOESI. Ces deux protocoles mettent en œuvre une politique d'écriture de type *Write-back*, c'est-à-dire que les écritures sont différées (au moment où la ligne sera évincée du cache). Contrairement à une stratégie de type *Write-through* où les écritures se font immédiatement en mémoire et sur les copies en cache.

2.4 Sources de défauts de cache

L'usage de cache(s) permet d'augmenter la vitesse d'exécution des programmes, or les décisions d'ordonnancement peuvent avoir un impact sur l'utilisation des caches et donc indirectement sur la vitesse d'exécution des programmes. Nous nous intéressons donc maintenant aux sources de défauts de cache, c'est-à-dire les circonstances qui peuvent provoquer l'accès à des lignes non présentes dans le cache. Plus particulièrement, nous étudions ce qui se passe lors d'une préemption et lors du partage d'un cache entre plusieurs programmes qui s'exécutent simultanément afin de mieux comprendre le comportement des programmes vis-à-vis des caches. Dans le chapitre 6, un ensemble de modèles qui prennent en compte ces comportements sont présentés.

2.4.1 Types de défauts de cache

Le modèle des trois C [HS89, HP12] propose une répartition des défauts de cache en trois catégories :

- *Compulsory* : l'accès à une information appartenant à un bloc mémoire accédé pour la première fois provoque obligatoirement un défaut de cache, même si le cache a une taille infinie. On appelle aussi ces défauts de cache *cold misses*.

- *Capacity* : les lignes qui sont éliminées du cache mais qui devront être rechargées compteront comme défaut de type *Capacity*. Ce type de défaut de cache disparaît avec un cache de taille infinie.
- *Conflict* : dans le cas de caches directs ou associatifs par ensembles, ce sont les défauts de cache liés au fait que plusieurs blocs partagent un même emplacement dans le cache. Ce type de défaut de cache n'existe pas pour un cache entièrement associatif.

La réduction du nombre de défauts de cache de type *compulsory* peut se faire au niveau du programme, de la phase de compilation, ou en augmentant la taille des blocs. L'ordonnanceur ne pourra pas avoir un rôle dans leur réduction, mais ces défauts de cache sont cependant intéressants pour quantifier le nombre de lignes de cache utilisées par un programme.

Les défauts de cache de type *capacity* et *conflict* augmentent lorsque plusieurs programmes doivent se partager un cache, que ce soit en s'exécutant en alternance, ou sur deux cœurs différents avec un cache partagé. On désignera les défauts de cache qui surviennent suite à une préemption par l'expression *context switch misses* [LGS+08]. Enfin, lorsque deux programmes s'exécutent en parallèle et partagent un même cache, un phénomène de *cache thrashing* peut survenir. Ce phénomène désigne le fait que deux programmes vont mutuellement exclure des lignes de cache appartenant à l'autre programme et perdre ainsi une partie de l'avantage du l'usage d'un cache.

C'est l'ordonnanceur qui décide d'effectuer des préemptions et qui choisit les tâches qui s'exécutent simultanément. Ainsi, un ordonnanceur peut prendre en compte la présence des caches dans ses décisions afin d'améliorer les performances. Une politique d'ordonnancement qui tient compte des caches est appelée une politique *cache-aware*.

2.4.2 Préemption et « context switch misses »

Un changement de contexte nécessite du temps pour sauvegarder l'état du processus et relancer l'exécution d'un autre. Cependant, ce coût direct est généralement bien inférieur au surcoût lié aux défauts de cache causés par l'utilisation du cache par un autre programme [MB91]. Fromm et Treuhaft font le même constat mais pondèrent leurs résultats en notant que le nombre de défauts de cache engendrés par une préemption est cependant faible en comparaison au nombre total de défauts de cache qu'un programme subit lors de son exécution [FT96].

Des études montrent que l'impact des préemptions dépend fortement des programmes et de la taille des caches. Par exemple, si deux programmes s'exécutent en alternance mais que leur empreinte mémoire cumulée est inférieure à la taille du cache, il n'y aura pas de défaut de cache de type *capacity* mais éventuellement quelques défauts de cache de type *conflict*, en fonction de l'associativité. En revanche, si les deux programmes tiennent individuellement dans le cache, mais que leur empreinte totale dépasse la taille du cache, alors les effets seront importants. Enfin, pour des programmes faisant déjà beaucoup de défauts de cache, l'effet des préemptions sera insignifiant en comparaison [LDS07].

Lors d'une préemption, la durée d'exécution de la tâche préemptée peut augmenter suite à la perturbation de l'état des caches. Ce délai supplémentaire est appelé *Cache-Related Preemption Delay* (CRPD) et est illustré par la figure 2.10. La prise en compte des caches dans le calcul de la pire durée d'exécution est à la fois complexe et pessimiste. Il est en

effet délicat de déterminer à l'avance quelles seront les lignes de cache évincées suite à une préemption. Ceci pose évidemment des difficultés pour prouver l'ordonnançabilité des systèmes en prenant en compte les caches.

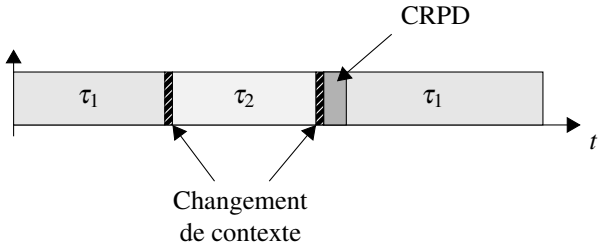


FIGURE 2.10 – Augmentation de la durée d'exécution du travail préempté suite à des défauts de cache supplémentaires.

Liu *et al.* donnent une explication détaillée de ce qu'il se passe au niveau d'un cache LRU suite à un changement de contexte [LS10]. La figure 2.11 montre à titre d'exemple l'état d'un ensemble d'un cache LRU d'associativité 8 avant et après une préemption. En noir, ce sont les lignes appartenant à un autre programme.

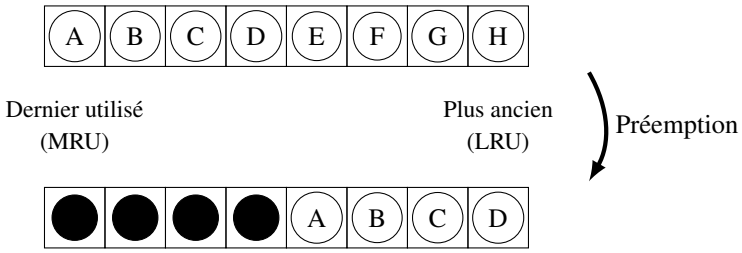


FIGURE 2.11 – État du cache à la reprise d'exécution, après une préemption.

Dans cet exemple, quatre lignes de cache ont été évincées. Cependant, ceci ne signifie pas que le nombre de défauts de cache engendrés par cette perturbation est de quatre (pour cet ensemble). Le nombre de défauts de cache supplémentaires provoqués par cette préemption peut en réalité être inférieur ou supérieur.

En effet, les lignes qui ont été évincées du cache peuvent soit ne plus être utilisées dans le futur, soit leur accès aurait de toute façon généré un défaut de cache (de type *capacity* notamment). Un programme qui réutilise très peu ses lignes et qui provoque de nombreux *cold misses* ne sera pas très sensible à une préemption. Liu *et al.* utilisent le terme de *replaced misses* pour désigner les défauts de cache concernant les lignes qui ne sont plus présentes au moment de la reprise de l'exécution et qui auraient été présentes sans la préemption [LGS+08].

De manière plus intéressante la perturbation causée par la préemption a provoqué l'apparition d'un « trou » dans le cache (lignes appartenant à un autre programme). Tant que ce « trou » reste présent dans le cache, il est susceptible de continuer à provoquer l'éviction de lignes supplémentaires. Les figures 2.12 et 2.13 permettent d'illustrer l'état du cache après l'accès à une ligne du cache.

Prenons d'abord le cas de l'accès à une ligne qui n'a pas été évincée (figure 2.12). La ligne de cache est alors simplement ramenée à la première position dans l'ordre LRU. Maintenant, voyons ce qu'il se passe lors de l'accès à une nouvelle ligne de cache. La figure 2.13 montre que l'ajout d'une nouvelle ligne se fait au détriment d'une ligne appartenant à la tâche et non au détriment d'une ligne appartenant à l'autre programme. Ainsi, non seulement

des lignes potentiellement utiles ont été retirées du cache, mais en plus, le programme qui reprend son exécution continue à exclure ses propres lignes de son cache. Fang Liu *et al.* utilisent le terme de *reorder misses* pour désigner les défauts de cache provoqués par la tâche elle-même à cause du trou créé par la perturbation [LGS+08].

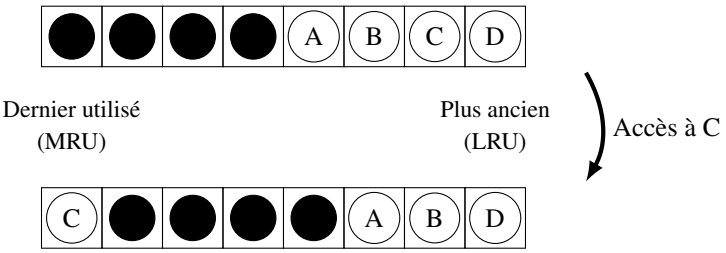


FIGURE 2.12 – État du cache suite à l'accès à une ligne non évincée.

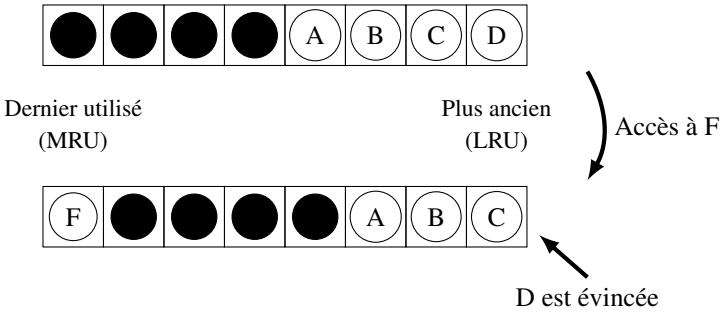
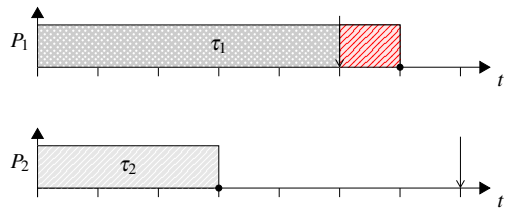


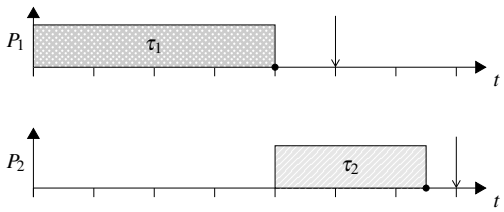
FIGURE 2.13 – État du cache suite à l'accès à une ligne qui n'est plus présente.

À moins d'être en mesure de simuler les caches en possédant la trace des accès mémoire, l'estimation du nombre de lignes évincées suite à une préemption est délicate. Cependant, le nombre de défauts de cache liés à une préemption est borné par le nombre de lignes dans le cache. Ceci justifie le choix fait dans plusieurs travaux de prendre en compte le coût des préemptions par des pénalités fixes [NS13].

2.4.3 Partage de cache et « cache thrashing »



(a) Exécution en parallèle de travaux provoquant une augmentation de la durée d'exécution et un dépassement d'échéance.



(b) Un simple décalage dans l'exécution des travaux permet d'éviter le dépassement d'échéance.

FIGURE 2.14

Dans une architecture multicœur, il est fréquent de trouver des caches partagés entre plusieurs cœurs [1]. Le partage du cache augmente le nombre de défauts de cache par rapport à une exécution isolée lorsque sa capacité n'est pas suffisante pour plusieurs tâches. La

1. Cette problématique se retrouve aussi sur les processeurs disposant de Simultaneous Multithreading (SMT) sur le premier niveau de cache.

figure 2.14 montre que l'augmentation de la durée d'exécution resultant de l'exécution simultanée des tâches partageant le même cache peut provoquer un dépassement d'échéance. Une décision d'ordonnancement différente aurait par ailleurs permis d'éviter ce problème. De manière générale, l'ordonnanceur peut décider d'éviter d'exécuter en même temps des tâches qui utilisent trop le cache lorsque la laxité est faible.

Des techniques proposent de partitionner les caches afin de contrôler le partage. De cette manière, la problématique du partage du cache est levée, mais cela pose également des problèmes pour une utilisation efficace du cache. Ces techniques sont présentées dans la partie 2.6.2 et nous ne nous intéressons ici qu'à ce qu'il se passe lorsqu'un cache est partagé sans mécanisme d'isolation.

Le partage du cache a peu de chance d'être équilibré : l'exécution de deux programmes en parallèle utilisant un cache commun de 256 Kio, n'est pas équivalent à l'exécution de ces deux tâches avec deux caches privés de 128 Kio si l'une des deux tâches a besoin de plus de 128 Kio de mémoire cache. Bien sûr, si le cache est suffisamment grand pour accueillir les deux programmes, le problème du partage du cache ne se pose presque pas [2].

En revanche, lorsque le cache est trop petit pour accueillir les deux programmes en même temps, les tâches vont exclure des lignes appartenant à l'autre et se pénaliser mutuellement. Là encore, cette pénalité n'est pas nécessairement équitable et dépend en réalité de nombreux facteurs difficiles à quantifier.

Afin de mieux comprendre les difficultés pour prédire le comportement d'un cache partagé, prenons l'exemple de deux tâches A et B qui s'exécutent simultanément sur deux cœurs différents et qui partagent un même cache. Si la tâche A fait beaucoup de *cold misses*, alors de nombreuses lignes seront ajoutées dans le cache ce qui peut provoquer l'éviction de lignes appartenant à la tâche B. Mais si la tâche B se limite à réutiliser des lignes très récemment insérées dans le cache, alors l'éviction de lignes plus anciennes n'aura aucun impact sur sa vitesse d'exécution. L'ancienneté des lignes accédées peut avoir un rôle important dans le nombre de défauts de cache d'une tâche dans le cas d'un cache partagé. Enfin, les tâches n'accèdent pas au cache à la même fréquence et une tâche qui accède beaucoup au cache aura davantage d'occasions d'évincer des lignes appartenant à une autre tâche.

2.4.4 Bilan

Cette partie a permis d'expliquer ce qu'il peut se passer dans un cache dans le contexte d'un système multitâche où il peut il y avoir des préemptions et des caches partagés. Nous retiendrons que le comportement des caches et leur impact sur la durée d'exécution des tâches est très complexe à évaluer.

En matière d'évaluation de performances dans le domaine du calcul parallèle, la problématique de l'estimation des effets des caches sur l'exécution des programmes s'est également posée et divers modèles ont été proposés. Ces modèles, statistiques, visent à donner une estimation du nombre de défauts de cache et par conséquent de la vitesse d'exécution des programmes, notamment dans le cas de caches partagés. Puisque notre objectif est de prendre en considération de manière réaliste l'architecture matérielle, nous pensons que ces modèles statistiques peuvent être une piste intéressante. Ces modèles ne permettront cependant pas de tenir compte de l'impact des caches dans le pire cas. Nous ne pourrons donc pas utiliser ces modèles pour des études d'ordonnançabilité.

2. Les défauts de type *conflict* peuvent cependant augmenter.

2.5 Caractérisation du comportement mémoire d'une tâche

Nous nous intéressons ici aux métriques disponibles pour caractériser le comportement d'une tâche vis-à-vis des caches. Dans la partie qui suit, nous verrons que certaines de ces métriques peuvent être utilisées par des ordonnanceurs pour utiliser au mieux les caches. Ces métriques servent également d'entrée pour des modèles statistiques que nous détaillerons dans le chapitre 6. Ces modèles permettent d'évaluer la durée d'exécution des tâches en fonction des caches disponibles sur un système. La plupart des informations présentées ci-dessous peuvent être recueillies lors de la phase de compilation [CDRPR00], par simulation [BLMT12, HE07] ou en exécutant le programme seul dans le système [SDR02, BH04].

Dans les métriques présentées ci-dessous, un accès au cache peut concerner l'accès à une instruction ou à une donnée. En conséquence, en fonction du type de cache considéré (données, instructions, ou les deux), ceci peut nous amener à ne pas prendre en compte les mêmes accès au cache dans le calcul de la métrique.

2.5.1 *Working Set Size*

Le modèle le plus simple pour définir une métrique d'observation du comportement du cache est le *Working Set Size* (WSS), ou *Working Set*, d'un programme. Ce modèle a été introduit par Denning pour étudier les accès aux pages mémoires [Den68]. Il définit le WSS d'un processus comme étant une fonction $W(t, T)$ qui a instant t donne le nombre de pages distinctes auxquelles il a accédé au cours des T dernière références. Cette valeur est un indicateur pour estimer la « localité » d'un processus, c'est-à-dire sa capacité pour travailler efficacement avec une portion réduite de mémoire à un instant donné. De nombreux travaux se sont attachés à proposer des modèles pour cette métrique ainsi qu'à l'utiliser pour optimiser l'attribution de la mémoire aux processus [SD72, BV13].

L'utilisation du WSS pour qualifier l'utilisation des mémoires caches a été proposée par Agarwal *et al.* [AHH89]. Pour cela, les auteurs modélisent les accès à un cache associatif en utilisant un WSS moyen. La définition introduite du WSS est le nombre moyen de blocs du cache différents auxquels un processus a accédé sur un intervalle de temps. Les travaux menés sur ce modèle indiquent clairement que le WSS est sensible à l'instant de la mesure à cause des phases dans l'exécution d'un programme (chargement des données, calcul, etc.). Cette métrique est donc un indicateur moyen du comportement d'un programme, mais ne permet pas de complètement le qualifier.

La compréhension de cette métrique est relativement intuitive : si la somme des *working set sizes* des tâches composant un système est inférieure à la taille du cache partagé, alors les tâches se perturberont peu [3]. A l'inverse, le WSS peut être un indicateur simple (mais imprécis) sur les perturbations que subiront les tâches si la taille du cache est insuffisante. Des travaux utilisent cet aspect afin de déterminer quelles tâches peuvent s'exécuter en parallèle afin de limiter leur interférence [ACD06].

2.5.2 Nombre de cycles par instruction

Le nombre de cycles par instruction (CPI) pour un programme correspond au nombre moyen de cycles processeur nécessaires à l'exécution d'une instruction. Ce CPI sert d'in-

3. Des perturbations pourront persister au niveau des ensembles de lignes dans le cas de caches associatifs par ensembles

dicateur de performance : plus le CPI est faible et plus le programme est efficace [MB91]. Le CPI d'une tâche s'exécutant seule dans un système est égal au rapport de sa durée d'exécution en cycles par le nombre d'instructions exécutées [HP12].

Le CPI est directement affecté par les défauts de cache car les délais supplémentaires pour accéder aux lignes non présentes dans le cache ralentissent l'exécution d'un programme.

Nous utiliserons plusieurs CPI par la suite :

- *cpi_alone* : le CPI de la tâche lorsqu'elle s'exécute seule ;
- *base_cpi* : le CPI de la tâche sans les pénalités liées aux accès mémoire ;
- *cpi* : le CPI de la tâche dans le système en tenant compte du partage éventuel des caches et/ou des *context switch misses*.

2.5.3 Fréquence d'accès au cache

Le nombre moyen d'accès au cache par instruction (abrégé en API, mais parfois appelé *mix* en fonction des auteurs) correspond au rapport entre le nombre d'accès au cache par rapport au nombre total d'instructions.

$$API = \frac{\text{Nombre accès cache}}{\text{Nombre d'instructions (IC)}} \tag{2.1}$$

En couplant l'API au CPI, on peut également en déduire la fréquence d'accès au cache d'une tâche x [CGKS05] :

$$Af_x = \frac{API_x}{cpi_x} \tag{2.2}$$

Cette métrique est donc un indicateur de l'usage du ou des caches par une tâche.

2.5.4 *Stack distance*

a) Définition

Un programme effectue un ensemble séquentiel d'accès à la mémoire aux travers des accès au cache. Une ligne est présente dans un cache LRU associatif si le nombre de lignes de cache distinctes accédées depuis le dernier accès à cette ligne est inférieur à la capacité du cache. Mattson *et al.* ont introduit la mesure de *Stack Distance* afin de déterminer le nombre de lignes différentes accédées entre deux accès consécutifs à une même ligne de cache [MGST70].

La figure 2.15 illustre cette notion de *stack distance*. Par exemple, la *stack distance* entre les deux accès à la ligne A est de 4 (accès aux lignes B, C, D et E).

b) Prise en compte de l'associativité

La mesure de *stack distance* peut prendre en compte les ensembles d'un cache, il suffit dans ce cas de ne considérer que les accès à des lignes du même ensemble dans le calcul de la distance (figure 2.15).

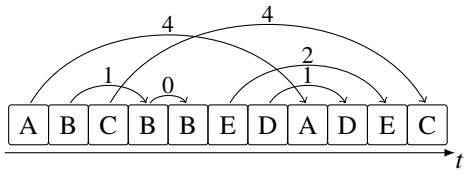


FIGURE 2.15 – Exemple de *stack distances*, les nombres correspondent aux distances.

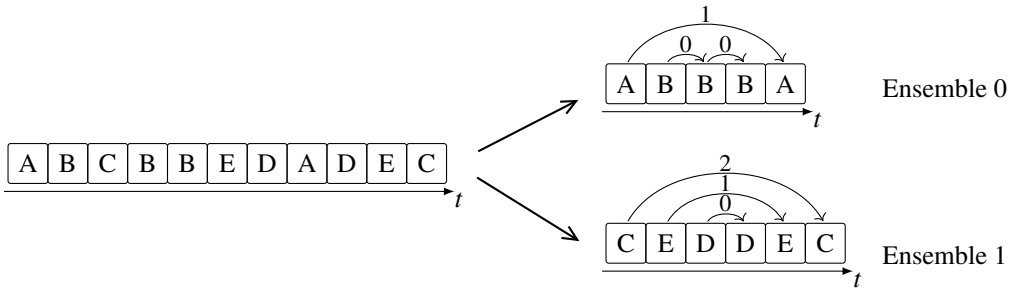


FIGURE 2.16 – Exemple de *stack distances* avec deux ensembles, les nombres correspondent aux distances.

c) *Stack Distance Profile*

La distribution de l'ensemble des *stack distances* forme le *Stack Distance Profile* (SDP). La figure 2.17 montre l'exemple d'une telle distribution sous la forme d'un histogramme. Le SDP contient de l'information sur la localité des accès mémoire d'un programme, c'est-à-dire sa propension à réutiliser une ligne de cache. La zone hachurée sur la figure représente les accès à une distance supérieure à l'associativité du cache et qui provoqueront un défaut de cache avec une politique LRU.

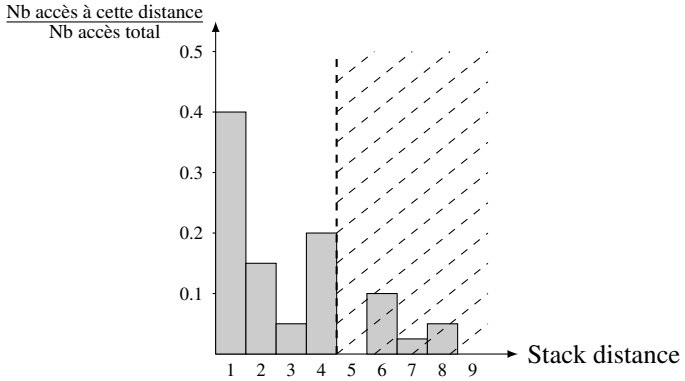


FIGURE 2.17 – *Stack Distance Profile* représenté sous la forme d'un histogramme.

Remarques : La mesure de SDP est complexe ce qui peut empêcher de le calculer pour des programmes trop longs [BK75, ACP02]. C'est pour cette raison que certains chercheurs se sont intéressés à la *reuse distance* qui est présentée ci-dessous.

Remarquons aussi que sur l'exemple de la figure 2.15, le nombre de distances est inférieur au nombre d'accès au cache. La différence correspond au nombre de lignes différentes accédées et qui provoqueront donc un *cold miss* lors du premier accès. Ainsi, la somme des valeurs du SDP ne fait pas 1.

2.5.5 *Reuse distance*

Suite à la remarque précédente, Berg *et al.* proposent de mesurer le nombre d'accès se faisant entre deux accès mémoire correspondant à une même ligne du cache. Ceci est plus rapide à calculer puisqu'il n'est alors plus nécessaire de maintenir une pile des adresses accédées et il suffit de mémoriser pour chaque ligne différente la date (exprimée en nombre d'accès depuis le début) du dernier accès.

Cette mesure est appelée *reuse distance* [BH04]. La figure 2.18 montre un exemple d'accès mémoire aux lignes de cache A, B, C, D et E ainsi que les *reuse distances*.

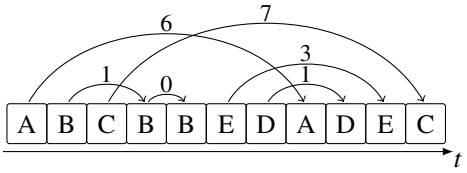


FIGURE 2.18 – Exemple de *reuse distances*, les nombres correspondent aux distances.

Il existe une technique statistique qui permet de calculer la distribution de *stack distances* à partir d'une distribution de *reuse distances* et qui donne selon les auteurs de très bons résultats [EH10].

2.5.6 Récapitulatif

Le tableau 2.2 résume les métriques servant à caractériser le comportement mémoire d'une tâche qui viennent d'être présentées.

Working Set Size (WSS)	Nombre de lignes de caches nécessaires pour assurer une bonne vitesse d'exécution à une tâche.
API	Nombre d'accès à la mémoire par instruction.
CPI	Nombre de cycles nécessaires en moyenne pour exécuter une instruction.
base_cpi	CPI sans les pénalités d'accès au cache.
cpi_alone	CPI pour une exécution seule.
cpi	CPI en tenant compte de l'impact des autres tâches.
Stack Distance	Nombre de lignes de cache différentes accédées depuis le dernier accès à la même ligne.
Stack Distance Profile (SDP)	Distribution normalisée de Stack Distances.
Reuse Distance	Nombre de lignes de cache accédées depuis le dernier accès à la même ligne.

TABLEAU 2.2 – Tableau récapitulatif des métriques.

2.6 Prise en compte des caches dans les systèmes ordonnancés

Au cours des parties précédentes, nous avons vu que les caches ont un impact sur la durée d'exécution des programmes et donc sur l'ordonnançabilité des systèmes. Le pro-

blème d'évaluation du WCET est fortement lié à l'ordonnancement, ainsi Ding *et al.* ont par exemple proposé une approche d'ordonnancement hors-ligne afin de pouvoir calculer le WCET plus précisément tout en prouvant que le système est ordonnançable [DZ13]. D'autres travaux se sont quant à eux focalisés sur des mécanismes pour contrôler, voire optimiser, ces effets afin de rendre le système plus prédictible, plus efficace dans l'utilisation des ressources ou pour améliorer les critère d'ordonnançabilité.

Plusieurs de ces méthodes se focalisent sur l'estimation du WCET en prenant en considération les caches partagés par plusieurs tâches (voir partie 2.4). Dans cette partie, nous verrons ainsi des techniques, nécessitant parfois des modifications du système (logicielles ou matérielles), qui permettent entre autre de limiter les interférences entre les tâches et ainsi de pouvoir calculer plus précisément le WCET. Cette estimation étant alors d'une plus grande précision, il est possible d'obtenir des conditions d'ordonnançabilité moins pessimistes.

Cette partie présente aussi une liste d'approches pour prendre en compte les caches dans l'ordonnancement. La plupart des algorithmes présentés ne permettent pas d'améliorer les conditions d'ordonnançabilité mais ils améliorent les performances des caches. Cette approche est particulièrement intéressante dans le cas de systèmes temps réel souples dans la mesure où les durées d'exécution des tâches sont réduites ce qui améliore les temps de réponse et réduit les risques de dépassement d'échéance.

2.6.1 Sauvegarde du cache

Lors d'une préemption, le système d'exploitation sauvegarde l'état de la tâche afin de pouvoir la recharger au moment de la reprise de son exécution. Il est possible d'étendre ce principe en sauvegardant le contenu des caches. Ceci augmente certes le coût de la préemption mais ce coût peut être intégré aux coûts de préemption et imputé à la tâche préemptive comme un coût fixe comme le montre la figure 2.19.

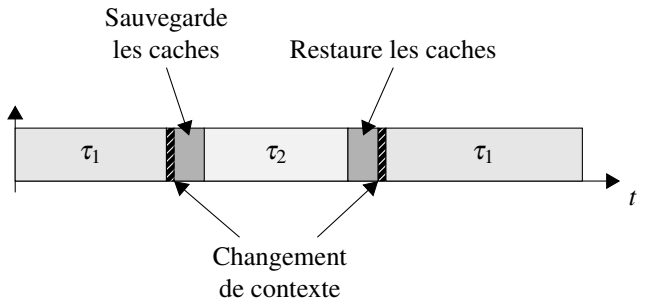


FIGURE 2.19 – Mécanisme de sauvegarde des caches pour ne pas impacter sur la durée d'exécution de la tâche préemptée mais uniquement sur celle de la tâche qui préempte.

Malheureusement, si la taille des caches est importante, ce coût peut être trop important. Whithman et Audsley proposent alors un mécanisme similaire où la tâche qui préempte ne sauvegarde que les lignes de cache qu'elle va utiliser [WA12].

2.6.2 Partitionnement du cache

Le partitionnement de cache consiste à créer des partitions afin d'éviter que plusieurs tâches ne puissent interférer entre elles par l'utilisation des mêmes emplacements. Ces partitions peuvent être faites soit de façon logicielle [PLM09], soit de façon matérielle [Kir89].

Pour un cache associatif par ensembles, une solution est de regrouper des ensembles pour former des partitions. Nous nous intéressons ici aux systèmes temps réel, mais notons que des travaux similaires sont également menés dans l'ordonnancement non temps réel [LLD+08, KCS04] pour des raisons de partage équitable des caches entre les tâches. Cette méthode offre la possibilité d'évaluer de manière plus précise les WCET en bornant les pires cas.

Le partitionnement peut être fixé au début de l'exécution (partitionnement statique) ou peut varier en fonction de la stratégie adoptée (partitionnement dynamique). De plus, le partitionnement de cache peut se faire au niveau des cœurs pour les caches partagés ou des tâches.

Le partitionnement statique au niveau des tâches permet d'éviter l'interaction entre tâches qui s'exécutent en parallèle ou de manière consécutive puisque chaque tâche dispose de son propre espace en cache. En revanche, s'il y a beaucoup de tâches, la taille des partitions sera fortement réduite. Par conséquent, elles risquent de ne pas avoir assez de mémoire cache pour s'exécuter rapidement. Altmeyer *et al.* étudient l'ordonnançabilité de systèmes de tâches sur une architecture monoprocesseur avec un ordonnancement à priorité fixe en comparant les résultats produits par des techniques de partitionnement des caches avec celles qui prennent en considération les coûts CRPD [ADLD14]. Ces évaluations sont faites sur différents *benchmarks* standards (voir annexe A.5) et en utilisant les outils d'analyse mis au point par les auteurs. Les résultats indiquent que la supériorité de l'une ou l'autre des approches dépend fortement des tâches. En particulier, lorsque la part du CRPD comparé au WCET est plus faible, l'approche non-partitionnée donne de meilleurs résultats que le partitionnement fixe optimal.

Le partitionnement statique au niveau des cœurs permet seulement de supprimer les interférences liées aux caches partagés entre tâches qui s'exécutent en parallèle. Puisque le nombre de cœurs est généralement bien inférieur au nombre de tâches, le partitionnement du cache au niveau des cœurs implique des parties de cache plus grandes. À titre d'exemple, Berna et Puaut proposent ainsi une heuristique qui décide du partitionnement des tâches sur différents cœurs et du partitionnement du cache entre les différents cœurs [BP12]. Le WCET est alors calculé en fonction de la taille du cache disponible sur chaque cœur.

Enfin, le partitionnement dynamique consiste à laisser le soin à l'ordonnanceur de décider pendant l'exécution du système des partitions de caches qui seront réservées pour les tâches. Ainsi, Guan *et al.* partent d'un système de tâches tel que pour chaque tâche est précisé l'espace en cache nécessaire et le WCET calculé en fonction de cet espace [GSYY09]. Leur politique, une variante de NP-FP, n'exécute une tâche que s'il y a assez d'espace cache disponible.

2.6.3 Verrouillage du cache

Le partitionnement du cache simplifie l'évaluation du WCET en limitant les interactions entre les tâches. Cependant, la prise en compte des caches dans l'évaluation du WCET reste néanmoins difficile. La technique de verrouillage du cache est une autre technique, pouvant être complémentaire au partitionnement [SM08], et qui permet de bloquer des lignes dans le cache. Ceci nécessite bien évidemment des caches particuliers.

Le verrouillage du cache peut être statique ou dynamique [CPRBM03]. Dans le cas d'un verrouillage statique, les données sont chargées dans le cache au moment du lancement du système puis elles sont verrouillées. Alors que dans le cas d'un verrouillage dynamique, il est possible de changer les lignes présentes en cache pendant l'exécution du système.

2.6.4 Co-ordonnancement

Les ordonnanceurs décident des tâches qui s'exécutent et donc leurs décisions peuvent naturellement prendre en compte le comportement mémoire des tâches pour utiliser au mieux les caches, notamment dans le cas de caches partagés. Ces ordonnanceurs sont appelés *cache-aware* pour cette raison.

Plusieurs travaux se basent sur le WSS des tâches (voir 2.5). Anderson *et al.* ont introduit le concept de *megatask* dans le but de réduire la mise en parallèle de tâches utilisant trop le cache partagé [ACD06]. C'est une façon de décourager les tâches ayant un fort usage du cache de s'exécuter en même temps. Calandrino *et al.* proposent un ensemble d'heuristiques visant à prendre en compte l'impact des caches pour améliorer les performances du système, tout en garantissant le respect des contraintes temporelles [CA08]. Enfin, Lindsay *et al.* utilisent le WSS des tâches pour guider le partitionnement et ainsi réduire les durées d'exécution [LR14].

Jain *et al.* utilisent quant à eux la notion de symbiose afin de caractériser le bon niveau de cohabitation des tâches entre elles dans le cadre d'architecture SMT [JHA02]. La symbiose se calcule alors en mettant en parallèle les tâches et en lisant la valeur des compteurs de performance offerts par l'architecture matérielle. En dehors du domaine temps réel, Fedorova *et al.* ont proposé un algorithme [FSS06] permettant d'accorder plus de temps aux tâches ayant été pénalisées dans leur usage du cache partagé et réciproquement. L'objectif est alors de partager de manière équitable l'usage d'un cache partagé et cela permet en définitif d'améliorer les performances globales du système.

Tous ces algorithmes ne permettent pas d'améliorer l'ordonnançabilité de systèmes temps réel durs, mais permettent néanmoins de réduire les durées d'exécution des tâches en pratique ce qui peut être utile pour des systèmes temps réel souples ou simplement pour réduire la consommation énergétique ou améliorer les temps de réponse.

2.7 Conclusion

Le mécanisme de cache permet d'augmenter fortement la vitesse d'exécution des programmes en réduisant les temps d'accès aux données et aux instructions. Or, réduire les durées d'exécution permet généralement d'augmenter le nombre de systèmes ordonnançables mais aussi d'améliorer les temps de réponse, ou encore réduire la consommation. Toutefois, le comportement de ces caches est difficile à prévoir ce qui pose des problèmes dans le cas de systèmes temps réel. Mais compte tenu de l'importance des caches, il serait dommage de simplement les désactiver. Plusieurs pistes sont donc activement étudiées pour permettre leur utilisation.

Dans le cas de systèmes temps réel durs, il est primordial de pouvoir statuer avec certitude sur l'ordonnançabilité d'un système. Ainsi, la première approche consiste à intégrer dans les analyses l'effet des caches sur les durées d'exécution. Cette approche est considérée comme étant particulièrement complexe en plus d'être souvent pessimiste. C'est pour cette raison que des techniques de partitionnement et verrouillage de cache ont été proposées afin de supprimer en partie ou intégralement les effets liés au partage des caches entre les tâches. Malheureusement, partitionner un cache peut dans certains cas ralentir la vitesse d'exécution des tâches comparativement à la mise à disposition de l'ensemble du cache.

Dans le cadre de systèmes temps réel souples, une utilisation effective des caches permet de réduire le nombre de dépassements d'échéance. Ainsi, au lieu de chercher à caractériser les effets des caches, l'objectif est ici d'en optimiser l'usage. Plusieurs politiques, dites *cache-aware* ont été proposés en ce sens. Ces dernières prennent en considération les caractéristiques des tâches dans leurs décisions d'ordonnancement.

L'ensemble des éléments présentés montrent que les caches ont un effet sur l'ordonnançabilité des systèmes et que inversement les choix d'ordonnancement impactent leur efficacité. Les caches ne devraient donc pas être ignorés dans les évaluations des algorithmes d'ordonnancement et cela tout particulièrement pour les architectures multiprocesseurs.

CHAPITRE 3

Motivations

Au cours du premier chapitre, une liste d'environ cinquante politiques d'ordonnancement pour architectures multiprocesseurs a été établie. Malheureusement, il est extrêmement difficile de les comparer car les différentes démonstrations théoriques et expérimentations disponibles ne réutilisent pas systématiquement les mêmes hypothèses, les mêmes entrées, les mêmes méthodes, les mêmes métriques, ni les mêmes implémentations.

Il existe plusieurs critères pour évaluer les algorithmes d'ordonnancement. Dans le cas de systèmes temps réel durs, la principale préoccupation est évidemment leur capacité à ordonnancer correctement le plus grand nombre de systèmes. Toutefois, d'autres éléments de performances ne doivent pas être pour autant ignorés comme les durées d'exécution, les temps de réponse, etc. De plus, dans tous les cas, l'évaluation de ces performances doit être la plus proche possible de la réalité. Les aspects opérationnels qui impactent temporellement l'exécution des systèmes à ordonnancer doivent ainsi être pris en compte. Parmi ceux-ci, nous avons choisi de nous intéresser aux surcoûts temporels liés aux appels de l'ordonnanceur et à la prise en compte des caches sur la durée d'exécution des tâches.

Ce chapitre expose ce qui a motivé ce travail de thèse. Pour ce faire, nous établissons pour commencer un constat des difficultés liées à l'évaluation d'algorithmes d'ordonnancement et à la prise en compte des aspects opérationnels. Ensuite, nous détaillons les objectifs que nous nous sommes fixés. Afin de les satisfaire, nous avons identifié de manière précise les besoins correspondants afin de déterminer la potentielle adéquation d'un outil existant. Parmi les outils à notre connaissance, quatre d'entre eux ont pu être évalués. Nous concluons sur une explication du choix de l'approche retenue.

3.1 Constats

Dans cette partie, nous faisons un certain nombre de constats concernant l'évaluation des politiques d'ordonnancement. Ce sont ces constats qui ont motivé ce travail de thèse.

3.1.1 De nombreuses politiques d'ordonnancement...

Le premier chapitre de cette thèse a présenté les principales approches pour l'ordonnancement de tâches temps réel sur des architectures multiprocesseurs. Nous en rappelons ici

brièvement l'historique. Dans un premier temps, les politiques d'ordonnancement monoprocesseurs ont été naturellement généralisées aux architectures multiprocesseurs à travers deux grandes approches : l'ordonnancement global et l'ordonnancement partitionné. Une simple généralisation de ces politiques ne permettant pas une utilisation optimale des processeurs disponibles, des recherches visant à améliorer l'ordonnançabilité des systèmes par des modifications mineures de ces politiques s'ensuivirent (exemple : EDZL ou EDF-US). Dans un même temps, des chercheurs se sont attelés à trouver de nouvelles approches afin de repousser la limite d'ordonnançabilité des systèmes par ces politiques. En 1996, l'introduction par Baruah *et al.* de la notion d'équité (*fairness*) a inspiré la création de politiques qui ont permis d'atteindre l'optimalité. Cependant, malgré de bonnes propriétés théoriques, les politiques PFair engendrent un trop grand nombre de préemptions, migrations et prises de décisions de l'ordonnanceur pour pouvoir être utilisées en pratique. Quelques années plus tard, des chercheurs ont montré comment réduire de manière significative le nombre de prises de décisions en relâchant la contrainte d'équité sans pour autant perdre en ordonnançabilité. Ces politiques, appelées BFair ou DP-Fair, font cependant toujours l'objet de critiques concernant leur nombre trop élevé de préemptions et migrations. Au milieu des années 2000, les politiques semi-partitionnées sont nées de cette volonté de réduire le nombre de préemptions et migrations. Pour cela, ces dernières cherchent à tirer profit des avantages offerts par les politiques globales et de ceux des politiques partitionnées, quitte parfois à faire un compromis entre limite d'utilisation et nombre de préemptions. Des politiques plus simples dérivées d'EDF ont cependant gardé un attrait important comme l'attestent les travaux récents ayant engendré les politiques G-FL et U-EDF.

Près d'une cinquantaine d'algorithmes d'ordonnancement ont ainsi été recensés au cours du premier chapitre (voir tableau 1.3). Pourtant cette liste est loin d'être exhaustive et de nombreuses variantes n'ont pas été présentées. Certaines permettent d'étendre un algorithme à de nouveaux modèles de tâches, d'autres proposent des améliorations liées à l'implémentation, et d'autres proposent des modifications mineures pour améliorer tel ou tel comportement de l'algorithme.

Bien que certaines politiques soient devenues obsolètes avec l'apparition de politiques plus récentes, aucune politique n'a su s'imposer véritablement devant les autres. Ce domaine de recherche est par conséquent très actif et de nouvelles politiques d'ordonnancement sont attendues. Il est donc nécessaire de disposer d'outils pour permettre leur étude, leur évaluation et leur comparaison.

3.1.2 Quelles approches pour l'évaluation de ces politiques ?

Trois grandes approches sont généralement utilisées pour l'évaluation de politiques d'ordonnancement :

- L'analyse théorique des algorithmes basée sur des modèles de systèmes.
- L'évaluation sur des systèmes réels. Dans ce cas, on a le choix entre utiliser des tâches réelles ou de simples tâches consommatrices de temps.
- L'utilisation d'outils de simulation. Ces derniers imposent une implémentation des algorithmes et la simulation utilise des modèles pour les comportements des éléments matériels et logiciels impliqués.

a) Analyse théorique

L'analyse théorique se base sur une modélisation des systèmes et, compte tenu de l'importance du respect des contraintes temporelles, elle permet en premier lieu d'établir des conditions analytiques d'ordonnançabilité. Malheureusement, la complexité des preuves font qu'un certain nombre de simplifications dans la modélisation du système sont nécessaires. Citons par exemple le coût nul des préemptions dans la majorité des études, ainsi que le coût nul des appels à l'ordonnanceur. De plus, les conditions d'ordonnançabilité, lorsqu'elles ne sont que suffisantes, affichent souvent un caractère très pessimiste ce qui conduit à surdimensionner les systèmes.

L'étude des autres propriétés du système tel que le nombre de préemptions ou les temps de réponse sont souvent très difficiles à exprimer sans tomber dans des bornes excessives. Les études théoriques vont plus facilement donner des indications concernant le pire cas que le comportement réel. Par conséquent, ces analyses sont fréquemment complétées par une évaluation empirique sur un système réel ou par simulation.

b) Évaluation sur système physique

Afin de montrer le bon comportement d'un ordonnanceur en pratique, il est possible de le mettre en œuvre sur une vraie architecture, d'y mesurer les indicateurs accessibles pertinents, puis de collecter et analyser les données alors obtenues. Les tâches exécutées sont soit issues de benchmarks, soit elles sont artificielles et dans ce cas elles s'exécutent pendant des durées générées aléatoirement. Plusieurs outils ont été conçus avec pour objectif de faciliter la mise en œuvre d'un nouvel ordonnanceur (voir annexe A.2). LITMUS^{RT} est probablement l'outil le plus connu car il a déjà été utilisé pour évaluer de nombreuses politiques d'ordonnancement. L'utilisation d'une architecture réelle apporte de nombreux avantages, notamment la prise en compte des spécificités de l'architecture matérielle et des surcoûts système.

L'exécution sur un système réel impose cependant le choix d'une plate-forme dont il sera difficile de modifier les caractéristiques. De plus, ces outils nécessitent un investissement en temps très important pour mettre en œuvre les ordonnanceurs, les évaluer et collecter les données que l'on souhaite mesurer. Ceci peut donc constituer un frein pour leur utilisation. Notons l'existence de travaux visant à permettre la spécification d'un ordonnanceur de manière générique et simplifiée, et qui génèrent automatiquement le code de l'ordonnanceur adapté au système d'exploitation [MLD05].

c) Évaluation par simulation

Afin de permettre la modification des caractéristiques du système, il est possible d'utiliser un simulateur d'architecture. Un simulateur d'architecture est un logiciel qui simule l'architecture matérielle d'un système. Les instructions qui doivent s'exécuter normalement sur un processeur physique sont en réalité interprétées par le simulateur. En plus de pouvoir émuler un processeur du point de vue fonctionnel (comme le fait par exemple l'émulateur de machine QEMU), ces logiciels sont aussi capables de simuler les aspects temporels. Une liste de tels outils est présentée dans l'annexe A.3. Ainsi, il devrait être possible en théorie d'exécuter les outils dédiés à l'évaluation sur système physique (exemple : LITMUS^{RT}). Néanmoins, en fonction du niveau de précision dans la simulation temporelle, la simulation de l'architecture est plus lente qu'une exécution sur système physique.

De plus, bien que cela puisse permettre de modifier les caractéristiques matérielles de la plate-forme, les autres difficultés, notamment sur la prise en main, restent de mise. Nous n'avons pas connaissance de travaux d'évaluation d'algorithmes d'ordonnancement utilisant un tel outil.

Les chercheurs utilisent à la place ce que nous appellerons la simulation d'ordonnancement. La simulation d'ordonnancement est une technique à mi-chemin entre une mise en œuvre sur un système réel et l'approche analytique qui en fait totale abstraction en se basant sur des modèles. Cette approche nécessite également la mise en œuvre de l'algorithme d'ordonnancement à étudier mais la simulation réalisée par l'outil pourra se faire à divers niveaux de « précision ». Ainsi, il est possible d'agir sur la durée d'exécution des travaux sur les processeurs, les pénalités liées au système, etc. Les éléments qui constituent le système (tâches, processeurs ou autre) sont modélisés et apportent au simulateur les informations utiles à leur simulation. Il n'y a généralement pas de code réel rattaché aux tâches. Bien que ce type de simulation se limite à des modèles de tâches relativement simples en comparaison à toute la finesse d'une exécution sur cible, les modèles utilisés peuvent cependant être potentiellement affinés afin de s'approcher d'un comportement réel.

L'utilisation de tels simulateurs d'ordonnancement est très fréquente pour étudier le comportement d'un algorithme et pour montrer son bon fonctionnement. En comparaison à l'utilisation de systèmes réels, les outils de simulation sont généralement plus simples à utiliser ce qui a contribué au succès de cette approche. Et ce, à la fois dans la phase de mise en œuvre d'une politique et dans la phase d'expérimentation. Il est également plus simple de faire varier les paramètres du système simulé et d'explorer ainsi des systèmes variés. De nombreux travaux utilisent cette approche pour calculer des métriques telles que le nombre de préemptions, migrations, dépassements d'échéance ou encore temps de réponse.

3.1.3 Des difficultés pour combiner les résultats existants...

Compte tenu du nombre important d'algorithmes d'ordonnancement, les évaluations qui sont faites dans les publications se limitent souvent à la comparaison de l'algorithme présenté avec quelques politiques notables. Ces évaluations sont généralement faites par les auteurs des algorithmes avec comme but d'en afficher la supériorité. De telles évaluations sont disponibles en grand nombre mais les différences dans les conditions expérimentales rendent impossible l'unification des différents résultats existants.

Premièrement, toutes les évaluations n'utilisent pas les mêmes protocoles expérimentaux (voir partie 3.1.2) et les outils utilisés ne sont pas toujours disponibles. De plus, les expérimentations diffèrent sur des paramètres ayant une forte incidence sur les résultats tels que les ensembles de tâches (choix des périodes, des durées d'exécution et du nombre de tâches) ou le nombre de processeurs.

Les codes sources des algorithmes d'ordonnancement ne sont pas systématiquement partagés avec le reste de la communauté. Par conséquent, les chercheurs doivent souvent implémenter de leur côté les politiques qu'ils souhaitent étudier et on ne peut pas être sûr que leur implémentation n'a pas une influence sur certains résultats. En effet, pour certaines politiques, des choix arbitraires peuvent être pris et ceux-ci ne sont pas toujours sans conséquence sur certaines valeurs mesurées.

Enfin, il existe une multitude de données pouvant être mesurées telles que le taux de systèmes ordonnançables, le nombre de dépassements d'échéance, le nombre de préemptions

ou les temps de réponse. Le choix des données mesurées varie d'une étude à l'autre et parfois la définition même de certains évènements n'est pas strictement identique.

3.1.4 Et des aspects opérationnels oubliés ou au second plan...

Le WCET est fréquemment employé pour modéliser de manière fixe et pire cas la durée d'exécution d'un travail. Or, dans un système qui s'exécute réellement, il existe une certaine variabilité dans la durée d'exécution des travaux. Ainsi, l'utilisation du WCET soulève des questions lorsqu'elle est utilisée pour l'évaluation empirique des performances des politiques en matière de préemptions, migrations ou autres métriques directement dépendantes de la durée d'exécution des tâches. À titre d'exemple, l'utilisation du WCET ne permet pas d'apprécier la capacité de certaines politiques à tirer profit de durées d'exécution plus faibles que prévu. La définition 1.10 (cf. chapitre 1) rappelle également que certains algorithmes ne sont pas prédictibles. Cela signifie que certains ensembles de tâches ordonnançables ne le sont plus lorsque la durée d'exécution des travaux est réduite.

Les résultats obtenus par des évaluations théoriques se basent en général sur des modèles simples, assez éloignés des considérations pratiques, et qui ignorent les surcoûts temporels liés aux préemptions et migrations. Néanmois, de nombreuses études comptabilisent leur nombre car ces événements peuvent être la source d'un ralentissement de l'exécution des tâches (surcoûts liés au changement de contexte et perturbation des caches). Mais ceci n'est pas satisfaisant pour deux raisons. La première est que l'impact réel d'une préemption ou d'une migration n'est pas constant et dépend de la tâche préemptée, de celle qui préempte et du moment. La seconde est que le ralentissement de l'exécution d'une tâche suite à une préemption pourrait dans une certaine mesure augmenter la charge du système et être la source de nouvelles préemptions par effet boule de neige. Par conséquent, un simple nombre de préemptions n'est pas forcément significatif. Les architectures multiprocesseurs apportent aussi davantage de complexité avec des caches partagés, de nouveaux bus de communication ou encore des interruptions inter-processeurs qui sont susceptibles d'introduire des latences supplémentaires. Ainsi, l'impact temporel de ces mécanismes matériels sur la durée d'exécution des travaux doit être pris en considération.

Parmi les éléments pouvant avoir une influence sur la durée d'exécution des travaux, il y a les mémoires cache. Nous avons vu par ailleurs dans la partie 2.6 que certaines politiques, dites *cache-aware* essaient de prendre en compte les caractéristiques des tâches pour améliorer les performances générales du système en utilisant au mieux les caches. En effet, les interférences sur les caches suite à une préemption peuvent causer des délais additionnels. De manière similaire, lorsqu'un cache est partagé entre plusieurs cœurs, l'exécution d'une tâche peut avoir un impact significatif sur la vitesse d'exécution d'une tâche s'exécutant sur un autre cœur (voir chapitre 2). L'évaluation de telles politiques n'a de sens qu'avec une prise en compte des caches.

Ces architectures soulèvent aussi de nouvelles questions au niveau du système d'exploitation : quel cœur devrait exécuter le code de l'ordonnanceur ?, quelles structures de données devraient être verrouillées ?, etc. Les recherches doivent maintenant s'attaquer aux problèmes liés à la mise en œuvre effective des algorithmes [CLB+06].

3.2 Objectifs

L'objectif premier de cette thèse est l'étude et l'évaluation de politiques d'ordonnancement multiprocesseur. Sa finalité est de mieux comprendre le fonctionne-

ment et les spécificités des nombreux algorithmes qui ont été proposés. Ceci devrait nous permettre à terme de conseiller dans le choix de tel ou tel ordonnanceur, d'améliorer certains algorithmes existants ou de proposer de nouvelles politiques.

Pour atteindre cet objectif, nous optons pour la simulation d'ordonnancement en raison des avantages présentés dans la partie 3.1.2. Cette approche doit être réalisée sur des expérimentations systématiques à grande échelle (c'est-à-dire sur des jeux de tests de taille significative et sur un large ensemble de politiques d'ordonnancement) dans des conditions expérimentales maîtrisées et clairement exposées. Il nous semble aussi nécessaire d'enrichir les modèles pour mieux prendre en compte les aspects opérationnels afin de pallier les problèmes relevés dans la partie 3.1.4.

Partant de ces constatations, notre objectif peut être décliné sous la forme de trois axes. Le premier est de mettre en œuvre les principales politiques d'ordonnancement existantes. Le second objectif est de permettre une automatisation des évaluations pour comparer dans les mêmes conditions les algorithmes. Enfin, le dernier axe est d'améliorer le niveau de détail de la simulation afin de prendre en compte certains aspects opérationnels.

a) Mise en œuvre des algorithmes d'ordonnancement

Afin de répondre à l'objectif de cette thèse, il est évidemment nécessaire d'offrir un moyen pour mettre en œuvre des algorithmes d'ordonnancement dans le cadre des évaluations. Cela passe par une connaissance et une appropriation pratique des politiques existantes, mais aussi par la mise en place d'outils efficaces pour pouvoir les implémenter et rendre reproductibles les expérimentations par la communauté scientifique.

À notre connaissance, le travail le plus avancé de classement et de recensement des algorithmes d'ordonnancement multiprocesseur a été publié en 2011 par Davis et Burns [DB11]. Or, de nombreux travaux ont été proposés depuis ce qui nécessite une actualisation. Au cours du chapitre 1, nous avons ainsi proposé un survol des principales approches et des algorithmes pour l'ordonnancement temps réel, en y intégrant notamment de nouveaux algorithmes apparus récemment.

Ce travail de bibliographie montre une très grande variété de méthodes pour mettre en œuvre les algorithmes d'ordonnancement. Il est donc nécessaire de disposer d'un outil d'évaluation avec des moyens flexibles et adaptables pour implémenter les algorithmes et ce quelle que soit leur approche. Les principaux algorithmes d'ordonnancement devront être proposés avec cet outil. De plus, l'outil doit offrir un support pour évaluer dans des conditions identiques les politiques existantes, mais aussi celles qui seront proposées dans le futur. Nous verrons dans le chapitre 4 comment ce point a été traité.

Enfin, nous avons remarqué la difficulté pour obtenir le code source de certains ordonnanceurs et pour reproduire certaines expérimentations (voir partie 3.1.3). Nous avons donc la volonté de rendre disponible nos implémentations ainsi que d'offrir la possibilité à quiconque de réutiliser notre plateforme pour compléter les évaluations dans les mêmes conditions (ordonnanceurs évalués sur les mêmes entrées et les mêmes métriques).

b) Mise en place des évaluations

Notre objectif est d'évaluer statistiquement de nombreuses politiques d'ordonnancement sur un ensemble varié de configurations. Il faut donc offrir des conditions expérimentales reproductibles et maîtrisées, ainsi que des moyens pour conduire efficacement ces études.

Pour cela, l'automatisation des expérimentations est un point essentiel. La génération des systèmes, leur simulation et l'extraction des données doit pouvoir se faire automatiquement et dans des conditions expérimentales explicites. En particulier, les différentes méthodes pour la génération des systèmes et notamment des tâches doivent être caractérisées. En effet, les principaux paramètres (nombre de tâches, nombre de processeurs, charge du système, etc.) ont une influence sur les comportements des algorithmes et nous souhaitons pouvoir étudier cette influence pour chaque paramètre indépendamment. Ce point est abordé dans le chapitre 4 sur l'outil d'évaluation, mais aussi explicitement exposé dans la partie 5 présentant les résultats des expérimentations.

En dehors des paramètres d'entrée des évaluations, nous portons aussi une attention particulière à la définition des métriques d'observation. Nous ne souhaitons pas nous limiter à l'ordonnançabilité qui est souvent la principale caractéristique étudiée dans les publications mais nous voudrions nous intéresser également aux performances des politiques en termes de surcoût à l'exécution, temps de réponse ou autre. En effet, de nombreuses politiques affichent des qualités théoriques similaires en matière d'ordonnançabilité, mais la prise en compte de critères supplémentaires pourrait justifier l'utilisation de l'une ou de l'autre.

c) Prise en compte des aspects opérationnels

Comme annoncé, il s'agit là de prendre en compte plus finement les aspects opérationnels, c'est-à-dire les éléments de l'architecture matérielle et logicielle qui ont une influence sur l'ordonnancement. En effet, l'architecture matérielle a un impact sur la vitesse d'exécution des tâches et le système d'exploitation introduit des surcoûts temporels notamment lors des changements de contexte ou lors des appels aux méthodes de l'ordonnanceur.

Parmi les éléments matériels qui influent sur l'ordonnancement, nous avons décidé de nous concentrer sur les caches. Comme exposé dans le chapitre 2, l'ordonnancement et l'efficacité de cet élément de l'architecture sont dépendants l'un de l'autre : les décisions d'ordonnancement agissent sur leur efficacité (partage du cache dû au parallélisme, perte de la cohérence suite à une préemption, etc.), et inversement, les mémoires caches ont un impact sur la vitesse d'exécution des tâches ce qui peut avoir des conséquences sur l'ordonnancement.

Afin de modéliser le comportement des caches au niveau qui nous intéresse, c'est-à-dire au niveau de l'évaluation des politiques d'ordonnancement, nous avons retenu les travaux de la communauté sur l'évaluation de performance des architectures. En effet, une grande partie d'entre eux s'intéressent aux mémoires caches et plusieurs modèles visent à estimer le nombre de défauts de cache et par conséquent la vitesse d'exécution des programmes. Ces modèles sont statistiques et se basent majoritairement sur le *Stack Distance Profile* [MGST70] (voir partie 2.5). Nous tâcherons de savoir si ces modèles satisfont ou non nos besoins pour évaluer les politiques d'ordonnancement temps réel à travers un simulateur d'ordonnancement.

Pour pouvoir répondre à cette question, nous nous sommes fixé comme critères les éléments suivants : les temps de calculs des modèles doivent être faibles pour permettre la conduite de nombreuses simulations ; la taille des caches et leur hiérarchie doit être un paramètre d'entrée afin d'étudier leur influence ; les caractéristiques de l'utilisation du cache par les tâches doivent pouvoir être générées automatiquement ; les résultats de la simulation doivent être proches de ce que nous pourrions obtenir sur cible physique.

Concernant les surcoûts liés au système, des pénalités temporelles pourront être appliquées au moment de la simulation. Pour cela, l'interface de programmation permettant la mise en œuvre d'un ordonnanceur doit être réaliste dans le sens où l'ordonnancement doit reposer sur des mécanismes que l'on retrouve de manière classique sur les systèmes. L'ordonnanceur pourra ainsi intervenir lors des évènements classiques de type activation ou fin d'exécution de travaux, mais il pourra également utiliser des timers pour éventuellement prendre des décisions d'ordonnancement conduites par le temps. De même, dans un système réel, le code de l'ordonnanceur s'exécute sur l'un des cœurs ce qui a un impact sur l'exécution du système. Ainsi, l'interface de programmation doit aussi pouvoir permettre de spécifier quel cœur exécute le code de l'ordonnanceur afin de pouvoir imputer au cœur le temps nécessaire à l'exécution du code. Enfin, dans le cas d'architectures multiprocesseurs, il peut être nécessaire d'interdire l'exécution simultanée de certains codes du système. Un système simple d'exclusion mutuelle doit ainsi pouvoir être défini. Tous ces éléments sont introduits dans le chapitre 4.

3.3 Besoins

Nous énonçons ci-après les exigences détaillées que nous estimons indispensables à la réalisation de nos objectifs. Elles constitueront une base pour l'évaluation d'outils candidats existants.

Besoin 1. Définir un algorithme d'ordonnancement

Besoin 1.1. *Pouvoir spécifier des politiques globales, partitionnées ou hybrides*

Description : L'outil doit supporter l'étude de politiques pour architectures multiprocesseurs issues d'approches variées. Ceci inclut donc des politiques globales, partitionnées, ainsi que des solutions hybrides (semi-partitionnée, clustering ou encore hiérarchique).

Besoin 1.2. *Proposer une interface de programmation simple*

Description : La mise en œuvre d'un algorithme d'ordonnancement doit être simple et rapide car il y a énormément de politiques à étudier. De plus, cette simplicité doit engager d'autres chercheurs à utiliser le même outil pour leurs expérimentations.

Besoin 1.3. *Proposer une interface de programmation réaliste*

Description : La mise en œuvre d'algorithmes d'ordonnancement sur un vrai système pose des questions d'implémentation dont les réponses peuvent influer sur le comportement obtenu. Il y a par exemple des problèmes liés à la précision (les timers utilisés sur une architecture réelle n'ont pas une précision infinie) ou encore des choix à faire comme par exemple le choix du processeur qui doit exécuter une méthode de l'ordonnanceur. De par leur impact, ces aspects ne doivent pas être négligés même en simulation.

Besoin 2. Prendre en compte l'aspect opérationnel

Besoin 2.1. *Supporter différents types de tâches*

Description : L'outil doit pouvoir gérer des tâches indépendantes, préemptives et périodiques. L'intégration et la simulation d'autres types de tâches en particulier des tâches sporadiques doit être possible sans difficultés majeures.

Besoin 2.2. *Pouvoir influer sur les durées d'exécution des travaux*

Description : On veut pouvoir spécifier et simuler des travaux à durée d'exécution variable et ne pas simplement se limiter au WCET.

Besoin 2.3. *Prendre en compte l'architecture matérielle*

Description : Les caractéristiques des processeurs doivent pouvoir être définies et prises en compte dans la simulation. Parmi les caractéristiques qui nous intéressent, nous pouvons citer par exemple la vitesse du processeur ou encore la hiérarchie mémoire (caches) et l'organisation de ces mémoires vis-à-vis des processeurs.

Besoin 2.4. *Prendre en compte les surcoûts liés à l'ordonnanceur*

Description : Certaines politiques ont une complexité algorithmique importante et il convient d'en tenir compte dans leur évaluation. Ceci englobe les surcoûts liés à l'appel des méthodes de l'ordonnanceur ainsi que les surcoûts système liés aux changements de contexte (sauvegarde et restauration des registres).

Besoin 3. Évaluer un ensemble de politiques d'ordonnancement

Besoin 3.1. *Générer des tâches*

Description : L'utilisateur doit pouvoir générer des ensembles de tâches pour les expérimentations en utilisant le générateur de son choix. La mise à disposition de générateurs classiques est un avantage, mais il est indispensable de pouvoir également utiliser un générateur extérieur.

Besoin 3.2. *Définir les paramètres de l'étude*

Description : Les différents paramètres, tels que l'intervalle d'étude et la précision de la simulation, doivent pouvoir être définis.

Besoin 3.3. *Automatiser des expérimentations*

Description : Afin de pouvoir conduire des expérimentations à grande échelle et donc à valeur statistique. Il est indispensable de mettre à disposition de l'expérimentateur des services permettant de conduire de manière automatique un grand nombre de simulations.

Besoin 3.4. *Recueillir des mesures de simulations*

Description : On veut pouvoir recueillir les indicateurs de performance les plus usuels tels que le nombre de préemptions, migrations ou les temps de réponse, mais il doit aussi être possible d'obtenir n'importe quel autre type de métrique (plus ou moins facilement selon sa complexité). Une représentation graphique de l'exécution du système est également souhaitée pour mieux comprendre le fonctionnement d'un algorithme.

Besoin 4. Être utilisable

Besoin 4.1. *Être librement accessible*

Description : L'outil doit être disponible gratuitement sans restrictions trop lourdes sur son utilisation (licence notamment).

Besoin 4.2. *Être facile à étendre*

Description : Le simulateur doit pouvoir être étendu pour y intégrer des modèles variés de tâches et d'architectures matérielles.

Besoin 4.3. *Exécutable sur des plates-formes répandues*

Description : Les prérequis informatiques pour l'exécution de l'outil doivent être facilement atteints.

Besoin 4.4. *Être facile à utiliser*

Description : L'installation et la prise en main de l'outil ne doit pas dissuader de potentiels utilisateurs.

Besoin 4.5. *Être documenté*

Description : Une documentation doit être fournie avec l'outil.

Besoin 4.6. *Être suffisamment rapide*

Description : Dans l'objectif de pouvoir lancer de nombreuses expérimentations, il est important que l'outil soit suffisamment rapide.

Besoin 4.7. *Pouvoir déboguer facilement*

Description : Certains algorithmes d'ordonnancement sont complexes et leur conception sans erreur est difficile. L'outil doit offrir des moyens adaptés pour identifier facilement l'origine des problèmes (représentation visuelle, lecture du contenu de variables, etc).

3.4 Évaluation des outils disponibles

De nombreux simulateurs ont été développés par des chercheurs pour évaluer des politiques d'ordonnancement. Malheureusement, nombre d'entre eux ont été pensés pour un objectif donné et ils sont difficilement adaptables à un autre objectif. Ceci a naturellement conduit à l'abandon de ces outils et beaucoup n'ont même pas été mis à disposition de la communauté. Dans cette partie, les outils MAST [HGGM01], Cheddar [SLNM04], STORM [UDT10] et Yartiss [CFM+12] sont présentés. D'autres outils existent ou ont existé tels que STRESS, PERTS, GHOST, RTSIM, FORTISSIMO, FORTAS ou encore RealtssMP pour ne citer qu'eux. Cependant, au moment de notre examen des outils disponibles, nous n'avons pas été en mesure, soit d'obtenir leur code source, soit de les compiler et les faire fonctionner. Une description de ces outils est disponible dans l'annexe A.1.

3.4.1 MAST

MAST [HGGM01] (Modeling and Analysis Suite for Real-Time Applications) est un ensemble d'outils permettant de modéliser des applications temps réel et d'effectuer une analyse temporelle de ces dernières. Il est développé depuis le début des années 2000 à l'Université de Cantabria et est disponible sous licence libre GNU GPL v2 [1].

1. Disponible à l'adresse : http ://mast.unican.es

MAST définit avant tout un modèle permettant de décrire un système temps réel distribué ainsi que les contraintes temps réel associées. C'est un modèle conduit par les événements ce qui lui permet d'établir des dépendances entre les différentes tâches. Plusieurs outils ont été développés autour de ce modèle. L'utilisation principale de MAST est l'analyse d'ordonnançabilité basée sur le pire temps de réponse, le calcul de temps de blocage ou encore l'analyse de sensibilité. Le modèle MAST et son logiciel éponyme sont encore activement développés. Une seconde version du modèle a vu le jour et ajoute de nouvelles fonctionnalités qui pourraient être intégrées à terme dans le modèle MARTE [HGD+13].

MAST dispose également d'un simulateur de comportement temporel : jSimMAST. Ce simulateur permet de fournir de nouvelles mesures complémentaires à l'analyse théorique. Contrairement à MAST qui est développé en Ada, jSimMAST est fait en Java. Cet outil n'a pas été pensé pour permettre de tester de nouvelles politiques mais il est cependant possible de modifier le code source et ajouter soi-même la nouvelle politique. Malheureusement, l'absence de documentation expliquant précisément comment développer une nouvelle politique ne rend pas les choses aisées. De plus, seuls FP et EDF sont actuellement disponibles et l'interface à respecter n'est pas simple à prendre en main.

3.4.2 Cheddar

Cheddar [SLNM04] est un outil qui permet de vérifier le respect des contraintes temporelles d'un système temps réel [2]. Il est développé sous licence libre GNU GPL par le laboratoire LISyC de l'Université de Bretagne Occidentale et par Ellidiss Technologies.

Cheddar propose à la fois un environnement de simulation, mais aussi des tests de faisabilité. Les algorithmes les plus usuels sont disponibles dans l'outil (RM, EDF, DM, LLF, FIFO). Cheddar est capable de prendre en entrée un modèle AADL ou un fichier dans le format spécifique de Cheddar.

Notons que de récents travaux visent à intégrer dans Cheddar la prise en compte des CRPD pour les caches d'instructions [TSRB14].

Tout comme MAST, Cheddar n'a pas été conçu pour évaluer des politiques d'ordonnancement mais plutôt les systèmes. Par conséquent, rien n'est fait pour faciliter la mise en œuvre d'une nouvelle politique. Il est nécessaire d'implémenter l'algorithme à l'intérieur de Cheddar, en Ada. Les fichiers correspondant aux mises en œuvre des politiques sont peu commentés et il n'y a pas, à notre connaissance, de documentation pouvant guider pour l'implémentation d'une nouvelle politique.

3.4.3 STORM

STORM [UDT10] est un simulateur d'ordonnancement développé à l'IRCCyN à Nantes et disponible sous licence libre GNU LGPL v2 [3]. Ce simulateur permet, à partir des caractéristiques du système (tâches et processeurs) et du choix d'une politique d'ordonnancement, de simuler l'exécution des tâches. Le résultat est un diagramme de Gantt ou un fichier de trace de l'exécution des tâches sur les processeurs.

Contrairement à MAST et Cheddar, STORM a été développé avec pour objectif l'étude des politiques d'ordonnancement. La mise en œuvre d'un algorithme se fait en Java en

2. Disponible à l'adresse : http ://beru.univ-brest.fr/ singhoff/cheddar/
3. Disponible à l'adresse : http ://storm.rts-software.org

respectant l'API du simulateur. De nombreuses politiques d'ordonnancement ont été développées pour STORM [4], montrant au passage qu'il est possible d'exprimer de manière unifiée des politiques globales, partitionnées ou hybrides. Le développement d'une nouvelle politique est relativement rapide et il est possible d'utiliser un débogueur Java pour analyser les bugs de l'implémentation d'une politique d'ordonnancement. Il existe une documentation et des exemples qui facilitent la prise en main de l'outil.

La génération des tâches n'est pas incluse dans l'outil mais celui-ci prend en entrée un fichier XML contenant toutes les caractéristiques du système. La génération de systèmes est donc facilement faisable à partir d'outils externes. Il est possible de définir plusieurs types de tâche et d'influer sur la durée d'exécution de celles-ci. STORM ayant été développé avec l'objectif d'analyser des politiques d'ordonnancement, il est également possible d'automatiser des expérimentations.

STORM satisfait la plupart de nos besoins, mais malheureusement la simulation se fait à pas fixe ce qui rend difficile l'intégration d'événements ayant une durée inférieure à celle d'un pas. De plus, le temps d'une simulation étant inversement proportionnel à la durée d'un pas, il est impossible de donner une valeur très petite à cette dernière. Les durées d'exécution, périodes et échéances doivent aussi être des multiples du pas utilisé. Modifier STORM afin de pouvoir exprimer des durées très petites (pour les surcoûts liés aux appels à l'ordonnanceur par exemple) nécessiterait une réécriture du noyau du simulateur. Une part importante du logiciel serait à modifier.

3.4.4 Yartiss

Yartiss [CFM+12] est un autre simulateur d'ordonnancement qui se rapproche de STORM dans ses objectifs. Il est développé par le laboratoire LIGM de l'Université Paris-Est Marne-la-Vallée sous licence libre GNU GPL [5].

Yartiss permet d'évaluer et comparer des politiques d'ordonnancement sur des systèmes multiprocesseurs. Il se distingue principalement par l'intégration de modèles énergétiques. L'équipe qui développe Yartiss a déjà de l'expérience dans la conception d'outils de simulation d'ordonnancement. Entre 2005 et 2008, ils ont conçu RTSS, mais d'après les auteurs, certaines parties ont été développées dans la hâte et le code n'était pas documenté. Ils ont donc décidé de développer un nouvel outil RTMSim entre 2008 et 2011 pour gérer les architectures multiprocesseurs. Finalement, en 2012, ils ont décidé d'écrire Yartiss en repartant de zéro, mais en tenant compte des erreurs passées. Un effort particulier a été fait pour offrir des moyens simples d'étendre le simulateur avec de nouveaux modèles.

Yartiss a fait le choix d'intégrer la génération des tâches (avec l'algorithme UUniFast-Discard) ainsi que l'analyse des résultats directement au sein du logiciel, en plus de pouvoir utiliser des outils externes. L'outil est également en mesure de lancer de multiples simulations et de générer des courbes concernant plusieurs métriques (préemptions, échecs, périodes d'inactivité, etc). L'intégration dans l'outil de la génération des systèmes jusqu'à la génération des courbes évite en principe de devoir recourir à des outils externes. Cependant, ceci a malheureusement participé à la complexification de l'interface utilisateur. Le code source du logiciel est bien documenté mais nous n'avons pas trouvé de documentation utilisateur, ni de documentation de l'architecture qui nous permettrait de comprendre comment étendre l'outil pour satisfaire nos besoins.

4. De manière non exhaustive : FP, RM, EDF, FIFO, EDZL, LLF, PD², LRE-TL, P-EDF, etc.
5. Disponible à l'adresse : http ://yartiss.univ-mlv.fr/

Yartiss utilise un moteur de simulation événementiel. Bien qu'il se focalise principalement sur la prise en compte de modèles énergétiques, il serait sans doute possible de l'étendre pour ajouter la prise en compte des surcoûts ou encore d'intégrer des modèles de tâches prenant en compte l'effet des caches dans la durée d'exécution des tâches. Cependant, Yartiss a été publié pour la première fois en 2012, soit un an après le début de nos travaux.

3.4.5 Conclusion sur les simulateurs

MAST et Cheddar sont des outils matures et ayant une bonne base d'utilisateurs, cependant, rien n'est prévu pour faciliter la mise en œuvre d'une nouvelle politique. Il n'y a pas de documentation expliquant concrètement comment procéder et il est nécessaire de s'interfacer au cœur du logiciel ce qui n'est pas une tâche simple (besoin 1 non satisfait). De plus, ces outils ayant été conçus dans l'optique d'analyser des systèmes et non d'évaluer des algorithmes d'ordonnancement, peu de choses sont prévues pour la génération de systèmes et l'automatisation des évaluations (besoin 3 non satisfait).

STORM et Yartiss sont les simulateurs les plus proches de nos besoins. De plus, de nombreuses politiques ont déjà été mises en œuvre sur STORM. Malheureusement, le noyau de simulation de STORM ne permet pas de simuler des durées aussi petites que ce dont nous aurions besoin. Modifier la simulation à pas fixe par une simulation à événements discrets nécessiterait de profondes modifications. Yartiss n'a pas ce problème mais sa documentation n'est pas suffisante pour prendre en main rapidement l'outil et développer une nouvelle politique. Cet outil n'a pas été retenu car il a été publié après le début de nos travaux mais nous l'avons inclus pour compléter ce panorama des outils disponibles.

Le tableau 3.1 résume le degré de satisfaction des besoins par les différents outils étudiés.

	Besoin 1	Besoin 2	Besoin 3	Besoin 4
STORM	***	*	**	***
Yartiss	***	*	***	**
Cheddar	*	**	*	**
MAST	*	*	*	**

Tableau 3.1 – Tableau récapitulatif de la satisfaction de nos besoins par les différents outils.

3.5 Conclusion

L'évaluation de politiques d'ordonnancement via des exécutions réelles a naturellement été l'une des premières pistes que nous avons explorées. En effet, d'une part, l'utilisation d'une vraie architecture offre une interface réaliste et d'autre part, les surcoûts et spécificités du système sont pris en compte dans l'évaluation. Notre attention s'est principalement focalisée sur LITMUS^{RT} car de nombreuses politiques d'ordonnancement ont été mises en œuvre sur cet outil. Nous avons cependant jugé que l'outil, dont la prise en main est complexe, ne permettait pas de mettre en œuvre facilement les dizaines de politiques d'ordonnancement dont nous avons besoin. De plus, il n'est pas possible de faire varier les caractéristiques matérielles et isoler certains paramètres. Enfin, nous nous demandons

dans quelle mesure les résultats obtenus dépendent du système d'exploitation utilisé et à quel point les résultats pourraient varier sur un système d'exploitation moins complexe.

Suite à cela, nous avons exploré la possibilité d'utiliser STORM, les autres outils ayant été écartés pour les différentes raisons évoquées précédemment. STORM est déjà utilisé pour mener des campagnes d'évaluation de politiques d'ordonnancement et de nombreux algorithmes ont été mis en œuvre. Mais notre objectif est d'aller plus loin dans la simulation et de prendre en considération les surcoûts liés au système et l'impact de certains éléments de l'architecture matérielle, en particulier les caches. STORM nécessiterait de profondes modifications pour lui permettre de prendre en compte ce niveau de détail dans la simulation. Nous avons jugé que sa réécriture serait finalement au moins aussi complexe que l'écriture à partir de zéro d'un nouvel outil, pensé dès le départ pour satisfaire nos objectifs.

En conséquence, nous avons pour objectif de réaliser un nouveau simulateur d'ordonnanceur qui soit conçu dès le départ pour permettre l'intégration dans la simulation des effets liés au matériel et au système. Nous qualifierons cette approche de simulation à « précision variable ». Le défaut intrinsèque de cette approche est qu'un tel simulateur n'aura jamais la précision d'un système réel. Mais un tel simulateur permettrait un prototypage rapide et ne nécessiterait pas de vraies tâches ni un système d'exploitation. De plus, des expérimentations poussées pourraient être facilement conduites et de nombreuses métriques pourraient en être extraites afin d'analyser les résultats. Ceci devrait être suffisant pour pouvoir obtenir des tendances générales.

Ainsi, notre contribution est un outil de simulation, appelé SimSo (« Simulation of Multiprocessor Scheduling with Overheads »). Les idées que nous avons jugées pertinentes dans les différents simulateurs ont été prises en considération au cours du développement de SimSo. Une présentation détaillée de cet outil et des expérimentations classiques menées avec SimSo sont disponibles dans la partie II de cette thèse. SimSo permet de simuler des coûts fixes lors des appels aux méthodes de l'ordonnanceur ou lors d'un changement de contexte. De plus, un effort particulier a été fait pour permettre de contrôler la durée d'exécution des travaux. Ceci permet notamment d'utiliser des modèles simulant les effets des caches. Les modèles disponibles sont présentés et évalués dans la partie III.

Deuxième partie

Simulation d'ordonnancement

CHAPITRE 4

SimSo : un outil pour la simulation d'ordonnancement

Ce chapitre présente SimSo [1], un simulateur d'ordonnancement conçu pour faciliter l'évaluation et la compréhension de politiques d'ordonnancement temps réel pour architectures multiprocesseurs. SimSo signifie « Simulation of Multiprocessor Scheduling with Overheads ». En effet, l'un des objectifs principaux de SimSo est la prise en compte dans la simulation des surcoûts liés à l'ordonnancement tels que ceux résultant des préemptions, migrations, ou encore des appels aux méthodes de l'ordonnanceur.

Afin de faciliter l'évaluation d'algorithmes d'ordonnancement, SimSo intègre des générateurs de tâches et offre la possibilité d'extraire tout type de métriques depuis une trace de l'exécution du système simulé. L'outil a été conçu dans l'optique de rendre aisée l'implémentation de politiques d'ordonnancement que ce soit des politiques globales, partitionnées ou hybrides. À ce jour, plus de vingt-cinq algorithmes d'ordonnancement ont été mis en œuvre pour SimSo.

Avant de rentrer dans les détails liés à l'architecture interne de SimSo, nous débuterons par une présentation générale de SimSo. Enfin, nous pourrons décrire son architecture et plus précisément comment les paramètres du système sont stockés en mémoire et comment fonctionne le cœur de la simulation. Puisque le cœur de cet outil est l'évaluation des algorithmes d'ordonnancement, une partie sera dédiée à la présentation de l'interface de programmation, des algorithmes déjà mis en œuvre et un exemple permettra de mieux comprendre comment implémenter un algorithme pour SimSo. L'objectif principal étant de pouvoir conduire des évaluations expérimentales des algorithmes, nous verrons quels générateurs de tâches sont disponibles et comment extraire depuis les simulations les données qui peuvent nous intéresser. Ce chapitre se terminera sur une présentation des deux modes d'utilisation possibles de SimSo, à savoir en utilisant l'interface graphique ou sous la forme d'un module Python.

4.1 Présentation générale

SimSo est un simulateur d'ordonnancement de tâches temps réel. Son niveau de simulation est ce que nous qualifions de « grain intermédiaire », c'est-à-dire que nous cherchons un ni-

1. Disponible à l'adresse : `http://homepages.laas.fr/mcheramy/simso/`

veau de simulation qui soit suffisamment précis pour observer les comportements souhaités sans pour autant simuler les détails d'une architecture réelle. Ceci permet notamment de gagner en efficacité calculatoire et en flexibilité vis-à-vis des systèmes étudiés. L'objectif de SimSo étant d'évaluer les performances des politiques d'ordonnancement, nous nous focalisons principalement sur le comportement temporel des tâches et des composants de l'ordonnanceur. Contrairement à des simulateurs d'architecture, aucun code n'est exécuté par une tâche, elle n'est représentée dans SimSo qu'à travers sa durée d'exécution qui sera évaluée à partir d'un ensemble de caractéristiques. Les événements d'ordonnancement sont quant à eux bien explicités dans la simulation et sont aussi caractérisés temporellement.

SimSo a été conçu de manière modulaire dans le but de pouvoir accueillir de nouveaux modèles et d'adapter la simulation aux besoins de ses utilisateurs. Bien que les travaux présentés dans cette thèse se concentrent sur des tâches périodiques, il est également possible de simuler des tâches sporadiques ou des tâches apériodiques dont l'activation est provoquée par la fin d'exécution d'un autre travail. De plus, SimSo offre la possibilité de contrôler les durées d'exécution des travaux (voir partie 4.2.4). Ce contrôle des durées d'exécution permet également de tenir compte des caractéristiques de l'architecture matérielle simulée.

La partie 4.6 présente les deux modes d'utilisation possibles de SimSo. À partir d'une interface graphique qui se veut être facile d'utilisation ou sous la forme d'un module Python ce qui permet un maximum de souplesse pour automatiser de multiples simulations et pour traiter les données qui en sont issues.

Le développement de SimSo a représenté une part importante du travail fourni durant cette thèse. L'écriture du simulateur en lui-même a nécessité du temps, mais l'investissement majeur a concerné la mise en œuvre de toutes les politiques que SimSo peut simuler aujourd'hui. La partie 4.3 présente la liste des ordonnanceurs disponibles et certaines difficultés rencontrées au cours de leur implémentation. À notre connaissance, aucun autre simulateur d'ordonnanceur ne dispose d'une telle bibliothèque d'ordonnanceurs.

Les travaux qui suivent ont été présentés à de multiples reprises au niveau local (présentations au LAAS [2], à l'ONERA [3] ou encore au Congrès des doctorants EDSYS [4]), au niveau national (École d'été Temps Réel [CHD13]) et enfin international (SIMULTECH 2013 [CDH13], WiP SIES 2014 [CHDD14], WATERS 2014 [CHD14] et session poster de ECRTS 2014).

4.2 Architecture

4.2.1 Simulation à événements discrets

Le cœur de SimSo repose sur un mécanisme de simulation à événements discrets. Contrairement à une simulation à pas fixes, la technique de simulation à événements discrets permet de disposer d'une grande précision et flexibilité dans la date des événements de la simulation. Ce choix s'est imposé par la nécessité de pouvoir traiter certains surcoûts affectant l'exécution des tâches et dont la durée est largement inférieure aux durées d'exécution et périodes des tâches (besoins 3.1 et 3.2).

2. Séminaire du thème Informatique Critique en 2013.
3. Séminaire DTIM en 2013. `http://seminaire-verif.enseeiht.fr/`
4. Congrès 2013, organisé à Tarbes. `http://edsys13.sciencesconf.org/`

Afin de réaliser le noyau du simulateur, nous avions le choix entre réaliser nous-même un moteur dédié à nos besoins, ou utiliser l'une des bibliothèques ou *frameworks* disponibles [WGG10]. Contrairement aux développeurs de Yartiss qui ont fait le choix d'implémenter tout le moteur de simulation, nous avons décidé d'utiliser la bibliothèque SimPy pour sa facilité de prise en main et parce qu'il répond parfaitement à nos attentes. SimPy 2.3 [Sim12] est une bibliothèque Python de simulation à événements discrets à base de processus. Cette représentation sous forme de processus permet de modéliser de façon intuitive le comportement du système simulé. Le choix de cette bibliothèque s'est fait après une évaluation des autres solutions disponibles et c'est ce qui a ensuite imposé le choix du langage Python pour l'écriture de SimSo.

Dans le vocabulaire de SimPy, inspiré du langage Simula, un processus est une entité capable d'attendre un signal, une condition, ou un certain temps. Lorsqu'un processus est actif, c'est-à-dire lorsqu'il n'est pas en phase d'attente, il peut exécuter du code, envoyer des signaux et réveiller d'autres processus. Tout processus est activé par un autre processus ou par l'objet principal de simulation.

Bien qu'il soit possible d'avoir des évènements à des dates exprimées sous la forme de nombres décimaux, le choix a été fait de se limiter à des entiers pour éviter d'éventuelles erreurs d'arrondi. Ainsi, l'unité interne de simulation est le cycle et représente un cycle processeur. Cependant, les attributs des tâches tels que la période ou l'échéance sont définis en millisecondes (nombres décimaux). La conversion entre cycles et millisecondes se fait alors par une constante paramétrable appelée *cycles_per_ms*.

4.2.2 Structures de données pour les paramètres de simulation

Avant de s'intéresser aux classes servant à la simulation du système, les classes qui permettent de mémoriser les caractéristiques des tâches, des processeurs et de tous les paramètres utiles pour la simulation sont présentés. La figure 4.1 donne un premier aperçu de ces classes.

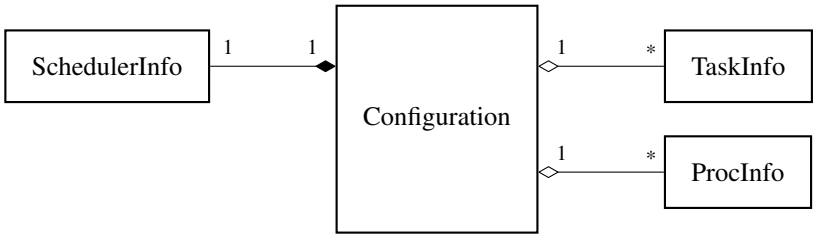


FIGURE 4.1 – Diagramme des classes mémorisant les caractéristiques du système à simuler.

a) TaskInfo

La classe *TaskInfo* est une classe servant uniquement à contenir les caractéristiques d'une tâche. Les principales informations pour une tâche périodique sont présentées dans le tableau 4.1. D'autres informations sont également présentes et utiles pour certains *Execution Time Models* (voir partie 4.2.4) ou en fonction des modèles de tâches utilisés (exemple : liste des dates d'activation pour une tâche sporadique). Enfin, un dictionnaire *data* offre la possibilité à l'utilisateur d'ajouter des attributs supplémentaires avec le nom et le type de son choix. Ces informations seront alors accessibles par l'ordonnanceur et pourront par exemple servir à fixer des priorités ou guider un partitionnement.

Attribut	Description	Type
name	Nom de la tâche	string
identifier	Identifiant unique	int
task_type	Type de la tâche (périodique par défaut)	string
period	Période de la tâche	float
activation_date	Date de première activation	float
wcet	Worst Case Execution Time	float
deadline	Échéance relative de la tâche	float
abort_on_miss	Terminaison du travail si dépassement d'échéance	boolean
data	Dictionnaire contenant des attributs supplémentaires	dict

Tableau 4.1 – Principaux attributs de la classe *TaskInfo*

b) ProcInfo

À l'instar de la classe *TaskInfo*, la classe *ProcInfo* permet de stocker les informations relatives à un processeur. Le nombre de paramètres est réduit (Tableau 4.2) mais peut également accueillir des attributs supplémentaires. Lors de l'utilisation d'un *Execution Time Model* simulant l'effet des caches, une liste de caches est également fournie.

Attribut	Description	Type
name	Nom du processeur	string
identifier	Identifiant unique	int
cs_overhead	Surcoût lors de la sauvegarde d'un contexte	int
cl_overhead	Surcoût lors du rechargement d'un contexte	int
speed	Vitesse (initiale) du processeur	float
data	Dictionnaire contenant des attributs supplémentaires	dict

Tableau 4.2 – Principaux attributs de la classe *ProcInfo*

c) SchedulerInfo

La classe *SchedulerInfo* permet de stocker les informations qui concernent l'ordonnanceur. Ces informations sont par exemple le chemin d'accès vers la classe Python de l'ordonnanceur, les surcoûts à appliquer lors d'une prise de décision de l'ordonnanceur, de l'activation d'une tâche ou de la terminaison d'une tâche. Tout comme les deux autres classes présentées ci-dessus, il est également possible d'ajouter des champs supplémentaires. Un exemple d'utilisation est le choix du paramètre K utilisé par l'ordonnanceur EKG.

Le tableau 4.3 résume la liste des attributs disponibles. Mais contrairement aux classes précédentes, cette classe possède des méthodes pour permettre le chargement dynamique du code de l'ordonnanceur et l'instanciation de la classe ordonnanceur au moment de la simulation.

d) Configuration

La classe *Configuration* regroupe l'ensemble des informations sur le système. On y retrouve donc la liste des tâches (*TaskInfo*), des processeurs (*ProcInfo*), les informations sur l'ordonnanceur (*SchedulerInfo*), ainsi que les informations générales sur la simulation comme

Attribut	Description	Type
name	Nom de l'ordonnanceur (dérivé du nom du fichier)	string
filename	Chemin, absolu ou relatif, vers la classe de l'ordonnanceur	string
overhead	Temps nécessaire pour exécuter la méthode schedule	int
overhead_activate	Temps pour exécuter la méthode on_activate	int
overhead_terminate	Temps pour exécuter la méthode on_terminated	int
data	Dictionnaire contenant des attributs supplémentaires	dict

Tableau 4.3 – Principaux attributs de la classe *SchedulerInfo*

la durée de la simulation, le nombre de cycles par millisecondes ou encore le modèle de temps d'exécution utilisé (voir partie 4.2.4).

Les modules *GenerateConfiguration* et *parser* permettent respectivement l'enregistrement et le rechargement de la classe *Configuration* en s'appuyant sur un fichier XML. La figure 4.2 présente un exemple de fichier XML manipulé par SimSo. Le format XML utilisé par SimSo est en partie inspiré par celui de STORM ce qui devrait par conséquent faciliter une éventuelle migration d'un outil à l'autre. Un schéma XSD [5] est proposé avec les sources afin de faciliter l'écriture d'outils capables de générer des fichiers lisibles par SimSo.

```
<?xml version="1.0" ?>
<simulation cycles_per_ms="1000000" duration="1000000000" etm="wcet">
    <sched className="../schedulers/RUN.py" overhead="0" overhead_activate="0"
        overhead_terminate="0"/>
    <processors>
        <processor id="1" cl_overhead="0" cs_overhead="0" name="CPU 1"/>
        <processor id="2" cl_overhead="0" cs_overhead="0" name="CPU 2"/>
    </processors>
    <tasks>
        <task id="1" task_type="Periodic" name="T1" WCET="2.5" period="11.0"
            abort_on_miss="yes" activationDate="0" deadline="11.0" />
        <task id="2" task_type="Periodic" name="T2" WCET="4" period="8.0"
            abort_on_miss="yes" activationDate="0" deadline="8.0" />
        <task id="3" task_type="Periodic" name="T3" WCET="2.1" period="3.4"
            abort_on_miss="yes" activationDate="0" deadline="3.4" />
        <task id="4" task_type="Periodic" name="T4" WCET="7.6" period="10.1"
            abort_on_miss="yes" activationDate="0" deadline="10.1" />
    </tasks>
</simulation>
```

Figure 4.2 – Exemple de configuration sauvegardée au format XML pour SimSo. Les attributs non spécifiés reçoivent des valeurs par défaut.

4.2.3 Classes pour la simulation

Cette partie présente les principales classes qui permettent la simulation du système. Les principaux éléments sont représentés sur la figure 4.3. Ces éléments sont les classes : *Model*, *Processor*, *Task*, *Job*, *Timer* et *Scheduler*. Les objets instanciés à partir des classes *Processor*, *Task*, *Job* et *Timer* sont des processus au sens de SimPy, c'est-à-dire des entités actives qui peuvent attendre des événements ou exécuter du code. La classe *Model* est le

5. Le langage XSD, pour *XML Schema Definition*, permet de spécifier les éléments d'un fichier XML. Contrairement aux fichiers DTD, il est possible de préciser les types des données contenues dans les attributs.

point d'entrée de la simulation. C'est elle qui est responsable d'instancier et de lancer les différents processus du système. Enfin, la classe *Scheduler* est fournie par l'utilisateur et ses méthodes sont appelées par les objets *Processor*. Bien entendu, l'utilisateur de SimSo pourra également utiliser l'un des ordonnanceurs déjà existants que nous mettons à disposition (voir partie 4.3.3).

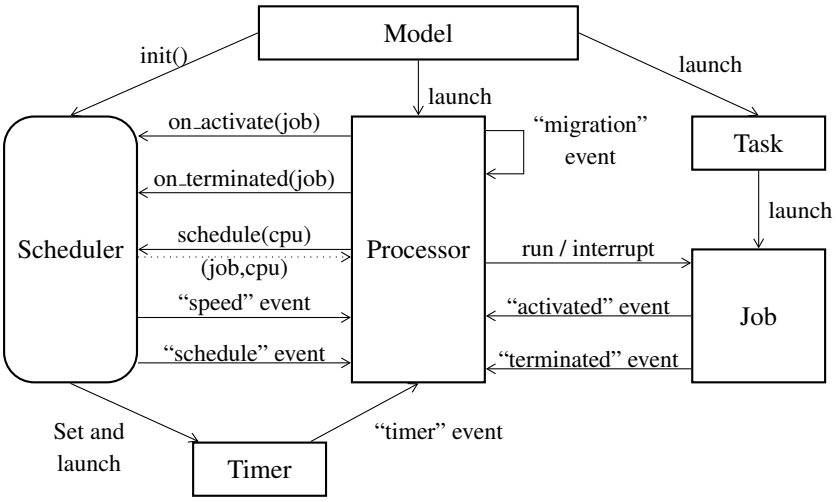


FIGURE 4.3 – Interactions entre les principales instances de classe. *Processor*, *Task*, *Job* et *Timer* sont des processus au sens de SimPy et peuvent avoir plusieurs instances.

a) Model

Dans le fonctionnement de SimPy, la classe *Simulation* permet d'offrir un environnement de simulation pour l'ensemble des processus. La classe *Model* étend cette classe (au sens de la Programmation Orientée Objet), et elle instancie les processeurs, les tâches, l'ordonnanceur et l'ETM. Cet environnement est passé en argument à chaque processus (entité active) ainsi qu'à l'ordonnanceur et à l'ETM au moment de leur création. C'est à travers cette classe que l'ordonnanceur peut obtenir la date courante et la durée de la simulation.

b) Task

Chaque tâche simulée est modélisée par une instance d'une classe fille de la classe abstraite *GenericTask*. La figure 4.4 montre les différents types de tâches actuellement supportés par SimSo. Leur rôle est de gérer la création des travaux et éventuellement de les interrompre en cas de dépassement d'échéance (paramètre *abort_on_miss*).

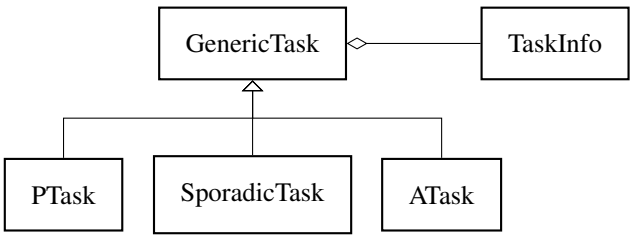


FIGURE 4.4 – Diagramme des classes de tâches.

La classe *Task* que l'on peut observer sur la figure 4.3 est en réalité une *Fabrique* (patron de conception) qui instancie une classe fille de *GenericTask* en se basant sur les caracté-

ristiques de la tâche et en particulier l'attribut *task_type* de l'objet *TaskInfo* qui est passé en argument. Cet objet *TaskInfo* est disponible dans l'objet *GenericTask* ce qui permet d'accéder aux caractéristiques de la tâche.

c) Job

Chaque travail est représenté par une instance de la classe *Job*. Cette classe simule, d'un point de vue temporel, l'exécution de la tâche. Dans la partie 4.2.4, nous verrons que SimSo était capable de simuler la durée d'exécution d'un travail de différentes manières.

La figure 4.3 permet de voir les interactions entre un objet de type *Job* et le reste du système. Comme cela a été dit précédemment, la création et l'activation d'un travail se font par la tâche. Ensuite, l'exécution est contrôlée par le processeur sur lequel s'exécute le travail. Enfin, lorsque le travail se termine, le processeur sur lequel il s'exécutait en est informé. C'est le processeur qui se chargera ensuite de transmettre l'information à l'ordonnanceur.

d) Processor

Chaque processeur physique est représenté par une instance de la classe *Processor*. Cette classe a un rôle central dans la simulation du système. En effet, cette classe se charge de simuler le comportement du système d'exploitation et du processeur. Le processeur contrôle l'état des travaux (en exécution ou en attente) en fonction des décisions de l'ordonnanceur. Le processeur s'occupe aussi d'appeler les méthodes de l'ordonnanceur pour l'informer des événements liés à l'ordonnancement et pour obtenir des décisions d'ordonnancement.

Cette centralisation au niveau de la classe *Processor*, qui est aussi un processus, permet de simuler les surcoûts système tels que les changements de contexte ou les appels à l'ordonnanceur. Ainsi, tout code simulé s'exécute virtuellement sur un processeur, que ce soit un travail ou un service du système d'exploitation (typiquement une fonction de l'ordonnanceur).

Le comportement d'un processeur est représenté de façon simplifiée par la figure 4.5. Si aucun évènement d'ordonnancement n'est à traiter, alors le processeur s'exécute normalement (branches 2 et 3). Dans ce cas, si un travail actif est assigné à un processeur, alors celui-ci s'exécute. Sinon le processeur est simplement mis en attente. Lorsqu'un signal est reçu par le processeur, alors celui-ci interrompt l'exécution éventuelle du travail en cours pour traiter cet évènement. Dans ce cas (branche 1), le processeur simule l'exécution du code du système d'exploitation sur le processeur. Il est alors tout à fait possible d'associer des délais aux appels des méthodes de l'ordonnanceur et même de créer des zones critiques si on veut éviter que deux processeurs puissent appeler l'ordonnanceur en parallèle.

La figure 4.3 montre les différents types d'évènements qui sont susceptibles d'interrompre l'exécution d'un travail. Certains proviennent des travaux qui signalent lorsqu'ils sont activés et lorsque leur exécution se termine. D'autres proviennent de l'ordonnanceur qui peut demander à un processeur d'exécuter sa méthode *schedule*, ou encore de changer de vitesse d'exécution. Un *Timer* peut également se déclencher et interrompre un processeur. Et enfin, un processeur peut interrompre un autre processeur pour appliquer une décision d'ordonnancement (*inter-processor interrupt*, IPI).

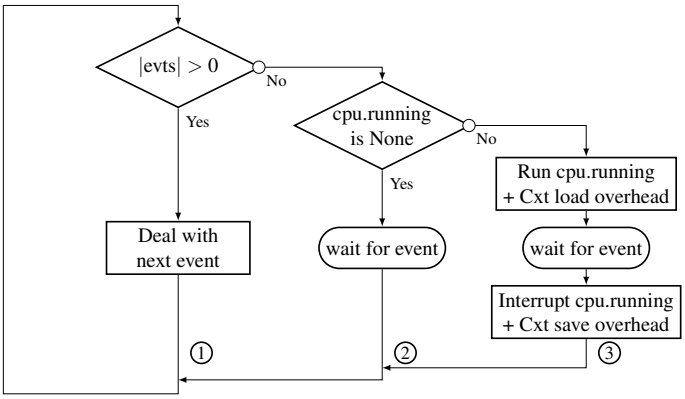


FIGURE 4.5 – Schéma d'exécution simplifié d'un *Processor*.

e) Timer

Dans une architecture matérielle classique, un timer est un composant électronique qui déclenche une interruption au bout d'un certain délai. Une fonction du système d'exploitation est appelée à la réception de cette interruption. Certains ordonnanceurs peuvent avoir besoin de timers pour prendre des décisions à des instants précis qui ne correspondent pas à des évènements de type activation ou fin de travail.

SimSo propose, à travers la classe *Timer*, un mécanisme similaire. Cette classe permet de gérer des appels à l'ordonnanceur après une certaine durée, de façon périodique ou non. Le processeur sur lequel doit s'exécuter la routine est décidé par l'utilisateur au moment de la création du timer. Ce choix peut avoir une importance sur les performances du système de par les surcoûts qui sont engendrés. En effet, comme sur un système réel, le déclenchement du timer interrompt l'activité courante d'un processeur pour exécuter la routine liée à cette interruption. Ceci peut donc potentiellement interrompre l'exécution d'un travail et par conséquent provoquer un changement de contexte qui sera simulé.

f) Scheduler

Contrairement aux autres objets présentés ci-dessus, l'ordonnanceur, représenté par la classe *Scheduler*, n'est pas un processus. L'ordonnanceur est une classe passive appelée par des objets *Processor* lors d'activations ou terminaisons de tâches, par des déclenchements de timers ou lorsqu'il est demandé de prendre une décision d'ordonnancement. Cette classe est fournie par l'utilisateur (qui pourra éventuellement réutiliser un ordonnanceur disponible en bibliothèque) et son interface est détaillée dans la partie 4.3.

4.2.4 Modèle de temps d'exécution

Suite aux observations faites dans la partie 3.1.4, nous avons voulu faire en sorte que la durée d'exécution des travaux puisse être facilement adaptée en fonction des besoins de l'expérimentation. Dans SimSo, ceci est pris en compte à travers ce que nous appelons l'*Execution Time Model* (ETM). Les modèles actuellement disponibles sont listés dans le tableau 4.4. Un effort particulier a été apporté pour isoler cette partie du reste de la simulation afin de faciliter la mise en place de nouveaux modèles.

Nom	Description
WCET	La durée d'exécution des travaux est toujours égale au WCET si la vitesse du processeur est de 100%. La durée d'exécution s'adapte proportionnellement à la vitesse du processeur.
ACET	Similaire au model WCET sauf le WCET est substitué par une valeur tirée aléatoirement selon une distribution normale centrée autour du ACET puis borné par le WCET.
Fixed Penalty	Similaire au modèle WCET sauf que la durée d'exécution est prolongée par une pénalité fixe après chaque préemption.
Cache Model	Utilisation de modèles statistiques pour l'évaluation de la durée d'exécution en tenant compte des caches (partagés ou non).

Tableau 4.4 – Liste des *Execution Time Models* disponibles.

Il n'existe qu'un seul objet ETM pour tout le système et celui-ci n'est pas non plus un processus. Cet objet est créé par la classe *Model* avant d'être passé en argument aux objets *Job* au moment de leur création. L'objet ETM est informé par les travaux de tous les évènements relatifs à leur exécution (voir figure 4.6). À partir de ces informations, cet objet sera capable de fournir à tout travail, une borne inférieure de sa durée d'exécution restante. Notons que l'objet ETM ne communique qu'avec les travaux dans un souci de pouvoir permettre l'ajout de nouveaux modèles sans avoir à modifier les classes qui forment le cœur de la simulation.

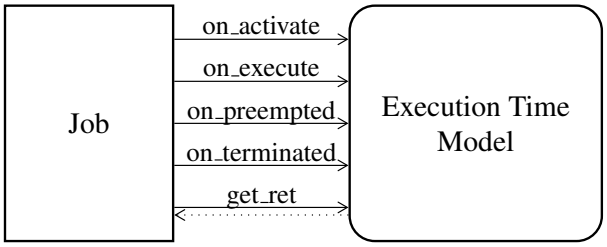


Figure 4.6 – Interface d'un *Execution Time Model*.

Certains ETM ne permettent pas de déterminer à l'avance quelle sera la durée d'exécution d'un travail. C'est par exemple le cas du modèle prenant en compte l'impact du partage de cache entre plusieurs tâches s'exécutant en parallèle puisqu'au moment où on exécute le travail, on ne sait pas encore quels travaux seront exécutés en même temps. Deux solutions se sont alors présentées. La première était de considérer, au moment de l'exécution d'un travail, que l'état du système ne changerait pas d'ici à la fin de l'exécution du travail. En cas de changement, il suffirait alors de réévaluer les dates des évènements de fin d'exécution des travaux. La seconde solution consistait à exécuter les travaux pour une durée inférieure à la durée réelle, puis de recommencer jusqu'à ce que l'exécution soit finie.

La première approche est intéressante mais le fonctionnement de SimPy ne la rend pas aisée à mettre en œuvre. Il faudrait pour cela envoyer une interruption à tous les processus *Job* puis les relancer pour la bonne durée. Et ceci à chaque changement dans le système (exécution d'un travail, fin d'exécution, timer, etc). La seconde solution est apparue plus simple à mettre en place et c'est pour cette raison qu'elle a été choisie. L'inconvénient de cette solution est que si la borne inférieure renvoyée est trop éloignée de la réalité, de nombreux appels seront nécessaires. La convergence est malgré tout garantie puisque l'unité de base de la simulation est entière. Si finalement ceci venait à poser problème, il n'est pas impossible de mettre en œuvre l'autre solution et seul le modèle qui simule les

effets des caches serait à modifier [6].

Ainsi, chaque travail simulant son exécution fait un appel à la méthode *get_ret* de l'ETM afin d'obtenir une borne inférieure de son temps restant d'exécution. Lorsque ce temps est écoulé, le travail appelle à nouveau cette méthode et ainsi de suite jusqu'à ce qu'elle retourne zéro.

Utilisation d'ETM pour la simulation d'une architecture DVFS

Grâce aux *Execution Time Models*, il est possible de simuler le changement de vitesse de fonctionnement d'un processeur ce qui permet d'étudier des algorithmes exploitant le *Dynamic Voltage and Frequency Scaling* (DVFS). Pour l'heure, seuls les modèles WCET, ACET et FixedPriority prennent en compte la vitesse du processeur. La durée d'exécution des travaux est proportionnelle à la vitesse du processeur, mais il est envisageable d'en changer le comportement.

L'ordonnanceur qui est simulé peut changer la vitesse de fonctionnement d'un processeur en lui envoyant un signal (signal « speed » sur la figure 4.3). Lorsque le processeur reçoit ce signal, et si un travail est en cours d'exécution sur celui-ci, il commence par interrompre le travail. Ensuite la vitesse du processeur est changée avant que le travail ne reprenne son exécution. Au moment de reprendre son exécution, le travail commence par appeler la méthode *get_ret* qui pourra prendre en compte la nouvelle vitesse pour calculer la durée d'exécution.

4.3 Ordonnanceurs

4.3.1 Fonctionnement et interface d'un ordonnanceur

Un ordonnanceur pour SimSo se présente sous la forme d'une classe Python qui hérite de la classe abstraite *Scheduler* (voir Figure 4.7). Cette classe est chargée dynamiquement au moment de la simulation afin de permettre à l'utilisateur de fournir l'ordonnanceur de son choix. Le chemin vers le fichier contenant la classe à charger doit pour cela être fourni à la classe *SchedulerInfo* comme cela a été expliqué précédemment dans la partie 4.2.2. Une autre classe abstraite, *PartitionedScheduler*, dérive de la classe *Scheduler* et facilite la mise en œuvre d'ordonnanceurs utilisant une approche partitionnée en permettant de créer un ordonnanceur partitionné à partir d'un ordonnanceur monoprocesseur et d'une méthode d'affectation des tâches aux processeurs.

L'interface de l'ordonnanceur (au sens des services disponibles) est inspirée de ce qui peut être trouvé sur des systèmes réels, tout en restant simple. Cette similarité avec des systèmes réels a pour but de mettre en évidence des problèmes pratiques concernant l'implémentation et qui ne sont pas pris en considération dans des études théoriques. Cette interface permet de développer des politiques de type partitionné, global ou hybride.

Les méthodes de la classe *Scheduler* listées ci-contre (voir figure 4.7) doivent être implémentées par la classe fille qui sera instanciée. Seule la méthode *init* devra être implémentée lors de l'utilisation de la classe *PartitionedScheduler*.

6. En effet, la durée renvoyée par la méthode *get_ret* par les autres modèles correspond également à la durée restante d'exécution si aucun changement dans le système ne se produit.

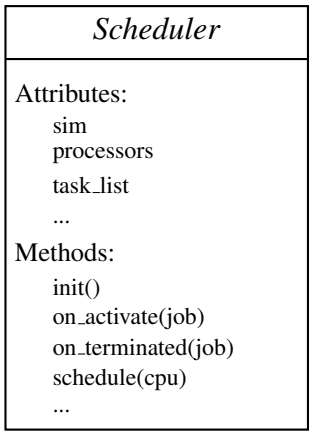


FIGURE 4.7 – Classe abstraite *Scheduler*.

- **init :** lorsque la simulation est démarrée, la méthode *init* est appelée. Ceci permet d'initialiser les structures de données de l'ordonnanceur et éventuellement configurer des timers.
- **on_activate :** cette méthode est appelée lorsqu'un travail est activé.
- **on_terminated :** cette méthode est appelée lorsque l'exécution d'un travail se termine ou lorsque celui-ci est avorté.
- **schedule :** cette méthode est appelée lorsqu'une décision d'ordonnancement doit être prise. Elle retourne la décision d'ordonnancement (un ou plusieurs couples (travail, processeur)) qui sera appliquée par le processeur.

D'autres méthodes peuvent également être implémentées telles que les méthodes *get_lock* et *release_lock* qui sont appelées avant et après l'appel à la méthode *schedule* et qui par défaut empêche l'exécution de la méthode *schedule* sur deux processeurs en même temps.

L'ordonnanceur n'étant pas un processus (au sens de SimPy), toutes ses méthodes sont appelées par un objet de type *Processor* (en dehors de la méthode *init*). Lorsque l'ordonnanceur doit prendre une décision d'ordonnancement (suite à l'arrivée d'un nouveau travail par exemple), alors ce dernier ne modifie pas l'affectation des tâches sur les processeurs directement mais informe un processeur de la nécessité d'exécuter la méthode *schedule*. Afin d'informer un processeur, l'ordonnanceur doit envoyer un signal « schedule » au processeur de son choix. Ce signal peut être émis lors de l'activation d'un travail, d'une terminaison ou dans la routine exécutée par un timer (une méthode de l'ordonnanceur). Si le processeur en question était en train d'exécuter un travail, alors ce dernier est interrompu le temps de l'appel à l'ordonnanceur. Le processeur est en charge d'appliquer la décision d'ordonnancement renvoyée par cette méthode. Si cette décision concerne un autre processeur, un signal sera envoyé à celui-ci ce qui pourra provoquer une interruption (il s'agit donc d'une interruption inter-processeurs).

Cette approche permet de simuler le fait que le code de l'ordonnanceur s'exécute en réalité sur un processeur physique et qu'il n'a pas un coût nécessairement nul. Par conséquent, cela oblige à réfléchir au choix du processeur à utiliser et cela permet de prendre facilement en compte dans la simulation les coûts liés aux appels des méthodes de l'ordonnanceur. En revanche, la méthode *on_terminated* est systématiquement appelée par le processeur qui exécutait la tâche. Il est possible de contrôler le processeur qui exécute la méthode *on_activate* en modifiant l'attribut *cpu* de la tâche lors de la terminaison du dernier travail ou dans la méthode *init* pour l'activation du premier travail.

4.3.2 Difficultés liées à la mise en œuvre d'ordonnanceurs

Les articles qui présentent les politiques d'ordonnancement se focalisent généralement sur les aspects théoriques en mettant de côté les aspects liés à une possible implémentation. Il est alors nécessaire de comprendre parfaitement la politique pour pouvoir la mettre en œuvre tout en s'assurant d'avoir bien respecté ce qui est présenté. Il est fréquent que certains points n'ayant pas d'impact sur l'ordonnançabilité du système soient omis par les auteurs. L'exemple le plus courant est la procédure pour affecter les tâches sélectionnées pour l'exécution aux processeurs qui n'a pas d'importance sous l'hypothèse d'un coût de migration nul. Or, ceci a un impact majeur sur le nombre de migrations.

De nombreux tests faits à partir de systèmes générés automatiquement ont permis de vérifier le bon comportement des ordonnanceurs. Des problèmes liés à la mise en pratique de certaines politiques ont alors surgi. À titre d'exemple, des politiques DP-Fair, théoriquement optimales, sont incapables d'ordonnancer certains systèmes chargés à 100% à cause d'erreurs d'arrondi [7] dans les divisions et certains choix d'implémentation sont plus ou moins robustes à des situations théoriquement impossibles. Le problème le plus fréquent que nous avons rencontré est lorsque l'ordonnanceur n'accorde plus de temps d'exécution à une tâche qui était sur le point de se terminer (à quelques microsecondes ou nanosecondes selon la précision utilisée) ce qui provoque un dépassement d'échéance. Mais dans d'autres cas, la présence dans le système d'une tâche active qui aurait théoriquement dû se terminer ou une fin de tâche légèrement trop tardive, peut provoquer des incohérences graves pouvant engendrer des boucles infinies ou l'absence d'exécution de tâches qui auraient dû s'exécuter [8]. Ces problèmes n'apparaissent que lorsqu'un travail s'exécute pendant tout son WCET ce qui est le cas avec l'ETM qui utilise le WCET. Cependant, ceci ne devrait pas poser de difficulté pour un système réel où le WCET n'est en pratique jamais atteint.

4.3.3 Ordonnanceurs disponibles

Une part importante du travail de thèse a été de constituer une large bibliothèque de classes d'ordonnanceur pour SimSo. Elle comprend actuellement plus de vingt-cinq algorithmes d'ordonnancement. Toutes les catégories d'ordonnanceurs sont ainsi représentées : en plus des politiques monoprocesseurs, il y a des politiques multiprocesseurs partitionnées, globales et hybrides dont la liste est donnée ci-après.

a) Ordonnancement monoprocesseur

Nom	Abréviation
Earliest Deadline First	EDF
Rate Monotonic	RM
Fixed Priority	FP

7. Exemple : si sur un intervalle de temps de 10ms, une tâche se voit affecter 1/3 du temps. Alors elle doit s'exécuter pendant 3.33...ms. En pratique elle s'exécutera donc très légèrement trop ou pas assez selon l'arrondi.

8. La simulation est interrompue en cas de décision incohérente telle que le placement d'un travail sur deux processeurs ou le choix d'un travail déjà terminé. En cas de boucle infinie, c'est à l'utilisateur d'interrompre manuellement la simulation.

b) Ordonnancement monoprocesseur avec DVFS

Nom	Abréviation
Static-EDF [PS01]	Static-EDF
Cycle-Conserving EDF [PS01]	CC-EDF

c) Ordonnancement global multiprocesseur par généralisation des algorithmes monoprocesseur

Nom	Abréviation
Global-EDF	G-EDF
Global-RM	G-RM
Earliest Deadline Zero Laxity [Lee94]	EDZL
Global-Least Laxity First [Mok83]	G-LLF
Modified Least Laxity First [OY98]	G-MLLF
Priority-Driven / EDF$^{(k)}$ [GBF02]	PriD
EDF-US [SB02]	EDF-US
Global-Fair Lateness [EA12]	G-FL
U-EDF [NBN^{+}12]	U-EDF

d) Ordonnancement multiprocesseur par approche partitionnée

Il est possible de coupler tout ordonnanceur monoprocesseur à un algorithme d'affectation des tâches aux processeurs à l'aide de la classe *PartitionedScheduler*. L'algorithme d'affectation peut être implémenté spécifiquement ou réutiliser l'une des méthodes disponibles dans SimSo :

- Manuel, en utilisant un paramètre de la tâche [9] qui précise le processeur qui lui est affecté ;
- First-Fit et Decreasing-First-Fit ;
- Next-Fit et Decreasing-Next-Fit ;
- Best-Fit et Decreasing-Best-Fit ;
- Worst-Fit et Decreasing-Worst-Fit.

e) Ordonnancement global de type PFair

Nom	Abréviation
Earliest Pseudo-Deadline First [AS00b]	EPDF
PD^2 [AS99]	PD^2
ER-PD^2 [AS00a]	ER-PD^2

9. Comme vu précédemment, il est possible d'affecter aux tâches des attributs supplémentaires, utilisables par l'ordonnanceur.

f) Ordonnancement global de type DPFair

Nom	Abréviation
LLREF (ou LNREF) [CR06]	LLREF
LRE-TL [FN09]	LRE-TL
DP-WRAP [LFS+10]	DP-WRAP
BF [ZMM03]	BF
NVNLF [FKY08]	NVNLF

g) Ordonnancement multiprocesseur par approche semi-partitionnée ou hybride

Nom	Abréviation
EKG [AT06]	EKG
EDHS [KY08d]	EDHS
RUN [RLM+11]	RUN

4.3.4 Exemple

Cette partie présente, à titre d'exemple, le code d'un ordonnanceur G-EDF pour SimSo. Le code source qui suit constitue l'une des possibles implémentations de cet algorithme (voir figure 4.8).

Lors de l'activation d'un travail, le travail est ajouté à la liste des travaux prêts et disponibles pour être exécutés (ligne 15). Lors d'une activation ou terminaison de travail, un signal « schedule » est envoyé au processeur sur lequel s'est exécutée pour la dernière fois la tâche [10], afin d'invoquer indirectement l'appel à la méthode *schedule* (lignes 18 et 25).

Aux lignes 38 à 41, l'algorithme choisit un processeur libre ou le processeur exécutant le travail ayant la plus faible priorité, c'est-à-dire le travail dont l'échéance absolue est la plus éloignée.

À la ligne 44, le travail prêt avec la plus grande priorité est sélectionné. Et aux lignes 48 et 49, l'ordonnanceur compare la priorité de ce travail avec celle du travail le moins prioritaire qui s'exécute. Si le travail en attente est plus prioritaire alors il remplace celui s'exécutant. Dans ce cas, le travail préempté est replacé dans la liste des travaux prêts (ligne 52) et celui qui était en attente est retiré de la liste des travaux prêts (ligne 50).

10. Dans le cas de la première activation d'une tâche, la tâche est positionnée sur le premier processeur. Ce comportement est personnalisable en modifiant manuellement l'attribut *cpu* des tâches dans le méthode *init*.

```
1  from core import Scheduler
2
3  class G_EDF(Scheduler):
4    """
5    Global Earliest Deadline First
6    """
7    def init(self):
8      self.ready_list = []
9
10   def on_activate(self, job):
11     """
12     Méthode appelée lors de l'activation d'un travail.
13     job: L'objet de type Job qui vient de s'activer.
14     """
15     self.ready_list.append(job)
16     # Envoi un signal "schedule" sur le dernier processeur sur
17     # lequel s'est execute la tâche.
18     job.cpu.resched()
19
20   def on_terminated(self, job):
21     """
22     Méthode appelée lors de la terminaison d'un travail.
23     job: L'objet de type Job qui vient de se terminer.
24     """
25     job.cpu.resched()
26
27   def schedule(self, cpu):
28     """
29     Méthode appelée par le processeur cpu afin d'obtenir une décision
30     d'ordonnancement.
31     La méthode doit retourner un ou plusieurs couples (job, processor).
32     """
33     decision = None # No change.
34
35     if self.ready_list:
36       # Get a free processor or the processor running the job with
37       # the latest deadline.
38       key = lambda x: (
39                1 if not x.running else 0,
40                x.running.absolute_deadline if x.running else 0)
41       cpu_min = max(self.processors, key=key)
42
43       # Get the job with the highest priority within the ready list.
44       job = min(self.ready_list, key=lambda x: x.absolute_deadline)
45
46       # If the selected job has a higher priority than the one
47       # running on the selected cpu:
48       if (cpu_min.running is None or
49             cpu_min.running.absolute_deadline > job.absolute_deadline):
50         self.ready_list.remove(job)
51         if cpu_min.running:
52           self.ready_list.append(cpu_min.running)
53         # Schedule job on cpu_min.
54         decision = (job, cpu_min)
55
56     return decision
```

FIGURE 4.8 – Exemple d'implémentation d'un ordonnanceur G-EDF pour SimSo.

4.4 Génération de tâches

L'évaluation empirique du comportement de politiques d'ordonnancement passe par la génération d'ensembles de tâches. Bini et Buttazzo ont montré comment la génération d'ensembles de tâches peut biaiser les résultats expérimentaux dans le cadre d'architectures monoprocesseurs [BB05]. Pour des tâches périodiques et indépendantes, il est possible d'agir sur les taux d'utilisation, les périodes, les échéances, l'utilisation totale et le nombre de tâches. Dans le cas multiprocesseur, plusieurs approches sont utilisées pour générer des ensembles de tâches.

Généralement, pour générer un ensemble de tâches, les algorithmes débutent par deux étapes distinctes. La première consiste à générer un ensemble de taux d'utilisation et la seconde étape génère les périodes. La génération des dates d'échéance dépend du type de la tâche (échéance contrainte, implicite ou arbitraire) et pourra se baser sur la période tirée aléatoirement. Enfin, la durée d'exécution pourra être déduite de la période et du taux d'utilisation.

Les méthodes de génération de tâches disponibles dans SimSo sont présentées ci-après, en séparant la génération des taux d'utilisation de la génération des périodes. Il existe cependant d'autres générateurs permettant de générer des systèmes de tâches à criticité mixte, ou avec un meilleur contrôle du taux d'utilisation et des périodes des tâches [GBF02]. Ces autres générateurs ne sont actuellement pas directement intégrés dans SimSo, mais l'outil est capable de prendre en entrée des ensembles de tâches générés par des outils externes. Ceci est faisable soit en générant les fichiers XML par un outil externe, soit en utilisant SimSo depuis un script Python (voir partie 4.6).

4.4.1 Choix des taux d'utilisation

Les principaux algorithmes de génération de tâches sont proposés dans SimSo (UUniFast-Discard, RandFixedSum, méthode de Ripoll et méthode de Kato) et sont présentés ci-dessous. La méthode de Kato et l'algorithme RandFixedSum sont évalués dans le chapitre 5.

a) Méthodes de Ripoll et Kato

L'un des premiers générateurs de tâches pour systèmes multiprocesseurs a été proposé par Ripoll *et al.* [RCM96]. Kato a utilisé une approche très similaires afin de générer des ensembles de tâches pour ses travaux [KY07]. La méthode utilisée par Kato prend en entrée le taux d'utilisation total désiré et les bornes des taux d'utilisation des tâches. L'algorithme consiste alors à tirer aléatoirement, et uniformément [11] sur l'intervalle donné, des taux d'utilisation jusqu'à ce que l'utilisation totale soit atteinte. Le dernier taux d'utilisation généré est alors tronqué pour obtenir précisément l'utilisation totale désirée. Le nombre de tâches n'est pas une entrée de l'algorithme et varie d'un ensemble généré à un autre.

- Cette méthode a l'avantage d'être particulièrement simple à mettre en œuvre ce qui a sans doute participé à sa popularité. Il est en outre particulièrement aisé d'intervenir sur la distribution des taux d'utilisation des tâches.

11. Le tirage utilisé par Kato est uniforme, mais en pratique il est possible d'utiliser n'importe quelle distribution aléatoire.

- L'inconvénient est qu'il est très difficile de contrôler le nombre de tâches. En effet, le nombre de tâches dépend du tirage des taux d'utilisation et de l'utilisation totale désirée.

La figure 4.9 montre la signature de la méthode *gen_kato_utilizations* qui permet la génération de plusieurs ensembles de taux d'utilisation. La méthode de Ripoll ne génère pas que des taux d'utilisation mais génère aussi les périodes. Son fonctionnement n'est donc pas détaillé ici mais l'algorithme est disponible sous le nom *gen_ripoll*.

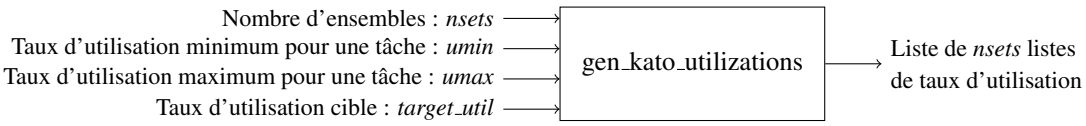


FIGURE 4.9 – Méthode pour générer des ensembles de tâches avec le générateur de Kato.

b) Algorithme UUniFast-Discard

L'algorithme UUniFast [BB05], conçu pour des architectures monoprocesseurs, permet la génération d'un ensemble de tâches avec comme paramètres d'entrée l'utilisation totale souhaitée et le nombre de tâches. L'extension de cet algorithme pour des taux d'utilisation supérieur à un, c'est-à-dire pour des systèmes multiprocesseurs, peut engendrer des tâches dont le taux d'utilisation dépasse un et qui ne sont donc pas ordonnançables. Pour éviter ce problème, Davis *et al.* ont proposé un nouvel algorithme, UUniFast-Discard, qui ignore simplement tous les ensembles de tâches incorrects [DB09].

- L'algorithme est relativement simple à mettre en œuvre et il permet de choisir indépendamment l'utilisation totale souhaitée et le nombre de tâches.
- Il est cependant très difficile de générer des ensembles corrects pour des systèmes très chargés avec peu de tâches.

La figure 4.10 montre la signature de la méthode *gen_uunifastdiscard* qui permet la génération de plusieurs ensembles de taux d'utilisation à l'aide de l'algorithme UUniFast-Discard.

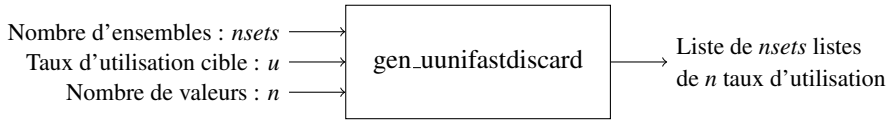


FIGURE 4.10 – Méthode pour générer des ensembles de tâches avec le générateur UUniFast-Discard.

c) Algorithme RandFixedSum

Enfin, Emberson *et al.* ont proposé d'utiliser l'algorithme RandFixedSum [12] qui permet de tirer aléatoirement un ensemble de N valeurs sur l'intervalle [0, 1] et dont la somme est égale à l'utilisation totale désirée [ESD10]. Cet algorithme donne des résultats de la même qualité que UUniFast-Discard, mais beaucoup plus rapidement.

12. L'algorithme a été initialement développé par Stafford pour Matlab avant d'être implémenté à nouveau en Python par Emberson. Le code embarqué dans SimSo se base sur la version Python développée par Emberson.

- Les avantages sont les mêmes que ceux de UUniFast-Discard, plus celui d'être beaucoup plus rapide pour générer des ensembles.
- Son implémentation est plus difficile. Cependant, il existe des implémentations libres pour Matlab et Python.

La figure 4.11 montre la signature de la méthode *gen_randfixedsum* qui permet la génération de plusieurs ensembles de taux d'utilisation à l'aide de l'algorithme RandFixedSum.

FIGURE 4.11 – Méthode pour générer des ensembles de tâches avec le générateur RandFixedSum.

4.4.2 Choix des périodes

Dans la littérature, les expérimentations présentées utilisent généralement un tirage des périodes selon une distribution uniforme sur un intervalle de valeurs. Davis a critiqué l'utilisation d'un tirage uniforme et préconise à la place d'utiliser un tirage selon une distribution log-uniforme [DB09]. En effet, sur un intervalle de 1 à 1000 ms, une distribution log-uniforme génère un nombre identique de périodes pour chaque intervalle (1 à 10 ms, 10 à 100 ms, 100 à 1000 ms) alors qu'une distribution uniforme génèrerait 90% des périodes dans l'intervalle 100 à 1000 ms.

Dans la pratique, les périodes ne sont cependant pas totalement aléatoires. Elles sont pour la plupart déterminées à partir de la dynamique des systèmes ou parfois fixées manuellement par des opérateurs humains qui ont préféré utiliser des valeurs entières et souvent multiples. Or, on sait par exemple que des périodes harmoniques permettent d'améliorer considérablement l'ordonnançabilité d'un système. C'est pour cette raison que SimSo propose entre autres un générateur de périodes qui utilise un ensemble discret de valeurs.

Le tableau 4.5 résume la liste des générateurs de périodes disponibles dans SimSo.

Distribution	Nom de la méthode
continue uniforme	gen_periods_uniform
continue log-uniforme	gen_periods_loguniform
discrète	gen_periods_discrete

TABLEAU 4.5 – Liste des générateurs de périodes disponibles dans SimSo.

4.4.3 Tâches sporadiques

SimSo est capable de simuler des tâches sporadiques. Contrairement à une tâche périodique, une tâche sporadique n'utilise pas la période mais une liste de dates d'activation qui sera calculée hors-ligne. Cette liste est spécifiée, pour chaque tâche, à travers l'attribut *list_activation_dates*.

Un générateur de dates d'arrivée est disponible et se base sur une loi de Poisson. Afin de déterminer la date de la prochaine arrivée, le générateur ajoute à la date de la dernière

arrivée la durée minimale d'inter-arrivée (P_i) puis effectue un tirage selon une loi de Poisson avec pour paramètre $\lambda = 1/P_i$. Et ainsi de suite, jusqu'à ce que la prochaine date d'arrivée dépasse la durée de la simulation. En moyenne il y a donc une arrivée toutes les $2 * P_i$ unités de temps.

4.5 Valeurs mesurées

La liste de tous les évènements liés à l'exécution des tâches sur les processeurs pendant la simulation du système est conservée sous la forme d'un ensemble de traces. Par exemple, chaque processeur contient la liste des travaux qu'il a exécutés (début d'exécution et interruption), et chaque tâche contient une trace des évènements tels que l'activation, le début d'exécution, l'interruption ou la fin d'exécution. Ces traces servent notamment de base pour afficher le diagramme de Gantt. L'accès à ces traces peut se faire de façon distincte ou agrégée, afin de compter certains évènements spécifiques.

Les métriques les plus usuelles peuvent être directement accédées à travers un ensemble de méthodes mises à disposition. Les principales métriques sont décrites ci-dessous et un résumé est proposé à la fin de cette partie.

4.5.1 Préemptions et migrations

Nous ferons la distinction entre une préemption, une migration de travail et une migration de tâche. La figure 4.12 illustre quatre types de situations relatives à l'exécution d'une tâche.

1. Préemption : un travail est interrompu et son exécution est reprise sur le même processeur.
2. Exécution d'une tâche sur le processeur sur lequel le dernier travail de cette tâche s'est exécuté.
3. Migration de travail : un travail est interrompu et son exécution reprend sur un processeur différent.
4. Migration de tâche : l'exécution de deux travaux successifs d'une tâche se fait sur des processeurs différents.

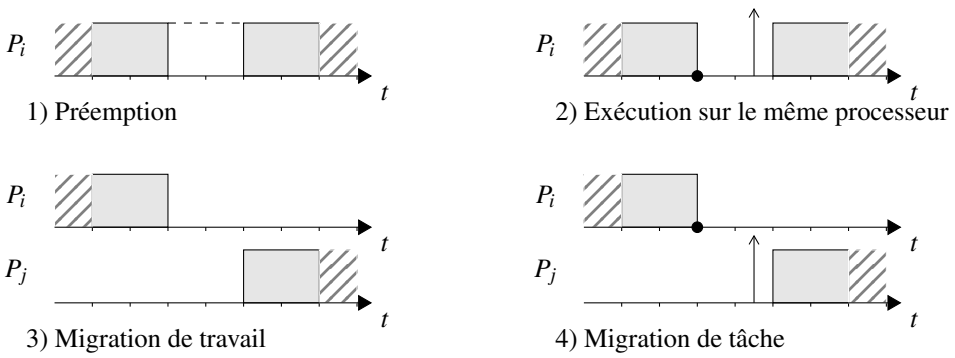


FIGURE 4.12 – Préemptions et Migrations de tâches

La figure 4.13 résume tous ces évènements sur un unique exemple. Dans cet exemple, quatre tâches (τ_1, τ_2, τ_3 et τ_4) s'exécutent sur deux processeurs (P_1 et P_2). Le j-ème travail de la tâche τ_i est noté $\tau_{i,j}$.

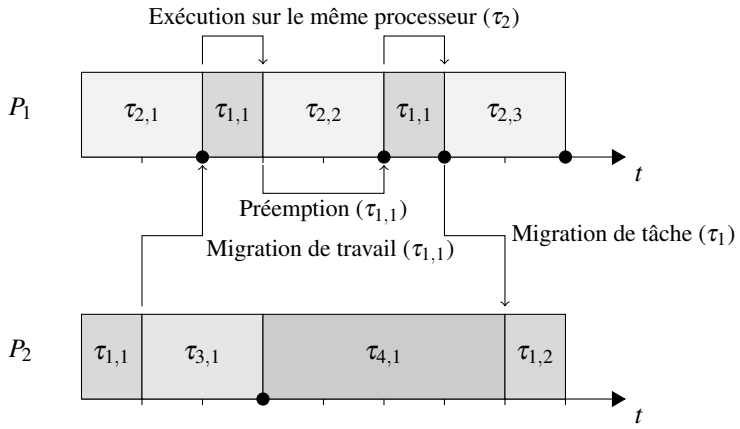


FIGURE 4.13 – Préemptions et migrations de tâches sur un exemple complet.

La migration de tâche est distinguée de la migration d'un travail car son impact sur les performances du système ne sont pas nécessairement les mêmes. En effet, deux instances d'une même tâche peuvent ne pas réutiliser les mêmes données (exemple : analyse d'une nouvelle image), ou bien, du fait de l'éloignement temporel des différentes activations, nous pourrions considérer qu'il est improbable que des données soient maintenues dans le cache . Dans ce cas, cette migration aurait assez peu d'influence sur le comportement du cache par rapport à une exécution sur le même processeur. Au contraire, une migration de travail est assimilable à une préemption (voir chapitre 2) et même en cas de reprise immédiate sur un autre cœur, le contenu des caches non partagés est potentiellement perdu. La réduction des migrations de travaux est donc prioritaire.

L'interruption d'un travail peut être provoquée par le système (exécution du code de l'ordonnanceur ou tout autre service de l'OS) ou être le résultat d'une prise de décision de l'ordonnanceur. Nous ferons alors la distinction entre les préemptions où un autre travail s'est exécuté sur le même processeur entre l'interruption du travail et sa reprise, des préemptions causées par le système.

4.5.2 Dépassements d'échéance

SimSo permet d'accéder pour chaque tâche au nombre de dépassements d'échéance (travaux avortés compris). Il est également possible d'accéder à la liste des travaux ayant dépassé leurs échéances pour des études ciblées. Un système sera alors par la suite considéré comme correctement ordonnancé sur l'intervalle de simulation s'il n'y a eu aucun dépassement d'échéance.

Au cours du chapitre suivant, nous nous intéresserons au taux de succès. Le taux de succès (relativement à la durée de simulation) correspond, pour un ensemble de systèmes simulés, au nombre de systèmes correctement ordonnancés divisé par le nombre de systèmes considérés. Cette valeur est calculée en dehors de SimSo en se basant sur le nombre de dépassements d'échéance. Notons qu'une telle valeur ne permet pas de faire la distinction entre un système qui a provoqué un seul dépassement d'échéance et un système ayant provoqué de multiples dépassements.

4.5.3 Temps de réponse et laxité normalisée

Pour chaque travail, le temps de réponse[13] est disponible. En soustrayant le temps de réponse à l'échéance relative d'un travail, nous obtenons sa laxité.

Afin de faciliter la manipulation de cette grandeur, nous avons retenu la définition de *normalized laxity*[14] proposée par Lelli *et al.* et qui consiste à diviser la laxité par la durée de la période [LFCL12]. L'équation 4.1 formalise la laxité normalisée.

$$Lnorm_{i,j} = \frac{D_i - R_{i,j}}{T_i} \tag{4.1}$$

4.5.4 Appels à l'ordonnanceur

Le nombre d'appels aux méthodes de l'ordonnanceur (*schedule*, *on_activate* et *on_terminate*) est compté. De plus, le coût total associé à chaque méthode est calculé. Attention, dans le cas de blocages (voir partie 4.3.1 concernant le système de verrous), le coût d'appel à une méthode peut dépasser son coût unitaire. Ainsi, le coût total peut dépasser la somme des pénalités multipliées par leur nombre.

Le nombre de déclenchements de timer est également comptabilisé.

4.5.5 Récapitulatif des mesures disponibles

Toutes les informations utiles sont regroupées dans l'objet *Results* qui est généré à la fin de la simulation. Cet objet contient des objets de type *TaskR*, *SchedulerR*, *ProcessorR* et indirectement des *JobR*. Ces classes permettent de contenir des métriques plus spécifiques.

Le tableau 4.6 contient la liste des principales métriques disponibles pour chaque travail. Certaines de ces métriques sont additionnées au niveau de la tâche comme le montre le tableau 4.7 et au niveau du système (tableau 4.8). L'objet *SchedulerR* contient les évènements concernant les appels aux méthodes de l'ordonnanceur (tableau 4.9).

Attribut	Description
preemption_count	Nombre de préemptions
preemption_inter_count	Nombre de préemptions avec exécution d'une autre tâche[15]
migration_count	Nombre de migrations de travail
activation_date	Date d'activation
computation_time	Durée d'exécution
end_date	Date de fin d'exécution
response_time	Temps de réponse
normalized_laxity	Laxité normalisée

Tableau 4.6 – Données disponible dans la classe *JobR*.

13. La définition du temps de réponse est donnée dans le chapitre 1 dans la partie e).
14. D'autres définitions de *normalized laxity* existent [DNS94].
15. Nous compterons séparément les préemptions pour lesquelles une tâche s'est exécutée entre l'interruption et la reprise d'exécution d'un travail.

Attribut	Description
resumption_count	Nombre d'exécutions d'un travail sur le même processeur
task_migration_count	Nombre de migrations de tâche
migration_count	Nombre de migrations de travail
preemption_count	Nombre de préemptions de travail
preemption_inter_count	Nombre de préemptions avec exécution d'une autre tâche
exceeded_count	Nombre de travaux ayant dépassé leur échéance

Tableau 4.7 – Données disponibles dans la classe *TaskR*.

Attribut	Description
total_preemptions	Nombre total de préemptions
total_migrations	Nombre total de migrations de travail
total_task_migrations	Nombre total de migrations de tâche
total_task_resumptions	Nombre total d'exécutions sur le même processeur
total_exceeded_count	Nombre total de dépassements d'échéance
total_timers	Nombre total de déclenchements de timer

Tableau 4.8 – Données disponibles dans la classe *Results*.

Attribut	Description
schedule_overhead	Surcoût total pour les appels à la méthode *schedule*.
activate_overhead	Surcoût total pour les appels à la méthode *on_activate*
terminate_overhead	Surcoût total pour les appels à la méthode *on_terminated*
schedule_count	Nombre d'appels à la méthode *schedule*
activate_count	Nombre d'appels à la méthode *on_activate*
terminate_count	Nombre d'appels à la méthode *on_terminated*

Tableau 4.9 – Données disponibles dans la classe *SchedulerR*.

4.6 Utilisation de Simso

SimSo dispose d'une interface homme-machine graphique qui offre une prise en main conviviale et rapide de l'outil. Cette interface permet de configurer un système, de le simuler et de visualiser les résultats (voir partie 4.6.1). Il est en outre possible de représenter l'exécution d'un système sous la forme d'un diagramme de Gantt ou d'obtenir des statistiques sur les métriques les plus courantes. L'utilisation de cette interface est particulièrement utile lors de la phase de développement d'une politique d'ordonnancement ou à des fins pédagogiques.

Cependant, cette interface graphique ne permet pas d'automatiser le lancement de multiples simulations. Il serait possible d'adapter l'interface dans ce but, mais un autre choix a été fait. En effet, SimSo se présente également sous la forme d'un module Python qu'il est possible d'utiliser à partir d'un script (voir partie 4.6.2). Comme nous le verrons, l'utilisation de SimSo en tant que module Python permet un maximum de flexibilité.

4.6.1 Interface graphique

L'interface graphique de SimSo permet d'étudier une politique d'ordonnancement en offrant la possibilité de configurer un système et d'analyser les données issues de la simulation. La figure 4.14 donne une vue d'ensemble de l'IHM de SimSo.

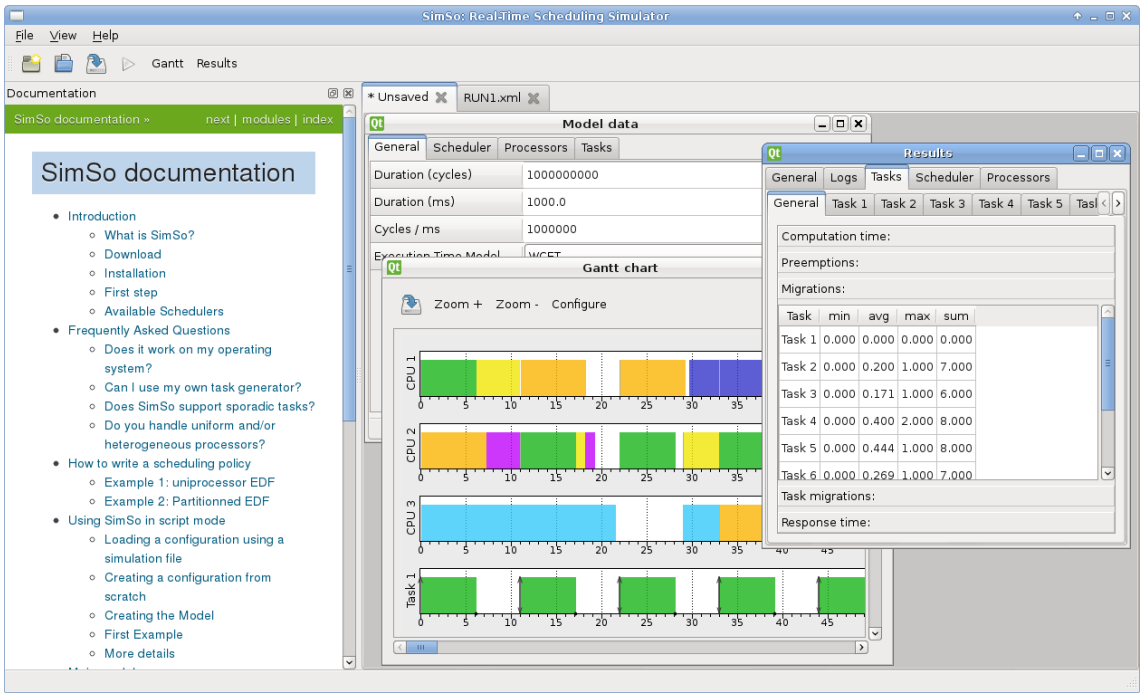


FIGURE 4.14 – Interface graphique de SimSo.

L'ensemble des paramètres généraux de la simulation peut être configuré graphiquement (voir figure 4.15) et ces paramètres peuvent être enregistrés ou chargés depuis un fichier XML. L'utilisateur peut choisir l'ordonnanceur à utiliser (voir figure 4.16a) et configurer les surcoûts à appliquer lors de l'appel à certains services. La déclaration des processeurs se fait naturellement (voir figure 4.16b) et il est possible de configurer la durée d'une sauvegarde de contexte (CS), la durée d'un rechargement de contexte (CL) et la vitesse initiale du processeur.

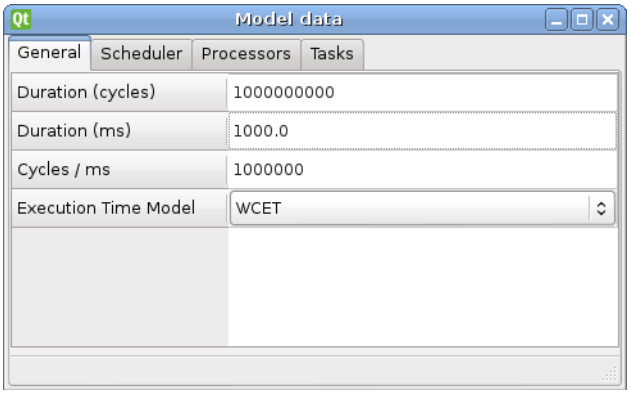
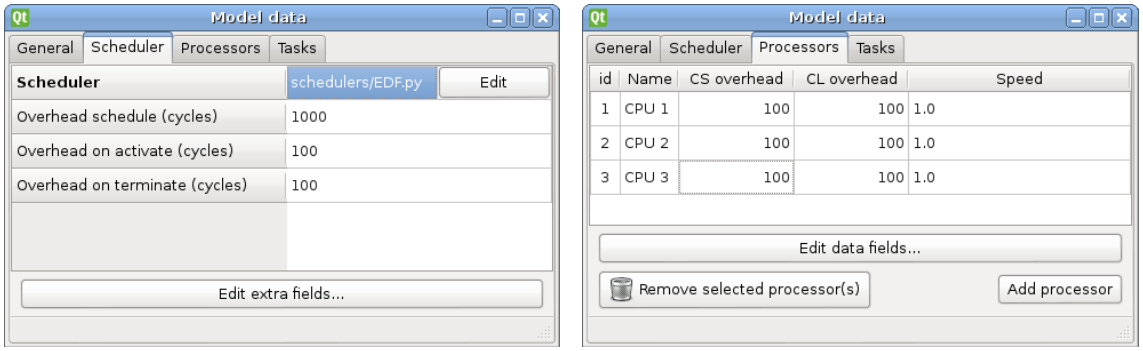


FIGURE 4.15 – Fenêtre de configuration.

La déclaration des tâches peut se faire manuellement comme pour les processeurs (voir figure 4.17) mais il est aussi possible d'utiliser un générateur de tâches. La figure 4.18 montre la fenêtre de dialogue qui permet de choisir l'algorithme de génération, l'utilisation



(a) Fenêtre de configuration de l'ordonnanceur. (b) Fenêtre de configuration des processeurs.

FIGURE 4.16 – Configuration de l'ordonnanceur et des processeurs.

totale, le choix des périodes mais aussi le nombre de tâches périodiques ou sporadiques dans le cas de l'utilisation du générateur RandFixedSum (le nombre de tâches n'étant pas une entrée du générateur de Kato).

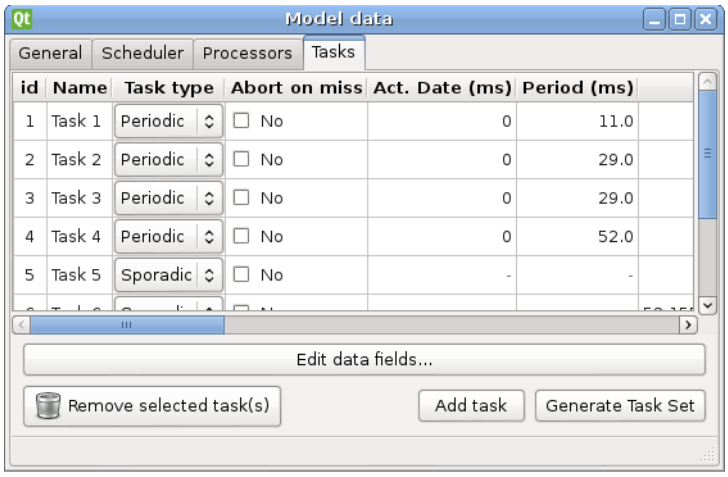


FIGURE 4.17 – Fenêtre de configuration des tâches.

Un diagramme de Gantt permet de visualiser l'exécution du système de manière simple (voir figure 4.19). Ce diagramme offre un moyen plus intuitif pour comprendre le comportement d'une politique. Cela facilite le débogage d'un algorithme en identifiant visuellement les comportements erronés. Pour compléter, un ensemble de données statistiques concernant des métriques usuelles est disponible (voir figure 4.20).

4.6.2 Mode script

L'utilisation de SimSo à partir de scripts Python permet d'automatiser la génération de systèmes, leur simulation et la collecte des grandeurs à étudier, en vue d'une évaluation à grande échelle. Ce mode d'utilisation apporte en particulier une flexibilité supplémentaire, que ce soit sur la génération des systèmes ou sur les métriques recueillies. Dans les faits, l'interface graphique de SimSo n'est rien d'autre qu'une application Python utilisant le module SimSo et il serait tout à fait possible de créer une nouvelle interface dédiée à un autre besoin ciblé.

Les principales phases permettant la conduite d'une simulation dans ce mode sont maintenant décrites. Pour plus de détails, le lecteur est invité à consulter la documentation

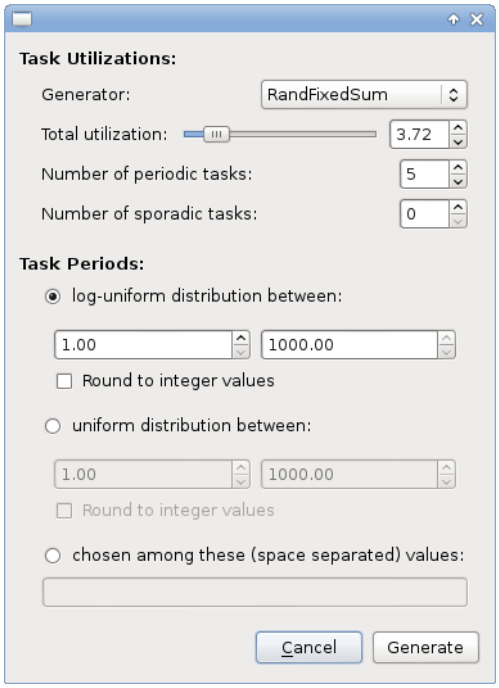


FIGURE 4.18 – Interface graphique pour la génération de tâches.

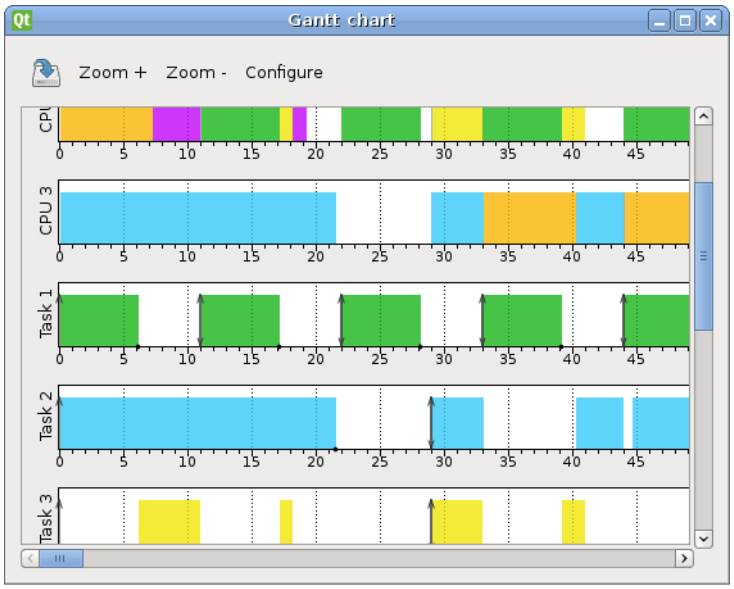


FIGURE 4.19 – Diagramme de Gantt généré par SimSo.

fournie avec le logiciel. Cette section se termine par une description de la mise en œuvre des expérimentations qui sont présentées au chapitre suivant.

a) Création d'un système

La première étape consiste à configurer le système à simuler. Pour ce faire, il est nécessaire d'instancier un objet de la classe *Configuration* (voir partie 4.2.2). Le constructeur de cette classe peut éventuellement recevoir en argument le chemin d'un fichier XML contenant une configuration existante.

Dans l'exemple suivant, une configuration vierge est créée puis le modèle de temps d'exé-

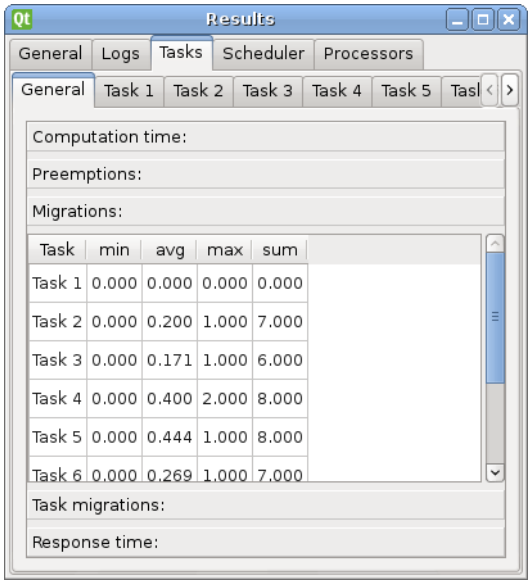


FIGURE 4.20 – Fenêtre présentant les métriques les plus usuelles.

cution est précisé, ainsi que la durée de simulation. Les attributs non configurés reçoivent une valeur par défaut.

```
from simso.configuration import Configuration

# Création nouvelle configuration.
configuration = Configuration()

# Utilisation du WCET pour les durées d'exécution des travaux.
configuration.etm = "wcet"

# Simulation sur une durée de 1000ms.
configuration.duration = 1000 * configuration.cycles_per_ms
```

b) Choix de l'ordonnanceur

Pour rappel, un ordonnanceur est une classe Python extérieure au simulateur. Il suffit alors d'indiquer le chemin vers le fichier Python qui contient cette classe :

```
configuration.scheduler_info.set_name("schedulers/RM.py")
```

Si nécessaire, il est possible de passer des arguments supplémentaires à l'ordonnanceur et qui seront disponibles dans un dictionnaire (voir partie c)) :

```
configuration.scheduler_info.set_fields({'var': (42, 'int')})
```

c) Création des processeurs

L'objet *configuration* possède une liste de *ProcInfo* et il est possible d'ajouter directement à cette liste un nouvel élément. Mais pour simplifier le travail de l'utilisateur, il est aussi possible d'utiliser la méthode *add_processor* afin d'ajouter un nouveau processeur. Les attributs non spécifiés recevront alors une valeur par défaut.

```
configuration.add_processor(name="CPU 1", identifier=1)
configuration.add_processor(name="CPU 2", identifier=2)
configuration.add_processor(name="CPU 3", identifier=3)
```

d) Création des tâches

De manière analogue, il est possible d'ajouter des tâches directement dans la liste de *TaskInfo* ou d'utiliser la méthode *add_task*. Là encore, les attributs non spécifiés recevront une valeur par défaut. En l'absence du type de tâche, la tâche sera considérée comme périodique.

```
configuration.add_task(name="T1", identifier=1, period=7, activation_date=0,
                       wcet=3, deadline=7)
configuration.add_task(name="T2", identifier=2, period=12, activation_date=0,
                       wcet=3, deadline=12)
configuration.add_task(name="T3", identifier=3, period=20, activation_date=0,
                       wcet=5, deadline=20)
```

SimSo propose quelques méthodes pour générer automatiquement des ensembles de tâches (voir section 4.4). À titre d'exemple, la figure 4.21 montre comment générer un ensemble de tâches en utilisant l'algorithme RandFixedSum avec une génération de périodes selon une distribution log-uniforme :

```
import simso.generator.task_generator as task_generator

n = 10  # Nombre de tâches
u = 3.5  # Utilisation totale
nsets = 1  # Nombre d'ensembles de tâches
period_min = 2
period_max = 100

utilizations = task_generator.StaffordRandFixedSum(n, u, nsets)
periods = task_generator.gen_periods_loguniform(
    n, nsets, period_min, period_max, round_to_int=True)

exp_set = task_generator.gen_tasksets(utilizations, periods)[0]

# Ajout des tâches au système :
id_ = 1
for (c, p) in exp_set:
    configuration.add_task(
        name="T{}".format(id_), identifier=id_, period=p, activation_date=0,
        wcet=c, deadline=p, abort_on_miss=True)
    id_ += 1
```

FIGURE 4.21 – Code permettant la génération d'un ensemble de tâches par l'algorithme RandFixedSum avec distribution des périodes log-uniforme.

e) Simulation

Une fois la configuration ainsi spécifiée, la première étape est de vérifier la cohérence de la configuration en appelant la méthode *check_all*. Cette étape lève une exception en cas d'erreur. Les erreurs peuvent être par exemple : périodes négatives, aucun processeur, échec au chargement de l'ordonnanceur, etc. Cette étape n'est cependant pas obligatoire

mais sert à éviter des simulations incorrectes. Ensuite, une instance de la classe *Model* doit être créée à partir de la configuration. Enfin, la méthode *run_model* peut être appelée pour lancer la simulation (voir figure 4.22).

```
# Vérification de la config.
configuration.check_all()

# Initialisation de la simu à partir de la config.
model = Model(configuration)
# Execution de la simu.
model.run_model()
```

FIGURE 4.22 – Exécution de la simulation depuis un script.

f) Lecture des résultats

Lorsque la simulation se termine, et si l'ordonnanceur n'a pas pris de décision erronée au cours de la simulation [16], alors les résultats sont disponibles dans l'attribut *results* de l'objet *Model*. Cet objet a été instancié à partir de la classe *Results* à la fin de la simulation et il facilite l'extraction de données usuelles (voir partie 4.5). Dans le cas où une donnée ne serait pas directement disponible, il est possible d'obtenir la trace de tous les évènements liés à l'exécution des tâches et aux appels à l'ordonnanceur qui se sont produits au cours de la simulation (voir partie 4.5).

g) Automatisation

Les expérimentations réalisées dans le chapitre suivant ont été automatisées à travers un script Python qui utilise SimSo en tant que module. Ce script a généré les configurations, les a simulées puis en a extrait des données pour les stocker dans une base de données. Un second script a permis le traitement statistique des données et la création des courbes. Notons également que le traitement des données aurait pu être réalisé dans un autre langage sans difficulté.

La génération des systèmes a été faite à l'aide des méthodes fournies par SimSo et qui viennent d'être présentées (algorithme RandFixedSum et distribution des périodes log-uniforme). Chaque configuration fût sauvegardée dans un fichier XML afin de pouvoir éventuellement la rejouer ou la simuler avec une nouvelle politique.

Afin de profiter au mieux des ressources disponibles, plusieurs simulations peuvent s'exécuter en parallèle à l'aide d'un pool de threads. Lorsqu'une simulation se termine, les données issues de la simulation (voir partie 4.5) sont stockées dans une base de donnée SQLite3. Il est possible à tout moment d'interrompre les expérimentations et le script reprendra depuis son point d'interruption en consultant la présence des résultats disponibles dans la base de données. L'algorithme d'un tel script est synthétisé par la figure 4.23.

16. Exemple d'événements provoquant une interruption de la simulation : exécution d'un travail inexistant, d'un travail terminé ou exécution d'un travail sur deux processeurs en même temps.

```
- Génération des configurations et sauvegarde dans fichiers XML.
- Pour chaque fichier XML (calcul distribué dans un pool de threads) :
    - Si résultats non présents dans base SQL :
        - Simulation à partir du XML
        - Extraction des résultats
        - Sauvegarde dans base SQL
```

FIGURE 4.23 – Pseudo-code correspondant au premier script.

Lorsque tous les résultats ont été obtenus et stockés dans la base de données, un second script permet l'exploitation des résultats. Les courbes ont été générées à l'aide de *matplotlib*, une bibliothèque pour Python permettant de générer des graphiques.

4.7 Conclusion

Ce chapitre a présenté SimSo, un simulateur d'ordonnancement temps réel pour architecture multiprocesseur. Ce simulateur a été conçu de manière modulaire afin de pouvoir être étendu facilement à de nouveaux modèles de tâches et d'architectures. Il est à souligner qu'une attention toute particulière a été portée au contrôle de la durée d'exécution simulée des travaux. Au cours du chapitre 7, l'intégration de modèles de cache dans SimSo sera détaillé.

Son interface graphique permet de configurer un système, de le simuler et d'afficher les résultats sous la forme d'un diagramme de Gantt et d'une liste de métriques calculées à partir de la trace de l'exécution simulée du système. Mais cet outil a été développé avant tout dans l'optique de pouvoir mettre en œuvre des évaluations à grande échelle. C'est pour cela que SimSo est également utilisable sous la forme d'une bibliothèque. Ceci permet de programmer la génération des systèmes et d'automatiser les simulations et l'extraction des résultats. Le chapitre 5 présente des résultats obtenus dans de telles conditions à l'aide de SimSo.

Enfin, l'interface de programmation pour la mise en œuvre d'un ordonnanceur a été décrite. Cette interface a été pensée pour être réaliste en se basant sur des éléments disponibles sur un vrai système, mais tout en essayant de rester simple. Plus de vingt-cinq algorithmes d'ordonnancement ont été mis en œuvre pour SimSo, montrant que l'interface est assez souple pour permettre l'implémentation d'algorithmes aux approches variées (global, partitionné, semi-partitionné, etc).

À l'heure actuelle, SimSo commence à être employé en dehors du cadre de cette thèse [BFO14]. La facilité de prise en main de l'outil, son aspect modulaire lui permettant de s'adapter à de nouveaux besoins et la bibliothèque d'ordonnanceurs disponibles sont certainement les aspects qui ont été les plus appréciés d'après les échanges avec des chercheurs. SimSo a aussi été utilisé à l'INSA de Toulouse dans la cadre d'enseignements sur les systèmes temps réel. L'interface graphique et la facilité pour paramétrer les systèmes de tâches permet aux étudiants de rapidement comprendre le comportement d'une politique d'ordonnancement et l'influence des différents paramètres.

CHAPITRE 5

Évaluation des politiques d'ordonnancement

Ce chapitre présente un ensemble d'évaluations des principaux algorithmes d'ordonnancement introduits au cours du premier chapitre et mis en œuvre dans SimSo. À cette fin, les politiques d'ordonnancement sont simulées à l'aide de SimSo sur un ensemble varié de configurations. Au total, une vingtaine de politiques sont simulées sur plus de 10 000 systèmes différents. Toutes les politiques sont ainsi évaluées et comparées dans des conditions identiques (mêmes jeux d'entrée et même outil).

L'objectif premier est de mieux comprendre le comportement des politiques et les paramètres qui ont une influence sur leurs performances en matière d'ordonnançabilité, préemptions, migrations et temps de réponse. À travers ces expérimentations, des résultats parfois connus sont précisés et de nouveaux sont présentés. Ce chapitre est aussi l'occasion de montrer que SimSo permet d'évaluer facilement des algorithmes d'ordonnancement.

Au préalable, nous présentons et justifions nos choix concernant les paramètres des systèmes qui sont générés et les métriques qui sont recueillies depuis les simulations. Ensuite, nous étudions les politiques aux approches globales, partitionnés, DP-Fair ou hybrides et nous analysons aussi l'usage du WCET et les bénéfices apportés par les politiques *work-conserving*.

5.1 Mise en place de l'évaluation

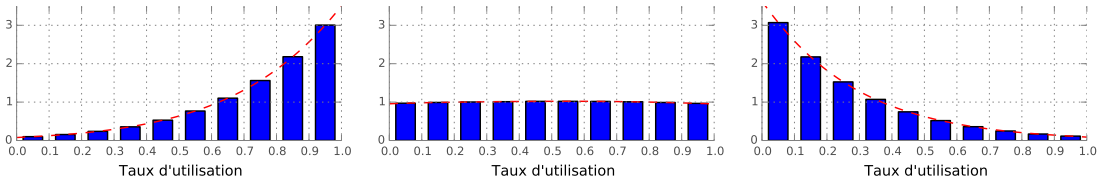
Les diverses évaluations proposées au cours de ce chapitre ont été réalisées dans des conditions identiques pour l'ensemble des algorithmes d'ordonnancement. En effet, nous avons vu dans le chapitre 3 qu'il est difficile d'avoir une vision unifiée des comportements des algorithmes d'ordonnancement car les évaluations sont réalisées pour des jeux de test souvent mal définis et toujours différents. Pour répondre à ce problème, les systèmes (ensembles de tâches et processeurs) que nous étudions sont identiques pour l'ensemble des politiques d'ordonnancement évaluées. De plus, après chaque expérimentation, un ensemble de métriques ont été extraites des simulations et stockées dans une base de données. Le principe de mise en œuvre d'expérimentations à grande échelle et de manière automatisée a été présenté au cours du chapitre précédent dans la partie g).

Dans cette première partie, nous justifions le choix des algorithmes pour la génération des tâches en les évaluant (section 5.1.1). Puis nous présentons les caractéristiques des systèmes générés (section 5.1.2). Enfin, les métriques qui ont été collectées sont présentées dans la section 5.1.3.

5.1.1 Comparaison des méthodes de génération

Dans la partie 4.4, nous avons vu les deux principales approches pour la génération des taux d'utilisation. La première, employée par Kato, effectue des tirages de taux d'utilisation pour les tâches, en nombre indéterminé a priori, jusqu'à ce que la somme atteigne la valeur fixée pour le taux d'utilisation total. La seconde approche (employée par les algorithmes RandFixedSum et UUniFast-Discard) génère un ensemble de N taux d'utilisation dont la somme correspond à la valeur demandée.

Il est important de comprendre l'impact des algorithmes sur le profil des tâches générées. En effet, l'approche de Kato permet de contrôler la distribution des taux d'utilisation mais le nombre de tâches est variable, tandis que l'approche au cœur de RandFixedSum et UUniFast-Discard consiste à adapter les taux d'utilisation pour atteindre l'objectif avec un nombre fixe de tâches. La figure 5.1 montre la densité de probabilité des taux d'utilisation des tâches pour un nombre variable de tâches et un même taux d'utilisation total tel qu'obtenu par l'algorithme RandFixedSum.



(a) Avec 16 tâches. (b) Avec 24 tâches. (c) Avec 48 tâches.

FIGURE 5.1 – Densité de probabilité des taux d'utilisation des tâches pour une utilisation totale de 12 avec l'algorithme RandFixedSum. Valeurs obtenues à partir de 10 000 ensembles de tâches.

Nous pouvons voir qu'à taux d'utilisation total constant, s'il y a peu de tâches alors les tâches seront majoritairement lourdes et au contraire, s'il y a beaucoup de tâches, elles seront majoritairement légères. L'algorithme RandFixedSum a cependant l'avantage de permettre un choix du nombre de tâches contrairement à l'approche employée par Kato *et al.*

En effet, dans le cas du générateur de Kato, si l'espérance de la loi de probabilité utilisée pour le tirage est de 0.5, alors le nombre de tâches sera en moyenne proche de deux fois l'utilisation totale. En pratique on obtient un nombre supérieur de tâches car une tâche supplémentaire est ajoutée pour obtenir précisément le taux d'utilisation total souhaité. La figure 5.2 montre le nombre de tâches obtenues en moyenne pour 10 000 ensembles de tâches en faisant varier l'utilisation totale cible et pour un tirage des taux d'utilisation uniforme sur [0, 1].

L'ajout d'une tâche supplémentaire pour compléter le système ajoute un biais dans la génération des taux d'utilisation. En effet, la dernière valeur tirée au hasard est tronquée pour obtenir précisément la somme voulue. Ceci signifie qu'on ajoute au système une valeur inférieure à ce qui était prévu. Cependant, cela ne concerne qu'une seule tâche sur tout

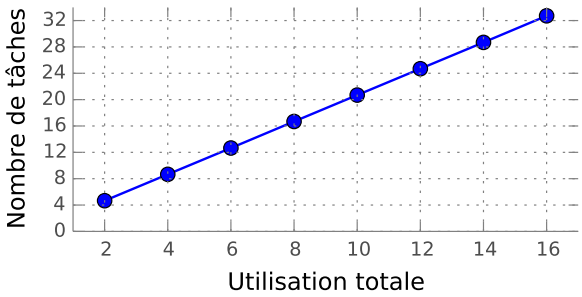
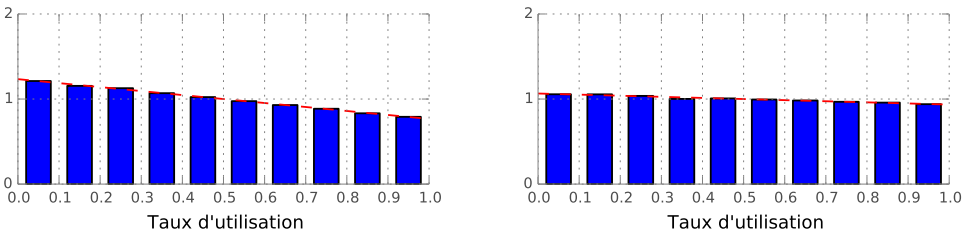


FIGURE 5.2 – Nombre moyen de tâches en fonction de l'utilisation totale avec le générateur de Kato.

le système et donc l'effet reste limité si le nombre de tâches est suffisamment grand. La figure 5.3 montre la densité de probabilité des taux d'utilisation obtenue sur une moyenne de 10 000 ensembles de tâches en faisant varier le taux d'utilisation total.



(a) Avec une utilisation totale de 4. (b) Avec une utilisation totale de 16.

FIGURE 5.3 – Densité de probabilité des taux d'utilisation obtenue à partir de 10 000 ensembles de tâches avec l'algorithme de Kato.

5.1.2 Génération des systèmes et simulations

Suite à l'étude sur les générateurs de taux d'utilisation qui vient d'être présentée, nous avons opté pour l'algorithme RandFixedSum en raison de la possibilité de contrôler le nombre de tâches. En effet, nous souhaiterions connaître le comportement de certaines politiques en faisant varier ce paramètre. La génération des périodes des tâches utilise une distribution log-uniforme des périodes sur l'intervalle 2 à 100 ms. La distribution log-uniforme a été choisie car nous avons été sensible aux arguments de Davis *et al.* qui ont été reproduits dans la partie 4.4.

Les tâches sont indépendantes, à échéances implicites et synchrones. Leur nombre est choisi dans l'ensemble {20, 30, 40, 50, 60, 70, 80, 90, 100}. Le nombre de processeurs est compris dans l'ensemble {2, 4, 8, 12, 16} et les surcoûts liés aux préemptions et aux appels à l'ordonnanceur sont tous fixés à zéro. Nous avons décidé de ne pas étudier les algorithmes sur des systèmes comportant plus de processeurs car la plupart d'entre eux n'ont pas été conçus dans cette optique et qu'il convient généralement de les coupler à des techniques de clustering par exemple. Le taux d'utilisation total est exprimé ici en pourcentage par rapport au nombre de processeurs (u_{sum}/m) et compris dans l'ensemble {70, 75, 80, 85, 90, 95, 97.5, 100}.

Pour chaque triplet (nombre de tâches, nombre de processeurs, taux d'utilisation), 20 systèmes ont été générés aléatoirement et ces systèmes ont été simulés en utilisant le

WCET comme modèle de temps d'exécution puis le modèle ACET [1]. Au total, c'est donc 14 400 systèmes qui ont été simulés par une vingtaine de politiques d'ordonnancement. De plus, certaines configurations supplémentaires ont également été simulées avec un ensemble réduit d'ordonnanceurs pour les besoins de certaines expérimentations.

Les systèmes ont été simulés sur une durée de 1000 ms. Compte tenu de l'utilisation de périodes aléatoires et d'un nombre de tâches important, il n'est pas possible de simuler sur l'hyper-période car celle-ci est trop grande, dans la quasi-totalité des cas. Cette durée, qui correspond à cinq fois la période maximale d'une tâche, semble suffisante pour obtenir des informations telles que le nombre de préemptions ou les temps de réponse. Une partie des évaluations ont été reproduites sur une durée de 2000 ms et l'écart relatif concernant le nombre de préemptions par seconde est en moyenne inférieur à 1% et ne dépasse pas 4%. La figure 5.4 illustre le nombre de préemptions observées en fonction de la durée de simulation. Nous ne pouvons cependant pas statuer avec exactitude sur l'ordonnançabilité des systèmes étudiés.

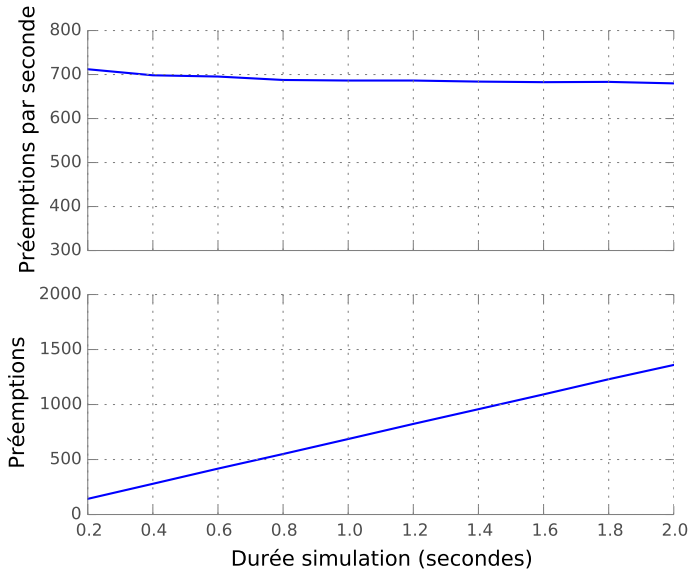


FIGURE 5.4 – Nombre de préemptions pour G-EDF en fonction de la durée de la simulation. Le système est composé de 50 tâches, 4 processeurs et le taux d'utilisation du système est de 97.5%.

En combinant toutes les configurations générées aux ordonnanceurs étudiés, ce sont au total plus de 300 000 simulations qui ont été exécutées par SimSo. Toutes les simulations n'ont pas été faites à la suite et des corrections ont parfois été apportées aux implémentations ce qui a nécessité de relancer une partie des simulations. Par conséquent, il est difficile de donner le temps de simulation total. Nous estimons cependant ce temps à environ une semaine sur une machine multiprocesseur dotée d'un total de 16 cœurs à 3GHz.

Le tableau 5.1 résume l'ensemble des paramètres importants pour les simulations.

1. Dans le cas de l'utilisation du modèle ACET, la durée moyenne est fixée à 75% du WCET et l'écart type à 10% du WCET.

2. Les travaux déjà en cours d'exécution continuent à s'exécuter sur le même processeur. Les autres travaux s'exécutent sur les processeurs restants.

Paramètre	Valeur
Durée de la simulation	1000 ms
Précision	10^{-9} s
Nombre de tâches (n)	{20, 30, 40, 50, 60, 70, 80, 90, 100}
Nombre de processeurs (m)	{2, 4, 8, 12, 16}
Utilisation totale relative (u_{sum}/m)	{70, 75, 80, 85, 90, 95, 97.5, 100}
Systèmes générés pour chaque configuration	20
Modèle de temps d'exécution	WCET et ACET
Comportement en cas de dépassement d'échéance	Avortement du travail
Ordonnanceurs	≈20 politiques, voir tableau 5.2
Algorithme de génération des u_i	RandFixedSum
Distribution des périodes	log-uniforme sur [2, 100] ms
Pénalité temporelle	Aucune

Tableau 5.1 – Résumé des paramètres importants.

Ordonnanceur	Commentaire
G-EDF	-
PriD	-
EDF-US	Paramètre $\xi = 1/2$
G-FL	-
EDZL	-
G-MLLF	-
Decreasing First-Fit P-EDF	-
Decreasing Worst-Fit P-EDF	-
LLREF	Affectation des tâches aux processeurs simple [2]
LRE-TL	-
NVNLF	-
DP-WRAP	-
RUN	-
EKG	Paramètre K égal au nombre de processeurs
U-EDF	-
PD2	Quantum de 0.1ms
ER-PD2	Quantum de 0.1ms
WC-RUN	Couplage de G-EDF à RUN
WC-U-EDF	Couplage de G-EDF à U-EDF

Tableau 5.2 – Ordonnanceurs évalués.

5.1.3 Valeurs mesurées

À la fin de chaque simulation, un certain nombre de métriques sont calculées puis enregistrées dans une base de données. Chaque simulation est identifiée par : (tâches, processeurs, taux d'utilisation relatif, ordonnanceur, identifiant de l'expérimentation allant de 0 à 19, utilise le WCET ou ACET).

Les métriques que nous recueillons après chaque simulation [3] et que nous utilisons dans ce chapitre sont :

3. Les métriques les plus usuelles offertes par SimSo ont été listées dans les tableaux 4.6, 4.7 et 4.8.

- Nombre de travaux
- Nombre de dépassements d'échéance (= avortements de travaux)
- Nombre de préemptions (avec exécution d'un autre travail avant la reprise)
- Nombre de migrations
- Laxité normalisée

Bien que cela ne soit pas utilisé dans ce chapitre, nous recueillons également :

- Nombre de migrations de tâche
- Nombre de déclenchements de timer
- Nombre d'appels à la méthode *on_schedule*
- Nombre de préemptions sans exécution d'un autre travail avant la reprise
- Message d'erreur en cas de problème (exemples : échec du partitionnement, ordonnancement d'un travail terminé ou exécution sur plusieurs processeurs)

Afin de tracer les figures correspondant aux résultats, des moyennes sont réalisées. En l'absence de précision, la moyenne est réalisée sur l'ensemble des domaines de définition des paramètres.

5.2 Évaluation des politiques de type G-EDF

5.2.1 Test d'ordonnançabilité GFB

Les tests d'ordonnançabilité pour G-EDF n'offrent pas, pour la plupart (les moins complexes), des conditions exactes d'ordonnançabilité. En conséquence, ces tests ne permettent pas de conclure sur l'ordonnançabilité pour un grand nombre de systèmes qui ne satisfont pas les conditions suffisantes mais satisfont cependant les conditions nécessaires. En outre, les résultats théoriques concernant G-EDF montrent des taux d'utilisation faibles dans certains cas (voir partie 1.3).

Pourtant, il semblerait que G-EDF donne en pratique de bons résultats. Lelli *et al.* ont par exemple montré que durant leurs expérimentations avec G-EDF, la plupart des systèmes n'ont pas provoqué de dépassement d'échéance alors qu'aucun système n'était théoriquement ordonnançable d'après les tests d'ordonnançabilité [LFCL12]. Bien qu'une preuve de l'ordonnançabilité soit la bienvenue pour des systèmes temps réel durs, quelques dépassements d'échéance peuvent être tolérés pour des systèmes temps réel souples. Ainsi, dans certains cas peu critiques, une preuve de l'ordonnançabilité n'est pas strictement nécessaire et le principal est que « cela se passe bien en pratique ».

Nous nous intéressons ici à la comparaison du taux de succès de G-EDF par simulation [4], au taux de systèmes ordonnançables d'après le test GFB (du nom des auteurs Goossens, Funk et Baruah) [GFB03]. Ce test permet de dire qu'un ensemble de tâches périodiques est ordonnançable par G-EDF si $u_{sum} \leq m \times (1 - u_{max}) + u_{max}$, mais ne permet pas de conclure si ce test n'est pas satisfait.

4. Nous rappelons que la durée de simulation est insuffisante pour être certain qu'un dépassement d'échéance n'aura pas lieu ultérieurement.

Résultats

L'ensemble des systèmes générés et simulés à l'aide de G-EDF a été testé à l'aide du test GFB. La figure 5.5 montre le taux de succès de G-EDF par la simulation et selon le critère GFB. Chaque point correspond au nombre de systèmes correctement ordonnancés sur un total de 900 systèmes (20 systèmes avec variation du nombre de tâches et de processeurs).

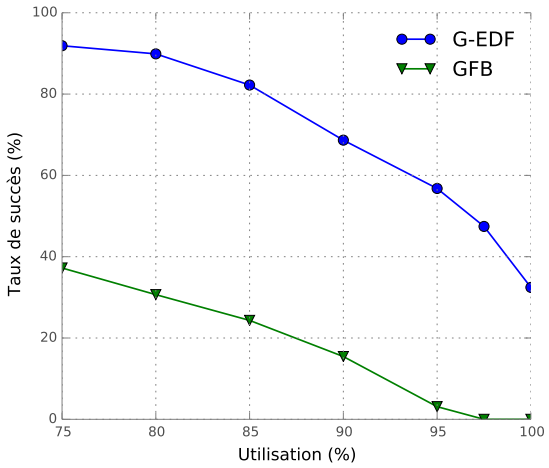


FIGURE 5.5 – Taux de succès de la politique G-EDF par simulation comparé au taux de systèmes ordonnançables selon le critère GFB en fonction du taux d'utilisation du système. Le nombre de tâches est compris dans {20, 30, 40, 50, 60, 70, 80, 90, 100} et le nombre de processeurs est compris dans {2, 4, 8, 12, 16}.

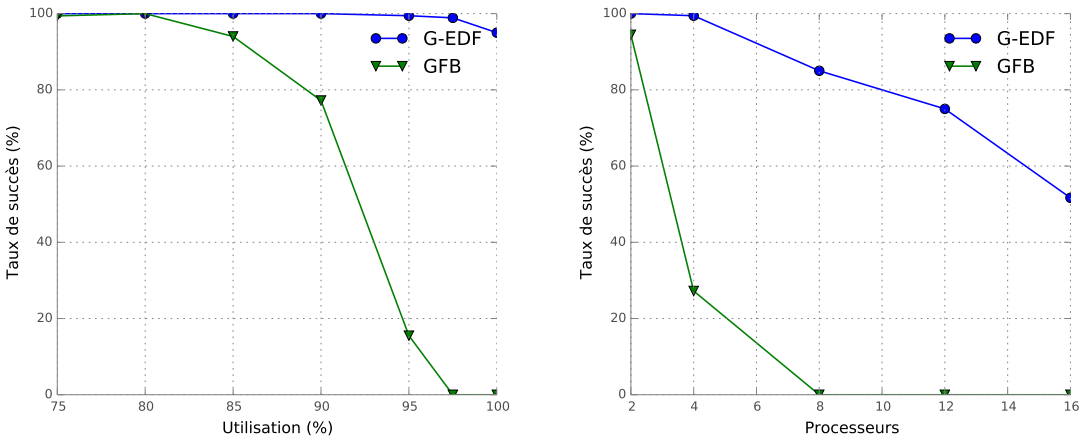
Nous voyons ici que pour un taux d'utilisation du système de 85%, plus de 80% des systèmes ont pu être ordonnancés sans aucun dépassement d'échéance pendant la simulation. A contrario, selon le test GFB, moins de 25% des systèmes sont ordonnançables. Pour un taux d'utilisation de 97.5%, le test n'a trouvé aucun système ordonnançable alors que la simulation n'a engendré aucun dépassement d'échéance pour près de la moitié des systèmes.

Sur la figure 5.6a, nous pouvons constater que l'algorithme G-EDF est capable d'ordonnancer sans dépassement d'échéance plus de 90% des systèmes simulés et chargés à 100% lorsqu'il n'y a que deux processeurs. Sur la figure 5.6b, nous voyons que le taux de systèmes correctement ordonnancés en simulation chute avec le nombre de processeurs. De plus, nous constatons que le test GFB donne de moins bons résultats avec l'augmentation du nombre de processeurs.

Conclusion

Les résultats obtenus sont en accord avec le constat de Lelli *et al.*, à savoir que le nombre de systèmes correctement ordonnancés par G-EDF est largement supérieur au nombre de systèmes théoriquement ordonnançables selon le test GFB. De plus, une étude similaire a été réalisée dans le passé avec un choix de périodes permettant de simuler sur toute l'hyper-période [BCL05a]. Leurs résultats montrent la même tendance avec un écart encore plus important, probablement lié au choix des périodes. Cette évaluation s'est limitée au test GFB mais il existe d'autres tests que l'on peut généralement combiner afin d'augmenter le nombre de systèmes dont l'ordonnançabilité est prouvée.

Ces expérimentations avec l'ordonnanceur G-EDF indiquent également qu'un grand nombre de systèmes n'ont pas provoqué de dépassements d'échéance, en particulier lorsque le



(a) Taux de succès en fonction du taux d'utilisation pour 2 processeurs.

(b) Taux de succès en fonction du nombre de processeurs pour une utilisation totale de 85%.

FIGURE 5.6 – Taux de succès de G-EDF par simulation comparé au taux de systèmes ordonnançables selon le critère GFB. Le nombre de tâches est compris dans {20, 30, 40, 50, 60, 70, 80, 90, 100}.

nombre de processeurs est faible. Ce constat nous incite donc à étudier plus en détail le comportement de G-EDF et de ses variantes censées l'améliorer.

5.2.2 Comparaison des algorithmes de type G-EDF

Au cours du premier chapitre, nous avons vu que plusieurs politiques visent à améliorer les performances de G-EDF. Dans un premier temps, les algorithmes EDF-US, PriD, G-FL et EDZL sont comparés à G-EDF en matière d'ordonnançabilité. Pour rappel, les algorithmes EDF-US et PriD s'inspirent des tests d'ordonnançabilité pour permettre d'améliorer le nombre de systèmes ordonnançables. Ces derniers fixent une priorité maximale aux tâches dont le taux d'utilisation est le plus fort. G-FL propose d'utiliser une priorité légèrement différente de G-EDF et prenant en compte le WCET. L'algorithme EDZL change dynamiquement la priorité des travaux dont la laxité devient nulle afin de les rendre plus prioritaires. Nous ajoutons à ces politiques G-MLLF, variante de G-LLF, qui utilise la laxité pour déterminer les travaux les plus prioritaires.

Nous évaluons ensuite ces algorithmes du point de vue des préemptions et migrations de travail qui sont engendrées. De par la simplicité du fonctionnement de G-EDF et en particulier du nombre limité de points d'ordonnancement, G-EDF devrait a priori engendrer relativement peu de préemptions et migrations. Les politiques EDF-US, PriD et G-FL n'engendrent pas plus de points d'ordonnancement mais affectent des priorités différentes aux tâches comparé à EDF. Ainsi, il est intéressant de voir quel impact cela peut avoir sur les résultats. EDZL introduit des points d'ordonnancement supplémentaires mais doit se comporter de manière identique à EDF en l'absence de dépassement d'échéance [PHK+05]. Enfin G-LLF est décrié pour son nombre de préemptions et migrations trop important [OY98]. Nous étudierons donc sa variante G-MLLF qui doit permettre d'atténuer ce défaut.

Ordonnançabilité

La figure 5.7 montre le taux de systèmes correctement ordonnancés par ces différentes politiques en fonction du taux d'utilisation du système. Chaque point correspond à la moyenne obtenue sur 900 systèmes en faisant varier le nombre de tâches dans 20, 30, 40, 50, 60, 70, 80, 90, 100} et le nombre de processeurs dans {2, 4, 8, 12, 16}.

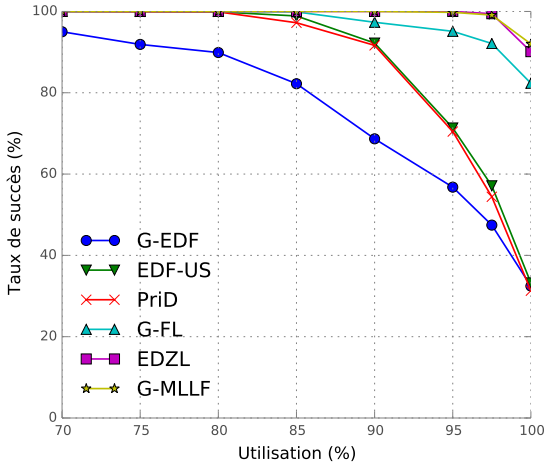


FIGURE 5.7 – Comparaison des politiques similaires à G-EDF en termes de taux de succès par simulation. Le nombre de tâches est compris dans {20, 30, 40, 50, 60, 70, 80, 90, 100} et le nombre de processeurs dans {2, 4, 8, 12, 16}.

Les politiques utilisant la laxité donnent les meilleurs résultats mais elles nécessitent cependant d'activer des timers et de changer dynamiquement la priorité des travaux. En revanche, la politique G-FL ne nécessite qu'une modification très mineure de G-EDF et pourtant les résultats sont bien meilleurs et relativement proches de ceux obtenus par EDZL et G-MLLF.

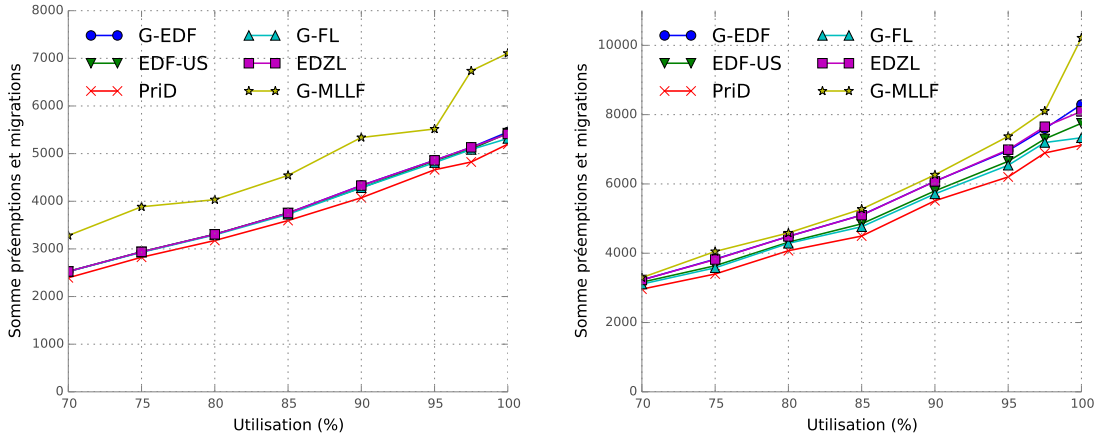
Les algorithmes PriD et EDF-US donnent des résultats très proches. Bien que les résultats obtenus soient bien meilleurs que ceux pour G-EDF, à partir d'un taux d'utilisation relatif d'environ 90%, les résultats s'effondrent jusqu'à atteindre ceux de G-EDF.

Préemptions et migrations

La stratégie de placement des tâches sur les processeurs n'est pas clairement définie par les algorithmes que nous étudions, or ceci peut potentiellement avoir un impact sur le nombre de préemptions et migrations. Bien que nous n'ayons pas constaté de différence notable entre deux implémentations différentes de G-EDF, cette question ne doit pas être ignorée lors de la mise en œuvre d'un algorithme. Compte tenu de la similitude entre les différentes politiques étudiées ici, tous les ordonnanceurs (en dehors de G-MLLF) sont basés sur la même implémentation que G-EDF afin de ne pas fausser les résultats. À chaque activation ou fin d'exécution, l'algorithme G-EDF commence par choisir soit un processeur libre soit le processeur qui exécute la tâche la moins prioritaire. En cas d'égalité, c'est le processeur sur lequel la tâche s'est activée ou terminée qui est choisi. Ce processeur exécute alors le travail en attente le plus prioritaire, sauf si le travail en cours d'exécution est plus prioritaire que ce dernier.

La figure 5.8 indique la somme du nombre de préemptions et migrations de travail en fonction de l'utilisation totale du système pour 100 tâches. L'influence du nombre de tâches sera étudié dans un second temps. À l'exception de G-MLLF, les résultats sont très

proches. Nous pouvons constater une augmentation linéaire en fonction de l'utilisation pour la plupart des politiques. Avec l'augmentation du nombre de processeurs, les résultats obtenus par G-MLLF s'approchent de ceux obtenus par les autres politiques.



(a) Avec 8 processeurs.

(b) Avec 16 processeurs.

FIGURE 5.8 – Somme du nombre de préemptions et migrations en fonction du taux d'utilisation avec 100 tâches.

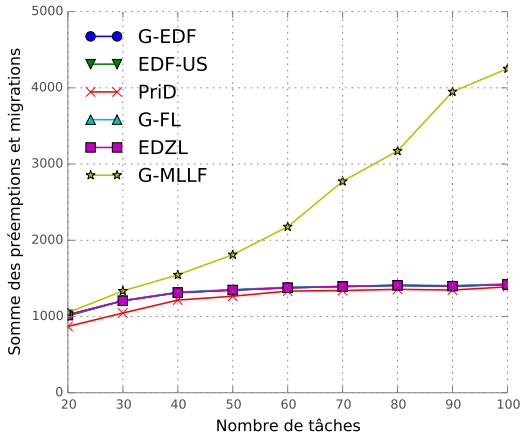
La figure 5.9 montre le nombre de préemptions et migrations en fonction du nombre de tâches avec un taux d'utilisation total de 95% et un nombre variable de processeurs. L'augmentation du nombre de processeurs engendre généralement une augmentation du nombre de préemptions et migrations. Cependant, lorsque le nombre de processeurs s'approche du nombre de tâches, nous pouvons également constater une diminution car l'ordonnanceur est de plus en plus souvent en mesure de placer des travaux seuls sur des processeurs ce qui empêche toute préemption.

La figure 5.9 permet aussi de confirmer l'observation faite ci-dessus qui indiquait que G-MLLF produit de meilleurs résultats avec un grand nombre de processeurs. L'intérêt de G-MLLF réside surtout dans son taux de succès (voir partie 5.2.2), les résultats obtenus dans cette partie indiquent que G-MLLF est une alternative qu'il convient de considérer lorsque le nombre de processeurs est important ou le nombre de tâches faible.

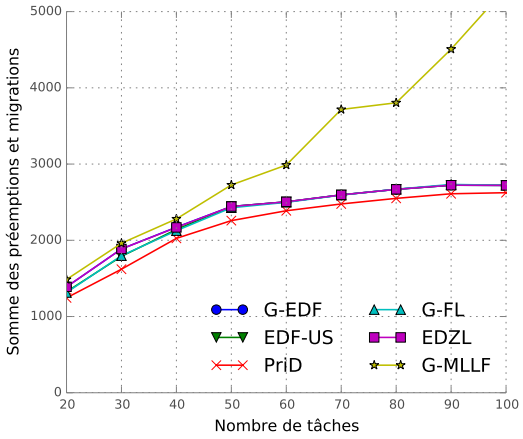
Conclusion

Les variantes de G-EDF que nous avons considérées réduisent de manière importante le nombre dépassements d'échéance. Pour ce faire, les algorithmes ont intégré le WCET dans le calcul des priorités : EDF-US et PriD le font à travers les u_i, et EDZL, G-FL et G-MLLF utilisent les C_i. L'algorithme d'ordonnancement G-EDF reste cependant la seule politique utilisable sans connaissance du WCET. Ceci peut constituer un avantage dans le cas de systèmes temps réel souples où le WCET n'est pas toujours disponible.

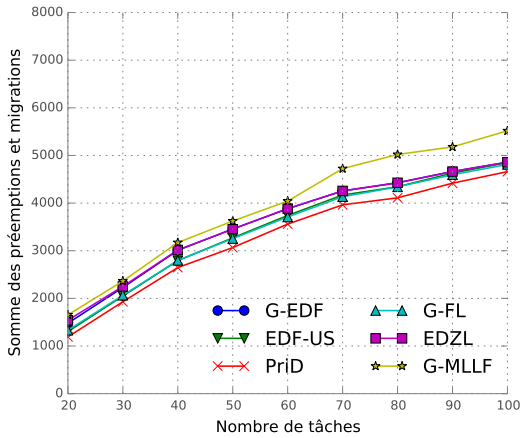
Au regard du nombre de préemptions et migrations, il n'y a pas de différences importantes entre les différents algorithmes étudiés, en dehors de G-MLLF. Le nombre de préemptions et migrations engendrées par ce dernier tend à se confondre avec ceux des autres algorithmes lorsqu'il y a peu de tâches ou beaucoup de processeurs. S'agissant de l'algorithme ayant ordonnancé sans dépassement d'échéance le plus grand nombre de systèmes, cet algorithme n'est donc pas à écarter.



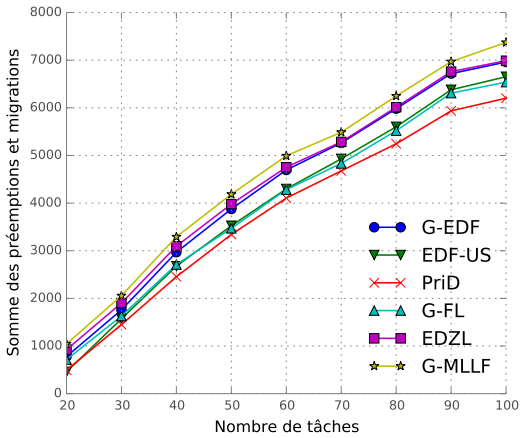
(a) Avec 2 processeurs, $\sum u_i = 1.9$.



(b) Avec 4 processeurs, $\sum u_i = 3.8$.



(c) Avec 8 processeurs, $\sum u_i = 7.6$.



(d) Avec 16 processeurs, $\sum u_i = 15.2$.

FIGURE 5.9 – Somme du nombre de préemptions et migrations en fonction du nombre de tâches avec un taux d'utilisation de 95%.

5.3 Ordonnancement partitionné

Le reproche principal souvent adressé à l'approche partitionnée est qu'il existe des systèmes au taux d'utilisation pourtant faible (qui tend vers 50%) qui ne sont pas ordonnançables. Les politiques semi-partitionnées permettent d'augmenter le nombre de systèmes ordonnançables en autorisant une migration contrôlée des tâches. Par ailleurs, certaines politiques telles que EDF-WM ou RUN débutent par une étape de partitionnement et si le système est partitionnable, alors ces politiques produisent un ordonnancement partitionné.

Les auteurs de RUN ont affirmé qu'au cours de leurs expérimentations, la plupart des systèmes peu chargés (taux d'utilisation inférieur à 75%) étaient partitionnables [5] et que des systèmes avec un taux d'utilisation allant jusqu'à 94% ont pu être partitionnés [RLM+11]. Dans un premier temps, nous reproduisons cette évaluation et nous l'étendons en étudiant l'influence du nombre de tâches et de processeurs sur les résultats. Ensuite, nous étudions l'impact de l'heuristique de partitionnement utilisée sur le nombre de systèmes ordonnançables et sur le nombre de préemptions. Et nous terminons par une comparaison de G-EDF et P-EDF.

5. La condition de remplissage d'un processeur est : $\sum u_i \leq 1$

5.3.1 Systèmes partitionnables

Dans l'article présentant RUN, les auteurs fournissent la figure 5.10 qui donne le taux de systèmes partitionnables selon un algorithme de placement *worst-fit* (voir partie 1.3.1). Ce taux est calculé à partir de 1000 ensembles de tâches pour chaque point.

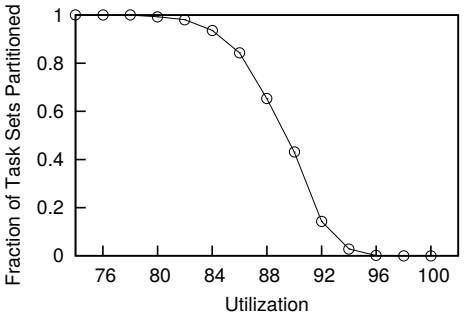


FIGURE 5.10 – Taux de systèmes partitionnables avec 24 tâches sur 16 processeurs selon les auteurs de RUN [RLM+11].

Ce résultat est intéressant car, comme nous le verrons au cours de ce chapitre, P-EDF fait partie des politiques faisant le moins de préemptions et migrations [6] tout en étant simple à mettre en œuvre. Nous voudrions donc savoir quelle est la part des systèmes partitionnables et par conséquent ordonnançables. Nous étendons donc les expérimentations faites par Regnier *et al.* à d'autres configurations.

Résultats

Nous avons commencé par reproduire les résultats donnés par Regnier *et al.* à l'aide du générateur RandFixedSum intégré dans SimSo et l'algorithme de partitionnement *Decreasing Worst-Fit* [7]. Nous avons gardé le même nombre de processeurs, à savoir 16 mais nous avons fait varier le nombre de tâches. Le taux de systèmes correctement partitionnés en fonction de l'utilisation et pour différents nombres de tâches est donné par la figure 5.11. Chaque point a été calculé à partir de 1000 ensembles de tâches.

Les mêmes expérimentations ont ensuite été relancées sur des systèmes dotés de 8 processeurs (voir figure 5.12). En observant attentivement les figures 5.11 et 5.12, nous constatons une forte similitude entre les résultats obtenus pour 40 tâches et 16 processeurs, et les résultats pour 20 tâches et 8 processeurs. La figure 5.13 reprend ces résultats en y ajoutant ceux obtenus pour 10 tâches sur 4 processeurs.

Conclusion

Les résultats obtenus à l'aide de SimSo sont quasiment identiques à ceux publiés par Regnier *et al.* [RLM+11] (les écarts s'expliquent par l'aspect aléatoire des tirages). Notons que, dans cette configuration, le nombre de tâches est relativement faible par rapport au nombre de processeurs ce qui est un cas défavorable. En effet, les résultats obtenus indiquent que le taux de systèmes partitionnables augmente avec l'augmentation du nombre de tâches ou la diminution du nombre de processeurs. Ceci s'explique par le fait que plus le nombre de tâches par rapport au nombre de processeurs est grand et plus les tâches sont

6. P-EDF ne fait pas de migration mais compense par une augmentation du nombre de préemptions.

7. Notre choix des périodes est différent mais seul le taux d'utilisation des tâches est pris en compte par les algorithmes de bin-packing que nous étudions. Regnier *et al.* utilisent également l'algorithme RandFixedSum pour générer les taux d'utilisation.

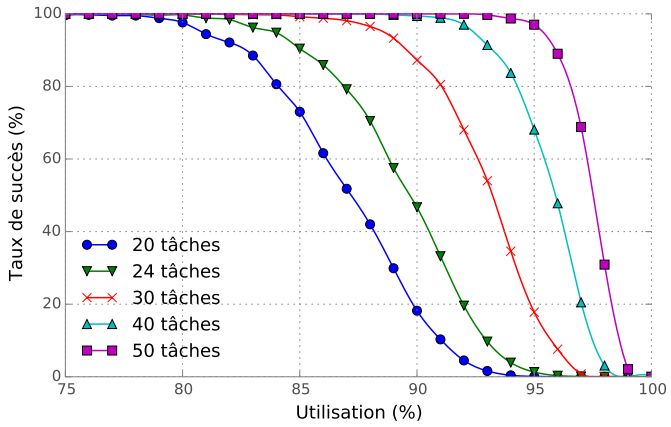


FIGURE 5.11 – Taux de systèmes partitionnables sur 16 processeurs en fonction du taux d'utilisation.

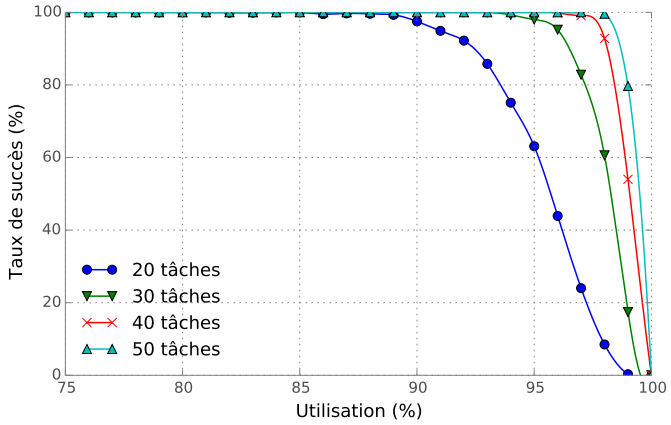


FIGURE 5.12 – Taux de systèmes partitionnables sur 8 processeurs en fonction du taux d'utilisation.

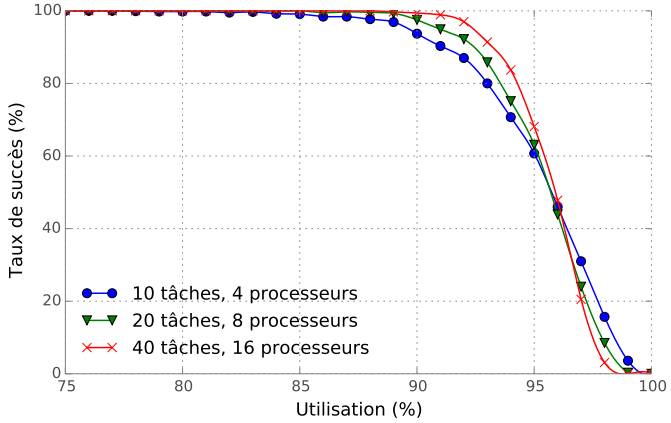


FIGURE 5.13 – Taux de systèmes partitionnables en fonction du taux d'utilisation.

légères. Or, le problème de bin-packing est plus facile à résoudre avec de petites valeurs. La figure 5.13 montre que les résultats obtenus pour un rapport constant entre le nombre de tâches est de processeurs sont très proches, mais sans être pour autant identiques.

Certains algorithmes tels que EDF-WM ou RUN vont donc se comporter dans de nombreux cas comme un algorithme P-EDF. Les résultats concernant P-EDF sont donc valables pour ces politiques lorsque le partitionnement réussit.

5.3.2 Impact du placement des tâches

Au cours de la section 1.3.1, nous avons vu l'existence de multiples algorithmes permettant l'affectation des tâches aux processeurs dans le cadre d'un ordonnancement par partitionnement. Nous nous intéressons ici à la comparaison des algorithmes de placement *Decreasing First-Fit* et *Decreasing Worst-Fit* lorsqu'ils sont utilisés pour ordonnancer les tâches à l'aide de l'ordonnanceur P-EDF. Dans un premier temps, nous traitons du taux de succès de l'étape de partitionnement de ces algorithmes. Ensuite, nous verrons si ces algorithmes ont également un impact sur les performances du système en matière de préemptions et temps de réponse.

Taux de succès

L'algorithme *Decreasing First-Fit* est connu comme étant l'un des meilleurs algorithmes gloutons pour le bin-packing. Les résultats issus des simulations montrent effectivement un plus grand ensemble de systèmes ordonnançables pour *Decreasing First-Fit*. Afin de confirmer et affiner ces résultats, ces deux algorithmes ont été testés sur 1000 ensembles pour chaque configuration de 30 tâches, 12 processeurs et un taux d'utilisation du système allant de 75% à 100% par pas de un.

La figure 5.14 montre le ratio du nombre de systèmes ordonnançables selon ces deux algorithmes. Nous pouvons constater que l'algorithme *Decreasing First-Fit* permet de partitionner un plus grand ensemble de systèmes que l'algorithme *Decreasing Worst-Fit*. Cette tendance se retrouve dans toutes les configurations que nous avons testées.

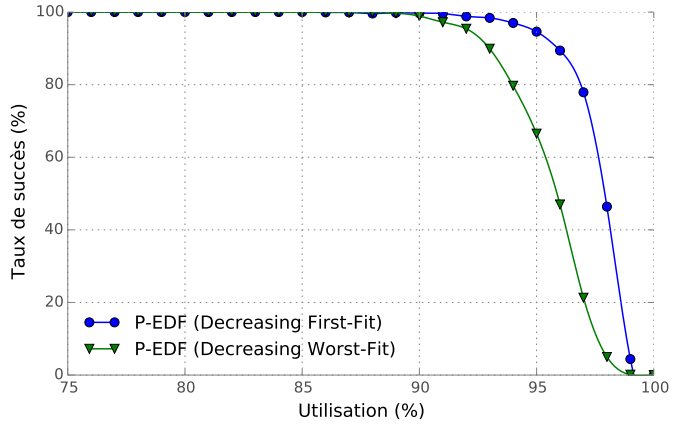


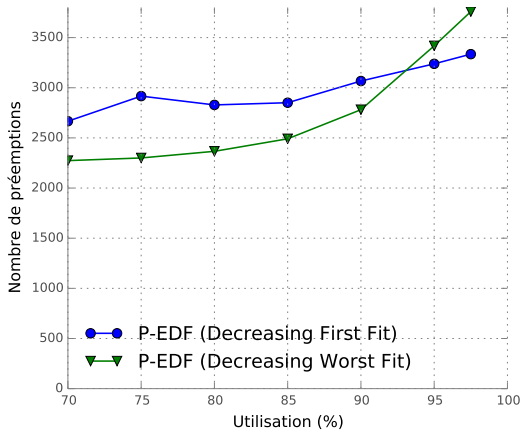
FIGURE 5.14 – Taux de systèmes ordonnançables selon l'heuristique de partitionnement avec 12 processeurs et 30 tâches.

Notons également que bien que *Decreasing First-Fit* soit capable de partitionner un ensemble plus grand de systèmes, il existe des systèmes que *Decreasing Worst-Fit* peut partitionner mais que l'algorithme *Decreasing First-Fit* ne peut pas.

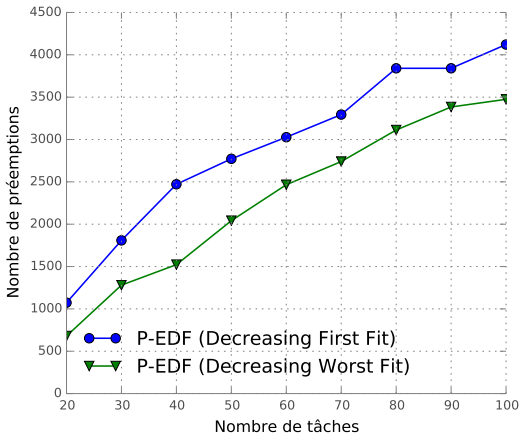
Préemptions

L'algorithme *Decreasing First-Fit* a pour objectif de réduire le nombre de processeurs nécessaires en concentrant les tâches sur les premiers processeurs. Par conséquent, pour un système peu chargé, certains processeurs peuvent être laissés inactifs ou faiblement chargés. Cette concentration des tâches sur les premiers processeurs peut intuitivement être la source de plus de préemptions que si l'occupation des différents processeurs était plus équilibrée. Au contraire, l'algorithme *Worst-Fit* va naturellement tenter d'équilibrer la charge des processeurs en plaçant chaque tâche sur le processeur le moins chargé.

Les résultats obtenus indiquent en effet que le placement *Decreasing Worst-Fit* permet de réduire le nombre de préemptions. C'est particulièrement le cas pour un système peu chargé (voir figure 5.15a). Le nombre de tâches ne semble cependant pas être un facteur important (voir figure 5.15b).



(a) Pour 12 processeurs, nombre de tâches compris entre 20 et 100.



(b) Taux d'utilisation de 75% et 12 processeurs.

FIGURE 5.15 – Nombre de préemptions observées en fonction de l'algorithme de placement utilisé.

Temps de réponse

La répartition équitable des taux d'utilisation produit par l'algorithme *Decreasing Worst-Fit* devrait également avoir un impact positif sur les temps de réponse des tâches. Pour mesurer cela, nous avons étudié la laxité normalisée pour l'algorithme P-EDF en utilisant les deux méthodes d'affectation des tâches aux processeurs. La figure 5.16 montre les résultats obtenus pour l'exécution de 100 tâches sur 16 processeurs en faisant varier le taux d'utilisation total. Nous voyons que dans cette configuration, l'algorithme *Decreasing Worst-Fit* est légèrement meilleur en matière de laxité normalisée. Ce résultat se retrouve pour toutes les configurations que nous avons analysées, avec un écart qui devient encore plus faible avec l'utilisation du modèle ACET.

Conclusion

Les résultats qui viennent d'être présentés montrent l'importance du choix de l'algorithme de placement des tâches. D'une part, il faut que l'algorithme de placement soit en mesure de partitionner l'ensemble de tâches, mais comme montré par la comparaison de ces deux algorithmes, le placement peut aussi avoir une incidence sur les performances du système en matière de nombre de préemptions et temps de réponse.

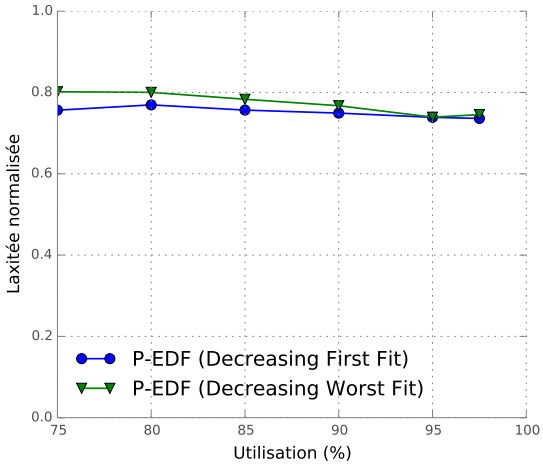


FIGURE 5.16 – Laxité normalisée en fonction du taux d'utilisation du système pour 100 tâches et 16 processeurs.

Puisque l'affectation des tâches aux processeurs se fait hors-ligne, il est possible d'imaginer de nouveaux algorithmes visant à améliorer les performances. Tout comme il serait possible de combiner les heuristiques existantes. Par exemple, il est possible de commencer par un *Decreasing Worst-Fit* qui provoque moins de préemptions et de meilleurs temps de réponse, et si le partitionnement échoue, l'algorithme *Decreasing First-Fit* pourrait être utilisé. De plus, le nombre total de systèmes partitionnables serait légèrement augmenté car certains systèmes sont partitionnables par ces algorithmes ne sont pas les mêmes.

5.3.3 Comparaison G-EDF et P-EDF

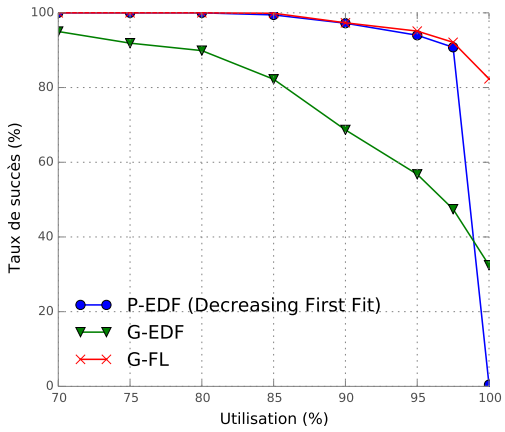
Ordonnançabilité

Du point de vue de l'ordonnançabilité, aucune des politiques G-EDF ou P-EDF ne domine l'autre. Autrement dit, il existe des systèmes ordonnançables par l'une qui ne le sont pas par l'autre et inversement. Nos simulations nous apportent en effet un certain nombre d'exemples qui confirment cela.

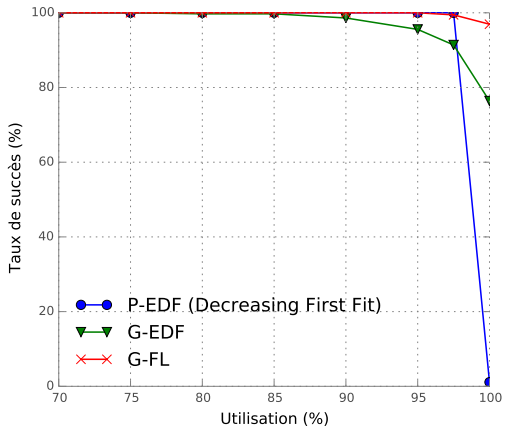
Nous avons souhaité évaluer le nombre de systèmes qui ont pu être ordonnancés sans dépassement d'échéance. Nous avons ajouté à ces deux algorithmes l'algorithme G-FL qui permet d'améliorer le taux de systèmes ordonnançables de G-EDF par une modification mineure (voir figure 5.7).

La figure 5.17a montre le taux de systèmes ordonnançables avec un nombre de processeurs compris dans {2, 4, 8, 12, 16} et un nombre de tâches compris dans {20, 30, 40, 50, 60, 70, 80, 90, 100}. Les expérimentations montrent que P-EDF a pu exécuter sans dépassement d'échéance un plus grand nombre de systèmes que G-EDF. En revanche, G-FL fait mieux que P-EDF. Attention cependant, pour P-EDF, nous avons la garantie qu'il n'y aura pas de dépassement d'échéance alors que pour G-FL, la valeur indique seulement qu'il n'y a pas eu de dépassement durant la simulation.

Mais nous savons également grâce aux résultats précédents que G-EDF est capable de donner de très bons résultats lorsqu'il y a peu de processeurs (moins de 4). La figure 5.17b donne le taux de systèmes ordonnançables avec un nombre de processeurs égal à 2 ou 4 et un nombre de tâches compris dans {20, 30, 40, 50, 60, 70, 80, 90, 100}.



(a) Pour un nombre de processeurs dans {2, 4, 8, 12, 16}.



(b) Pour un nombre de processeurs dans {2, 4}.

FIGURE 5.17 – Comparaison des politiques P-EDF, G-EDF et G-FL en terme de taux de succès en fonction du taux d'utilisation du système. Le nombre de tâches est compris dans {20, 30, 40, 50, 60, 70, 80, 90, 100}.

Préemptions et migrations

La différence majeure entre G-EDF et P-EDF est que P-EDF n'autorise pas les migrations. D'un côté, G-EDF est capable d'utiliser un processeur libre là où P-EDF devra peut-être créer une préemption. Mais d'un autre côté, le fait d'utiliser une seule liste de tâches augmente la compétition et donc probablement le nombre de préemptions. Mais en pratique, G-EDF et P-EDF réagissent aux mêmes évènements et on peut donc s'attendre à un nombre similaire de préemptions et migrations si on somme ces valeurs.

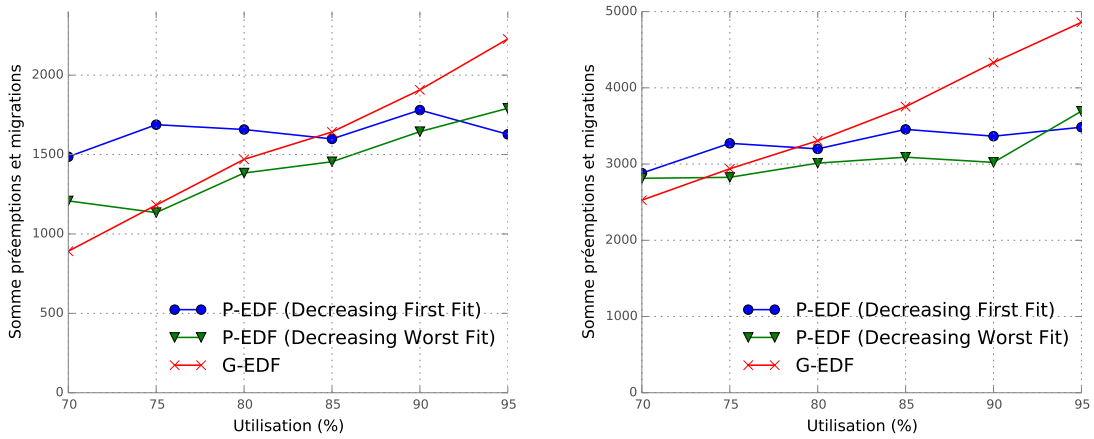
Comme le montrent les figures 5.18 et 5.19, le nombre de préemptions et migrations provoquées par G-EDF dépend linéairement du taux d'utilisation du système alors que le nombre de préemptions provoquées par P-EDF varie faiblement. Par conséquent, à partir d'un certain taux d'utilisation, P-EDF devient avantageux. Ces figures montrent aussi que G-EDF est plus intéressant que P-EDF lorsqu'il y a peu de tâches comparé au nombre de processeurs.

Conclusion

En conclusion, P-EDF représente un meilleur choix que G-EDF dans une majorité de cas que ce soit en matière d'ordonnançabilité ou de préemptions et migrations. Mais l'algorithme G-EDF se révèle malgré tout intéressant dans le cas où le système est composé de moins de 4 processeurs et avec un nombre de tâches assez faible.

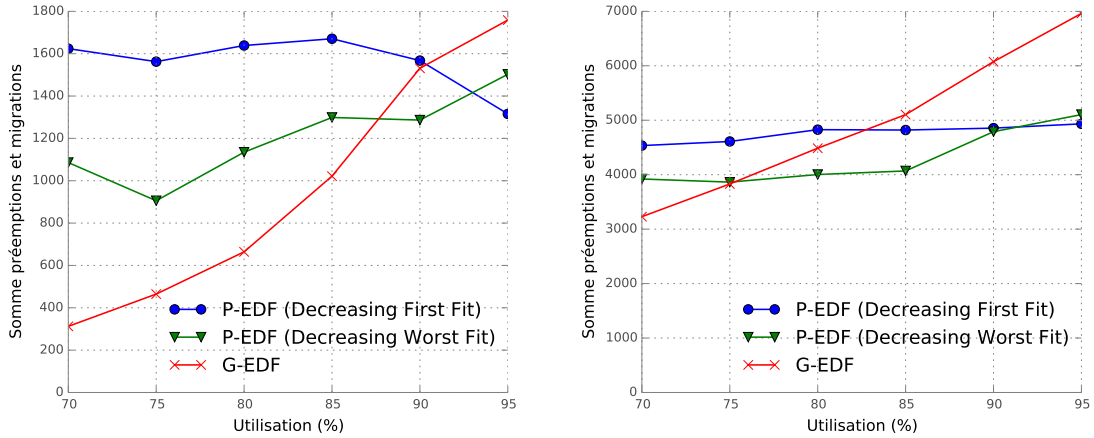
5.4 Comparaison des politiques DP-Fair

LLREF est l'une des premières politiques d'ordonnancement publiée de type DP-Fair [CR06]. Elle a donc naturellement servi de base ainsi que de point de comparaison pour les politiques DP-Fair suivantes. Dans cette partie, puisqu'il s'agit de politiques optimales, nous nous intéressons à la comparaison entre LLREF et les politiques LRE-TL et DP-WRAP seulement en matière de préemptions et migrations. La partie 1.5.2 a introduit les notions



(a) Avec 30 tâches (b) Avec 100 tâches

FIGURE 5.18 – Somme du nombre de préemptions et migrations pour un système composé de 8 processeurs.



(a) Avec 30 tâches (b) Avec 100 tâches

FIGURE 5.19 – Somme du nombre de préemptions et migrations pour un système composé de 16 processeurs.

minimales à connaître au sujet des politiques DP-Fair. En revanche, une bonne connaissance du fonctionnement des politiques discutées ici est nécessaire pour comprendre les explications qui suivent.

5.4.1 Comparaison de LRE-TL et LLREF

LRE-TL est une politique d'ordonnancement DP-Fair largement inspirée de LLREF et qui apporte deux contributions par rapport à LLREF. La première, qui ne sera pas évaluée ici, est le support de tâches sporadiques. La seconde est la réduction du nombre de préemptions et migrations. Les auteurs annoncent une amélioration d'un facteur égal au nombre de tâches. En réalité, il s'agit d'un pire cas et nous souhaiterions vérifier ces résultats par simulation.

Contrairement à LLREF (voir partie 1.5.2), LRE-TL ne trie pas les tâches par ordre de *local-remaining execution time* à chaque réordonnancement et limite au maximum les

interruptions de tâche. Outre une amélioration du temps de calcul de l'ordonnanceur, ceci devrait logiquement permettre aussi de provoquer moins de préemptions et migrations. Dans la suite, on note n le nombre de tâches et m le nombre de processeurs.

Les auteurs écrivent [FN09] :

> Because LLREF sorts the tasks upon every B or C event, a single event could cause m tasks to be preempted. Upon resuming execution, each of these tasks might end up on a different processor. Thus, the maximum number of preemptions and migrations within each TL-plane is $O(mn)$. By contrast, LRE-TL preempts only when absolutely necessary – i.e., only when a C event occurs. Because utilization decreases over time, the number of C events within a TL-plane is at most $(m-1)$. Hence the number of preemptions and migrations is $O(m)$.

La première affirmation est que LLREF peut provoquer jusqu'à m préemptions (ou migrations) suite à un évènement B ou C à cause du tri. Le nombre maximum de préemptions et migrations dans chaque intervalle TL-plane est donc bien de $O(mn)$. Mais ceci ne nous informe pas sur la valeur réelle observée en pratique.

Concernant LRE-TL, le nombre d'évènements C par TL-plane ne peut effectivement pas dépasser m pour un système ordonnançable. Cependant, le découpage des tâches dans des intervalles de temps DP-Fair implique des préemptions (ou migrations) lorsqu'une dotation locale a été consommée (évènement B) mais que le travail n'avait pas fini son exécution. Ceci est illustré à travers l'exemple donné par la figure 5.20. Le nombre total de préemptions et migrations ne peut cependant pas dépasser n ce qui reste bien inférieur à LLREF.

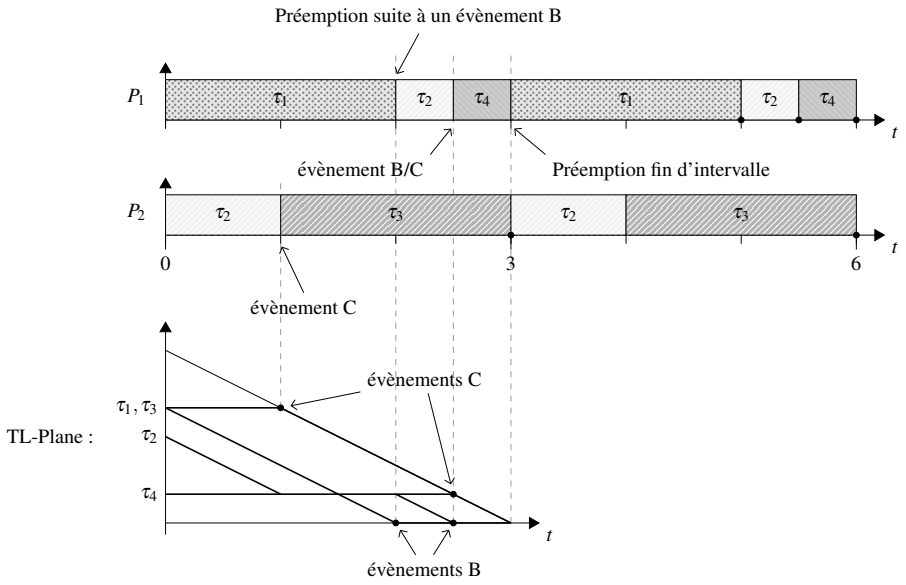
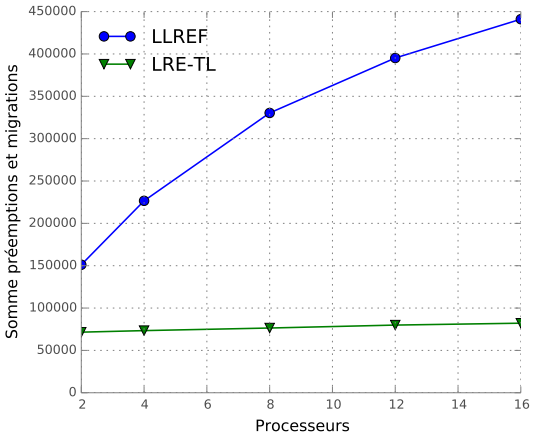
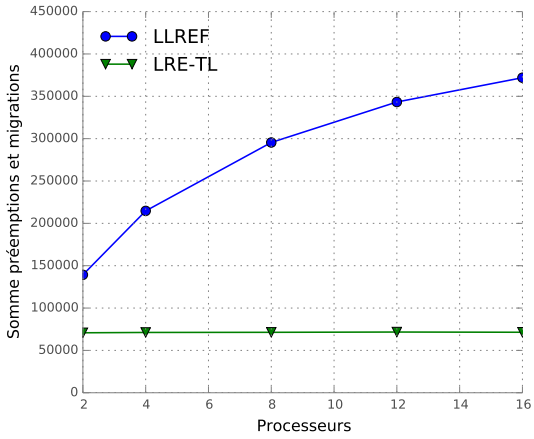


FIGURE 5.20 – Ordonnancement LRE-TL des tâches $\tau_1(4,6)$, $\tau_2(3,6)$, $\tau_3(2,3)$ et $\tau_4(1,6)$. Les tâches τ_1 et τ_4 sont préemptées mais ce n'est pas suite à un évènement C.

La figure 5.21 montre que le nombre de préemptions et migrations ne semble pas dépendre du nombre de processeurs pour LRE-TL contrairement à LLREF. Nous voyons un net avantage de LRE-TL sur LLREF dans toutes les configurations de plus de 40 tâches. De plus, le taux d'utilisation total du système a un impact faible sur le nombre de préemptions et migrations puisque les résultats restent proches entre les figures 5.21a et 5.21b.



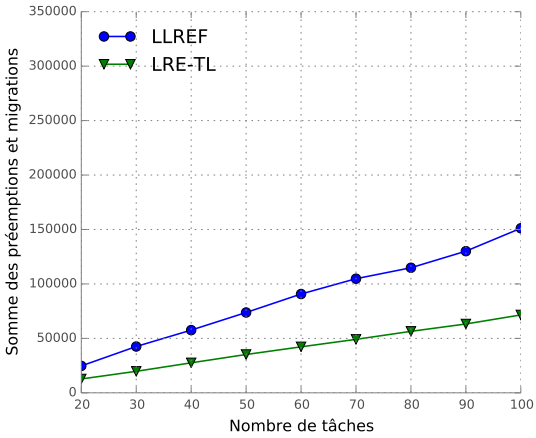
(a) Système chargé à 100%.



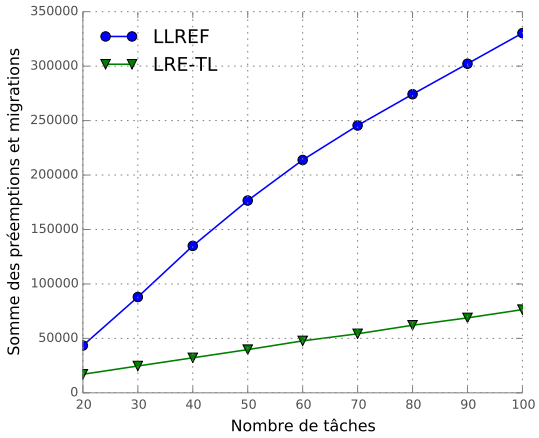
(b) Système chargé à 80%.

FIGURE 5.21 – Nombre de préemptions et migrations en fonction du nombre de processeurs pour un système comprenant 100 tâches et chargé respectivement à 100% et 80%.

Au contraire, les figures 5.22a et 5.22b montrent que, pour LRE-TL comme pour LLREF, le nombre de préemptions et migrations est une fonction affine du nombre de tâches. La pente de la courbe correspond aux résultats de LLREF augmente avec le nombre de processeurs. Et nous constatons à nouveau que les résultats concernant LRE-TL ne dépendent pas du nombre de processeurs contrairement à LLREF.



(a) Avec 2 processeurs.



(b) Avec 8 processeurs.

FIGURE 5.22 – Nombre de préemptions et migrations en fonction du nombre de tâches pour un système chargé à 100% avec respectivement 2 et 8 processeurs.

Conclusion

Les auteurs de LRE-TL annoncent que leur algorithme est capable de réduire le nombre de préemptions et migrations par rapport à LLREF par un facteur égal au nombre de tâches (n) en passant de $O(mn)$ pour LLREF à $O(m)$ pour LRE-TL [FN09]. Expérimentalement, nous voyons à travers les figures 5.22 et 5.21 que le nombre de préemptions et migrations pour LLREF est de l'ordre de $O(mn)$ comme attendu (augmentation linéaire en fonction de m et de n). En revanche, le nombre de préemptions et migrations pour LRE-TL ne dépend pas du nombre de processeurs (figure 5.21) mais dépend linéairement du nombre de tâches (figure 5.22) soit $O(n)$. Nos résultats montrent donc une réduction d'un facteur

légèrement inférieur au nombre de processeurs (m).

Bien que nous n'ayons pas retrouvé les mêmes résultats que ceux prédits par les auteurs de LRE-TL, nous nous accordons à dire que LRE-TL permet de réduire de façon significative le nombre de préemptions et migrations. Pour des raisons de place, tous les résultats ne peuvent pas être affichés, mais les résultats indiquent aussi que la réduction la plus importante est faite sur le nombre de migrations.

5.4.2 DP-WRAP

Levin *et al.* ont proposé en 2010 l'algorithme DP-WRAP [LFS+10]. Cet algorithme se base sur l'algorithme de McNaughton pour déterminer les tâches à exécuter dans un intervalle de temps. Les auteurs de DP-WRAP jugent également inutile le tri effectué dans LLREF. De plus, ils annoncent que leur politique engendre moins de préemptions et migrations que LLREF d'après quelques simulations.

Concernant le nombre de préemptions et migrations, les auteurs annoncent que DP-WRAP produit au maximum $(n-1)$ préemptions et $(m-1)$ migrations par intervalle [LFS+10] :

> Theorem 6 : The DP-WRAP scheduling algorithm with mirroring in odd slices will produce at most $n-1$ context switches and $m-1$ migrations per slice.

Les simulations mettent en évidence un nombre de préemptions et migrations très similaires entre DP-WRAP et LRE-TL. À titre d'exemple, la figure 5.23 donne le nombre de préemptions et migrations en fonction du nombre de tâches pour un système chargé à 100% et doté de 8 processeurs. Aucune différence notable n'a été remarquée entre ces deux politiques pour l'ensemble des configurations simulées.

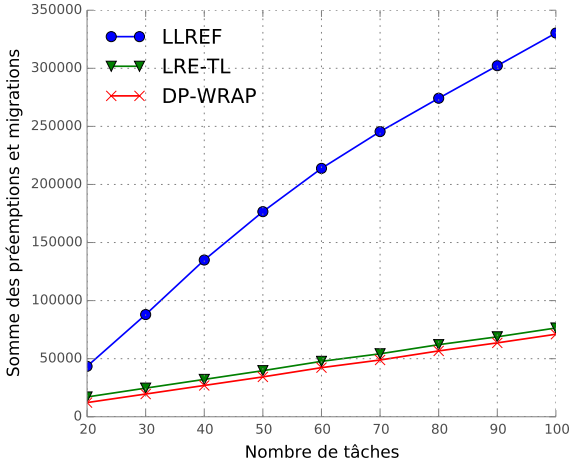


FIGURE 5.23 – Nombre de préemptions et migrations en fonction du nombre de tâches pour un système chargé à 100% et 8 processeurs.

Conclusion

L'algorithme DP-WRAP utilise une distribution temporelle (des durées d'exécution locales sur les intervalles et sur les processeurs) très différente de celle de LRE-TL qui se base sur la notion de TL-plane. Pourtant, les nombres de préemptions et migrations engendrées par ces deux politiques sont très proches.

5.5 Comparaison des politiques U-EDF, RUN et EKG

Nelissen *et al.* ont effectué des simulations afin de comparer leur politique U-EDF aux politiques EKG et RUN en matière de préemptions et migrations [NBN+12]. Dans cette partie, nous commentons les résultats de cette étude et nous proposons d'évaluer à notre tour ces trois algorithmes d'ordonnancement en étendant l'analyse déjà faite.

Génération des tâches

Les auteurs ont généré 1000 ensembles de tâches par expérimentation. Une expérimentation correspondant ici à un certain nombre de processeurs et un taux total d'utilisation. La génération des taux d'utilisation des tâches a été faite avec la méthode de Kato où le taux d'utilisation des tâches est compris dans l'intervalle [0.01, 0.99]. Chaque tâche possède une période entière comprise entre 5 et 100 selon une distribution uniforme. Les résultats obtenus pour un taux d'utilisation total égal au nombre de processeurs sont résumés par la figure 5.24.

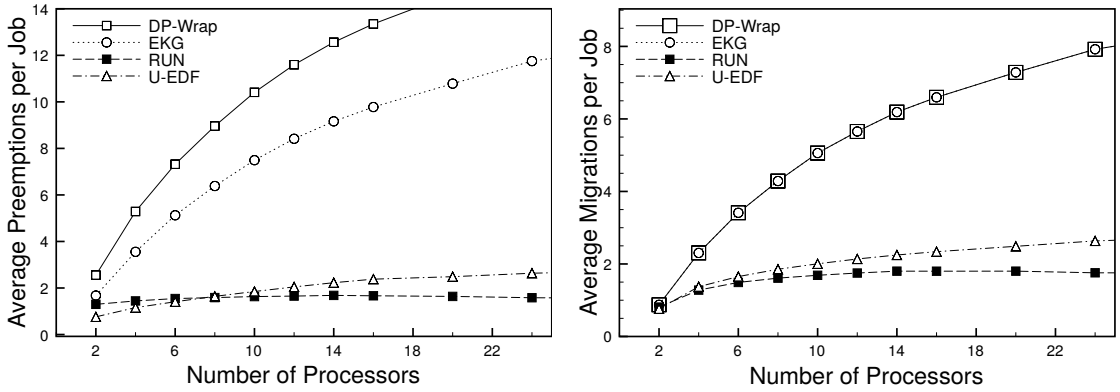


FIGURE 5.24 – Résultats obtenus par les auteurs de U-EDF pour un taux d'utilisation du système de 100% [NBN+12].

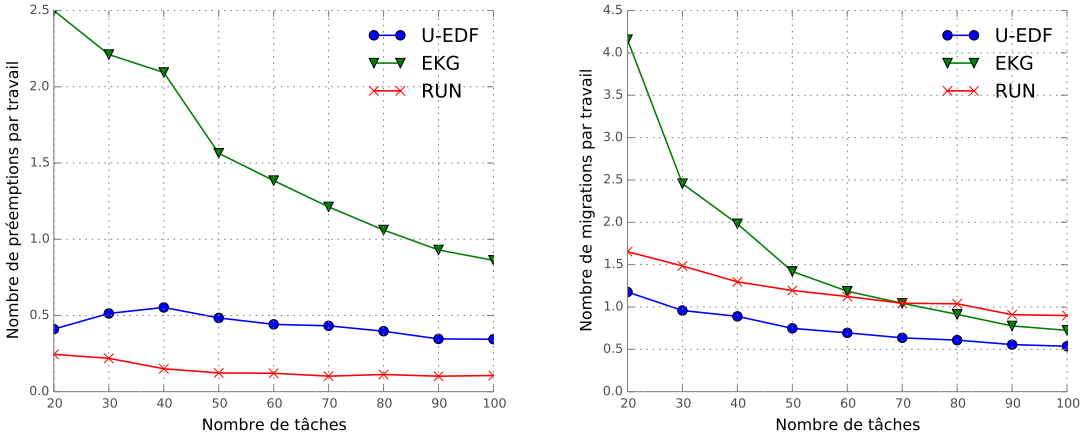
La méthode de génération de tâches utilisée fait que le nombre de tâches par ensemble varie en fonction du taux d'utilisation désiré (voir section 5.1.1). Une augmentation du nombre de processeurs est associée à une augmentation du nombre de tâches. Or, ces facteurs sont tous deux susceptibles de provoquer une augmentation du nombre de préemptions et migrations. Dans l'évaluation de Nelissen *et al.*, le nombre moyen de tâches par ensemble est environ égal au double du nombre de processeurs. L'inconvénient majeur de cette approche est que le nombre de tâches est ainsi relativement faible. En effet, l'ordonnancement de 4 tâches sur 2 cœurs ou 16 tâches sur une machine possédant 8 cœurs n'est pas représentatif de tout type de système.

Notre méthode de génération de tâches est différente et permet de dissocier l'augmentation du nombre de tâches de celle du nombre de processeurs. Nous étudions ici plus précisément l'impact de l'augmentation de chaque facteur.

Comparaison avec EKG

Comme le montre la figure 5.24, les résultats obtenus pour EKG ne sont pas très bons. Or le nombre de préemptions et migrations par travail pour EKG diminue rapidement avec une augmentation du nombre de tâches comme le montrent nos expériences (voir figure 5.25). Puisque le nombre de tâches ordonnancées dans les simulations faites par Nelissen *et al.* est de l'ordre du double du nombre de processeurs, soit moins d'une vingtaine dans

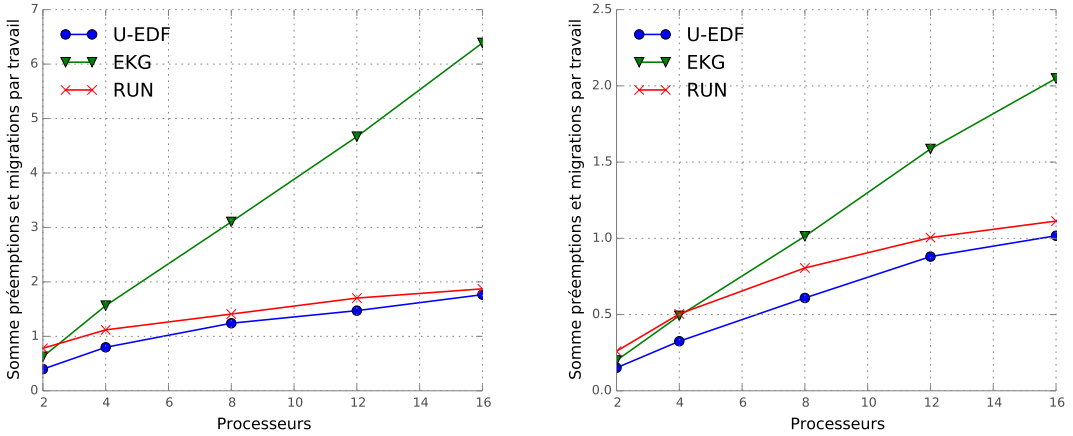
la plupart des cas, on peut en déduire que les systèmes simulés à l'origine des résultats de la figure 5.24 sont très défavorables à EKG.



(a) Préemptions par travail (b) Migrations par travail

FIGURE 5.25 – Nombre de préemptions et migrations par travail en fonction du nombre de tâches, avec un taux d'utilisation de 100% et 12 processeurs.

La figure 5.26 montre que l'augmentation du nombre de processeurs semble avoir un impact plus fort pour EKG que pour les autres politiques. Nous pouvons donc en conclure que l'augmentation du nombre de préemptions et migrations par travail est effectivement liée à l'augmentation du nombre de processeurs et non à l'augmentation du nombre de tâches. Les résultats obtenus par Nelissen *et al.* (voir figure 5.24) montrent que l'augmentation du nombre de préemptions et migrations faiblit très légèrement. Ceci est vraisemblablement dû à l'augmentation du nombre de tâches qui a pour effet de réduire le nombre de préemptions et migrations par travail.



(a) 30 tâches (b) 100 tâches

FIGURE 5.26 – Nombre de préemptions et migrations par travail en fonction du nombre de processeurs, avec un taux d'utilisation de 100%.

Enfin, notons que ces expérimentations sont faites pour un taux d'utilisation total de 100%, or l'un des avantages notable de EKG est la possibilité d'ajuster la valeur du paramètre K pour réduire de façon significative le nombre de préemptions et migrations au détriment de la limite d'utilisation (jusqu'à produire un ordonnancement de type P-EDF). De plus,

des améliorations possibles de EKG ont récemment été proposées mais n'ont pas été mises en œuvre ici [NFG12].

Pour les deux raisons présentées ci-dessus, l'avantage des politiques U-EDF et RUN sur la politique EKG n'est pas aussi important que ne le laisse paraître la figure 5.24. Mais, nos résultats montrent cependant que le gain de ces politiques en termes de préemptions et migrations par rapport à EKG est significatif dès lors que le nombre de processeurs est supérieur à 4.

Comparaison U-EDF et RUN

Dans la partie 5.3.1, nous avons observé que la quasi-totalité des systèmes pour lesquels le nombre de tâches divisé par le nombre de processeurs dépasse 2 et dont le taux d'utilisation total du système est inférieur à environ 90% sont partitionnables. Avec un nombre de tâches 5 fois supérieur au nombre de processeurs, la plupart des systèmes avec un taux d'utilisation proche de 97.5% sont partitionnables. En conséquence, l'algorithme RUN ne produit aucune migration dans la majorité des cas comme le montre la figure 5.27 qui donne le nombre de préemptions et migrations en fonction du taux d'utilisation total du système.

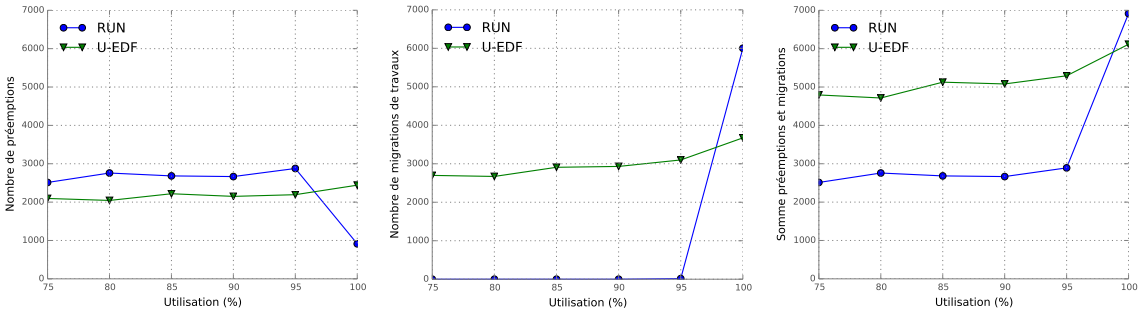


FIGURE 5.27 – Nombre de préemptions et migrations en fonction du taux d'utilisation total du système pour des configurations de 50 tâches et 8 processeurs.

Les résultats présentés à travers la figure 5.27 montrent que l'algorithme RUN engendre moins de préemptions et migrations que U-EDF lorsque les systèmes peuvent être partitionnés (dans ce cas, RUN se comporte comme P-EDF) mais pour un taux d'utilisation de 100%, U-EDF fait mieux que RUN. Ce résultat est généralisable à la majorité des configurations que nous avons testées à l'exception des cas où le nombre de processeurs dépasse 12. En effet, le nombre de préemptions et migrations engendrées par U-EDF augmente avec l'augmentation du nombre de processeurs plus rapidement que pour RUN. La figure 5.28 montre le nombre de préemptions et migrations en fonction du nombre de processeurs pour ces deux algorithmes pour des systèmes dont le taux d'utilisation est de 100% et avec 50 tâches.

Conclusion

Dans cette partie, nous avons réalisé des expérimentations sur les algorithmes EKG, U-EDF et RUN dans des conditions différentes de celles de Nelissen *et al.* Plus précisément, nos méthodes pour la génération des taux d'utilisation et des périodes sont différentes et n'utilisent pas les mêmes paramètres. L'implémentation des algorithmes a également été faite de manière totalement indépendante en nous basant uniquement sur la description des algorithmes telle qu'exposée dans les articles originaux. Nos résultats sont en accord avec ceux présentés par Nelissen *et al.* mais notre générateur de taux d'utilisation nous a également permis d'identifier plus précisément les facteurs de l'augmentation du nombre de préemptions et migrations.

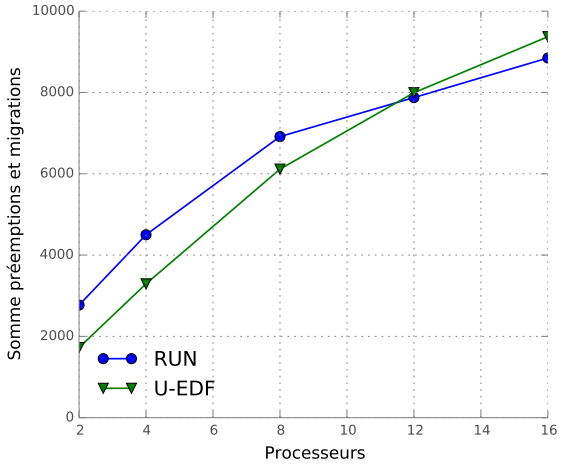


FIGURE 5.28 – Préemptions et migrations pour les algorithmes U-EDF et RUN en fonction du nombre de processeurs pour un taux d'utilisation total de 100% et pour 50 tâches.

Dans un premier temps, nous avons identifié les raisons expliquant les mauvais résultats obtenus pour EKG dans l'évaluation de Nelissen *et al.*. La première raison est liée au faible nombre de tâches par rapport au nombre de processeurs. Et la seconde est due à l'augmentation plus rapide, comparativement aux autres algorithmes, du nombre de préemptions et migrations en augmentant le nombre de processeurs. De plus, le paramètre K peut être ajusté pour réduire le nombre de préemptions et migrations pour des systèmes moins chargés, et des améliorations de EKG ont été proposées mais n'ont pas été mises en œuvre.

Dans un second temps, U-EDF et RUN ont été comparés. Il en ressort que RUN engendre un ordonnancement similaire à P-EDF dans la plupart des cas lorsque le taux d'utilisation du système est inférieur à 100%. Ceci rend RUN meilleur que U-EDF en matière de préemptions et migrations dans ces circonstances. Dans les cas où les systèmes ont un taux d'utilisation total de 100%, U-EDF engendre moins de préemptions et migrations que RUN, sauf si le nombre de processeurs dépasse environ 12. Dans ce cas, les performances de U-EDF pourraient être améliorées en le combinant à du clustering.

5.6 Usage du WCET

Dans la littérature, les études empiriques sont généralement conduites de sorte à ce que les tâches atteignent leur pire temps d'exécution (WCET) pour chaque travail. Ceci semble cependant très pessimiste comparé à la réalité pour deux raisons. La première est que le WCET est une borne supérieure qui n'est en pratique jamais atteinte. La seconde raison est qu'il est encore plus improbable que toutes les tâches consomment tout leur WCET au même moment.

Afin d'améliorer le nombre de systèmes ordonnançables, la plupart des politiques ont rajouté des évènements d'ordonnancement supplémentaires en se basant notamment sur le WCET des tâches. C'est par exemple le cas des politiques *fair* qui se basent énormément sur le WCET dans leurs décisions d'ordonnancement. Cependant, le WCET est difficile à évaluer précisément ce qui conduit souvent à des WCET très pessimistes. Pour des systèmes critiques, le WCET est à la base des preuves d'ordonnançabilité. Mais dans

le cas de systèmes temps réel souples, nous pouvons nous interroger sur la pertinence de baser nos décisions sur le WCET probablement très éloigné de la réalité. À titre d'exemple, Konig *et al.* expliquent que les applications de contrôle moteur ne sont pas ordonnançables d'après les WCET mais que pourtant ces systèmes sont fonctionnels en pratique [KBS+09].

Nous discutons dans cette partie des problématiques liées à l'utilisation du WCET pour évaluer les performances d'algorithmes d'ordonnancement comme cela a été fait dans ce chapitre. Et de manière plus générale, nous ouvrons la discussion sur le fait d'utiliser le WCET comme un paramètre important pour un algorithme d'ordonnancement. Cette problématique a fait l'objet d'un article publié dans la session *work-in-progress* de SIES 2014 [CHDD14].

5.6.1 Simulation d'ordonnancement avec durées aléatoires des travaux

L'utilisation du WCET des tâches correspond souvent au pire cas pour un système du point de vue de l'ordonnancement, bien que cela ne soit pas toujours vrai (voir partie e) sur les anomalies d'ordonnancement). Mais cela pourrait également induire une mauvaise évaluation des performances des politiques d'ordonnancement. Par exemple, certaines politiques sont naturellement robustes à un pic de charge ponctuel et peuvent s'adapter à cette demande. De plus, les évaluations expérimentales de politiques d'ordonnancement utilisant le WCET pourraient favoriser les politiques utilisant le WCET comme paramètre. Il semble donc raisonnable d'étudier les performances des politiques d'ordonnancement en se basant non seulement sur le WCET mais aussi sur des durées d'exécution aléatoires inférieures et non constantes.

Nous avons vu dans la section 4.2.4 que SimSo offre la possibilité de choisir la durée d'exécution des travaux à l'aide de modèles de temps d'exécution. L'un de ces modèles permet notamment d'utiliser une durée d'exécution aléatoire pour chaque travail. Nous utiliserons donc ici le modèle ACET qui, pour rappel, se base sur un tirage aléatoire selon une loi normale. Pour cela, une valeur moyenne et un écart type sont imposés par l'utilisateur (ici la moyenne est fixée à 75% du WCET et l'écart-type est égal à 10% du WCET). Ainsi, pour chaque travail, une durée d'exécution est déterminée aléatoirement puis bornée par le WCET.

5.6.2 Évaluation basée sur le WCET

Dans un premier temps, nous évaluons le nombre de préemptions et migrations des politiques G-EDF, RUN et U-EDF en utilisant le WCET des tâches puis en utilisant le modèle ACET. Les algorithmes RUN et U-EDF ont été choisis car nous venons de voir que ces algorithmes donnent de bons résultats et nous avons ainsi souhaité poursuivre l'évaluation. Enfin, G-EDF nous intéresse car c'est l'une des rares politiques d'ordonnancement à ne pas utiliser le WCET dans ses prises de décision.

Résultats

La figure 5.29 montre le nombre de préemptions et migrations en fonction du taux d'utilisation du système. L'axe des abscisses du bas correspond au taux d'utilisation des systèmes utilisant le WCET tandis que l'axe supérieur correspond à leurs taux d'utilisation lorsque le tirage ACET est appliqué. À partir de ces résultats, nous faisons plusieurs observations ci-dessous.

Concernant G-EDF, nous pouvons observer que les résultats obtenus avec un taux d'utilisation de 75% avec le WCET sont très proches de ceux obtenus avec un taux d'utilisation de 100% avec l'ACET (soit une utilisation effective de 75%). Par ailleurs, le nombre de préemptions et migrations croît linéairement dans les deux cas, mais avec une pente légèrement supérieure dans le cas de l'utilisation du WCET. L'algorithme G-EDF engendre moins de préemptions et migrations que RUN et U-EDF lorsque le modèle ACET est utilisé alors que RUN est dans certains cas meilleur lorsque le WCET est utilisé.

En considérant les simulations utilisant le WCET et en excluant le cas où l'utilisation totale est de 100%, nous voyons que RUN fait environ 30% de préemptions et migrations en moins que U-EDF (voir partie 5.5 pour plus de détails). Cependant, lorsque le modèle ACET est utilisé, ces deux politiques donnent des résultats très similaires. Dans le cas où l'utilisation du système est de 100%, le nombre de préemptions et migration pour RUN double (résultat détaillé dans la partie 5.5) et U-EDF donne alors de bien meilleurs résultats, en particulier avec le modèle ACET.

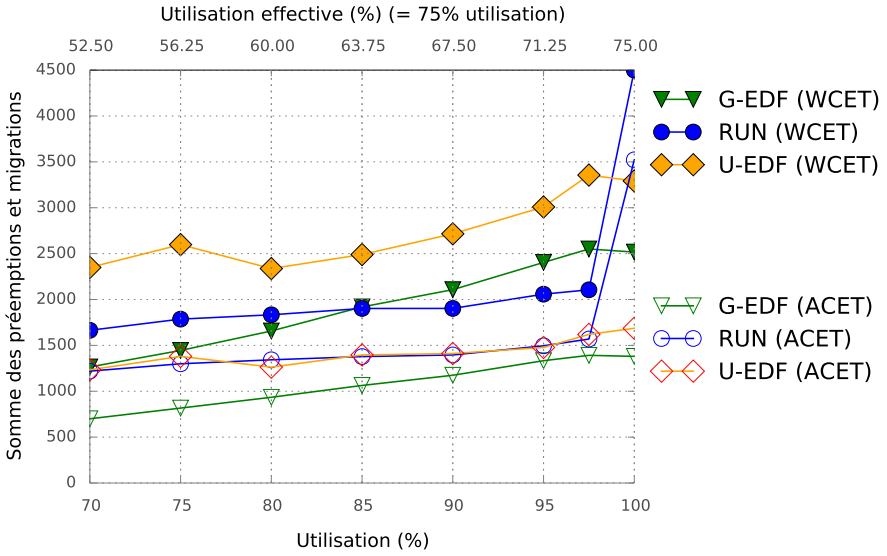


FIGURE 5.29 – Nombre de préemptions et migrations pour un système composé par 50 tâches sur 4 processeurs, en utilisant les ordonnanceurs G-EDF, RUN et U-EDF.

Conclusion

D'un point de vue algorithmique, G-EDF ne base pas ses décisions d'ordonnancement sur le WCET. Ainsi, la réduction des durées d'exécution apporte un gain immédiat sur le nombre de préemptions et migrations. L'algorithme U-EDF utilise pour sa part le WCET mais là encore, l'utilisation de durées inférieures engendre une réduction importante du nombre de préemptions et migrations. En revanche, le gain pour RUN est beaucoup plus faible. Mais notons cependant que des améliorations pour permettre à RUN de tirer profit de durées d'exécution plus courtes ont été proposées récemment [CMV14] mais n'ont pas encore été mises en œuvre dans SimSo.

En conclusion, le nombre de préemptions et migrations baisse en utilisant des durées aléatoires inférieures au WCET des tâches, cependant cette baisse varie de manière significative d'un algorithme à l'autre. Ainsi, les évaluations empiriques du nombre de préemptions et migrations qui se basent uniquement sur le WCET ne permettent pas d'apprécier la capacité des algorithmes à réduire le nombre de préemptions et migrations lorsque la charge du système est inférieure à l'utilisation dans le pire cas.

5.6.3 Ordonnancement d'un système en surcharge

Nous souhaitons ici vérifier le constat de Konig *et al.* selon lequel, de nombreux systèmes qui ne sont pas théoriquement ordonnançables sont en pratique fonctionnels [KBS+09]. Pour cela, nous avons décidé d'évaluer les politiques G-EDF, EDF-US, PriD, EDZL, G-FL et G-MLLF en utilisant le modèle ACET sur des systèmes dont le taux total d'utilisation varie de 70 à 130%. La durée moyenne d'exécution des travaux est fixée à 75% du WCET des tâches, ainsi le taux d'utilisation effectif est de l'ordre de 97.5% mais il arrive des cas où ponctuellement le système doit faire face à une surcharge.

Résultats

La figure 5.30 montre les résultats obtenus en moyenne pour un nombre de tâches et de processeurs variant sur tout leur domaine de définition. Les résultats indiquent que pour les systèmes simulés avec un taux d'utilisation de 130%, les politiques G-EDF, PriD et EDF-US sont capables d'en ordonnancer la moitié et les algorithmes G-FL, EDZL et G-MLLF sont capables d'ordonnancer environ 90% d'entre eux. De plus, les observations faites dans la partie 5.2, à savoir que les algorithmes donnent de meilleurs résultats avec peu de processeurs et beaucoup de tâches, restent valables ici.

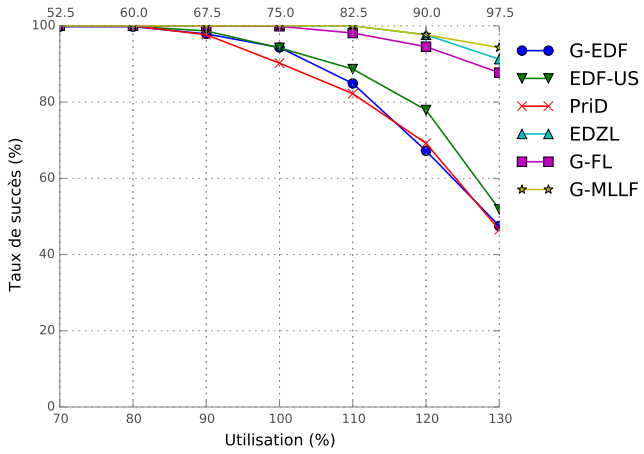


FIGURE 5.30 – Taux de succès en fonction de la charge avec le modèle ACET pour un nombre de tâches dans {20, 30, 40, 50, 60, 70, 80, 90, 100} et un nombre de processeurs dans {2, 4, 8, 12, 16}.

Enfin, pour les systèmes qui ont provoqué au moins un dépassement d'échéance, nous constatons que le nombre de dépassements reste faible. La figure 5.31 montre le ratio de travaux avortés suite à un dépassement d'échéance par rapport au nombre total de travaux pour un système composé de 50 tâches et 8 processeurs avec un taux d'utilisation total de 130%. Sur la figure nous utilisons une représentation standard à base de boites à moustaches [8]. Nous constatons que le taux d'échecs est extrêmement faible, en particulier pour EDZL, G-FL et G-MLLF.

Conclusion

Nous pouvons donc en conclure que des politiques telles que G-EDF sont capables de s'adapter à des surcharges temporaires. De plus, le nombre d'échecs reste relativement

8. Une boite à moustaches permet une représentation statistique des données. Chaque boîte s'étend du premier quartile au dernier quartile des valeurs, le trait correspond à la médiane et les « moustaches » sont situées à une distance de 1.5 interquartiles des bords de la boite.

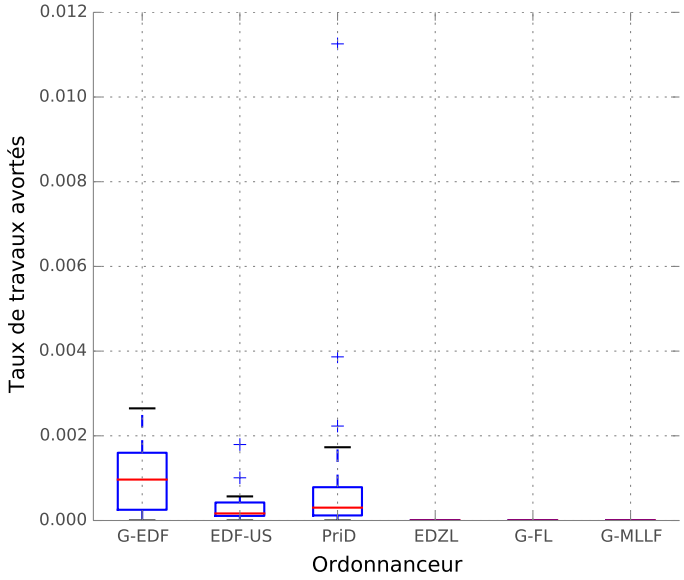


FIGURE 5.31 – Ratio de travaux avortés suite à un dépassement d'échéance pour un taux d'utilisation total de 130% avec 50 tâches et 8 processeurs avec le modèle ACET.

faible ce qui peut être acceptable pour des systèmes moins critiques. Des politiques telles que EKG, U-EDF, RUN, DP-WRAP ou LRE-TL ne permettent pas l'ordonnancement de systèmes dont le taux d'utilisation dépasse les 100% pour des raisons algorithmiques. Par conséquent, des politiques simples de type G-EDF et ses variantes donnent certes en théorie des limites d'utilisation plus faibles que les politiques sus-citées mais en pratique elles sont aussi capables d'ordonnancer des systèmes que ces politiques optimales ne peuvent pas gérer. De manière liée, une erreur dans l'estimation du WCET pourrait avoir de graves conséquences avec certaines politiques alors que le problème resterait minime avec des politiques telles que G-EDF et ses variantes.

5.7 Avantages liés aux politiques conservatives

Dans cette partie, nous nous intéressons aux gains apportés par des variantes *work-conserving* de certaines politiques.

Dans un premier temps, nous considérerons les politiques PFair et DP-Fair qui sont basées sur la notion d'équité. Autrement dit, un temps proportionnel au taux d'utilisation est alloué pour chaque travail. Leurs variantes *work-conserving* permettent à des tâches de prendre de l'avance sur leur exécution lorsqu'un processeur devient libre. Nous verrons que ceci a un impact très positif sur le nombre de préemptions et migrations mais aussi les temps de réponse. Les évaluations se feront en utilisant le modèle WCET puis avec le modèle ACET puisque ces algorithmes sont susceptibles de tirer avantage de durées d'exécution plus courtes que prévu.

Suite aux résultats obtenus pour ER-PD2 et NVNLF, nous avons voulu savoir s'il était possible de réduire le nombre de préemptions et migrations engendrées par les politiques RUN et U-EDF. Ces politiques ne sont pas conservatives, il existe donc des périodes où au moins un processeur est inactif alors que des tâches prêtes sont en attente. Nous proposons

un moyen simple pour combiner ces politiques à G-EDF pour exécuter les travaux en attente sur les processeurs laissés inutilisés.

5.7.1 Ordonnancement ER-Fair

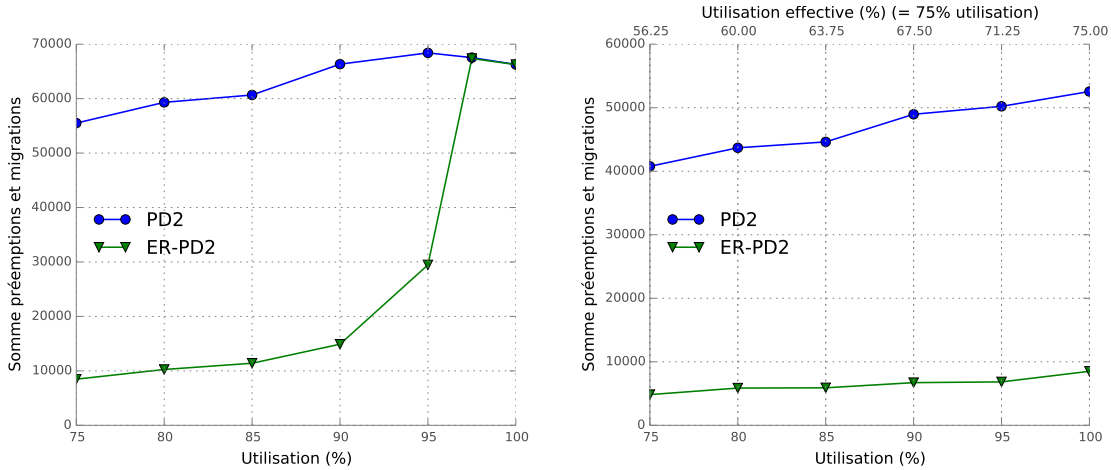
Les politiques ER-Fair sont des politiques PFair pour lesquelles la contrainte d'équité a été en partie relâchée pour autoriser une tâche à avoir de l'avance (voir chapitre 1). Pour des systèmes dont le taux d'utilisation est inférieur à 100%, ceci permet aux tâches de profiter des périodes d'inactivité des processeurs pour prendre un peu d'avance sur leur exécution. Ceci a un impact important sur le nombre de préemptions puisque les travaux ne sont plus interrompus pour la simple raison qu'ils allaient prendre de l'avance. Ceci est illustré par la figure 5.32. Nous évaluons ici les algorithmes PD2 et ER-PD2 en matière de préemptions et migrations puis en matière de laxité.

(a) Ordonnancement PD2. (b) Ordonnancement ER-PD2.

FIGURE 5.32 – Ordonnancement selon les politiques PD2 et ER-PD2 d'une tâche ayant pour durée d'exécution 3 et une échéance et période de 6. Le quantum utilisé ici est de 1.

Préemptions et migrations

La figure 5.33 présente le nombre de préemptions et migrations en fonction du taux d'utilisation des systèmes en utilisant les modèles WCET et ACET. En présence de périodes d'inactivité des processeurs, nous constatons que ER-PD2 permet effectivement de réduire de manière significative le nombre de préemptions. Ce résultat se retrouve pour les autres configurations testées avec un gain très faible lorsqu'il y a beaucoup de tâches et peu de processeurs et au contraire une réduction du nombre de préemptions et migrations de près de 50 fois pour 16 processeurs et 20 tâches (taux d'utilisation de 100% et utilisation du modèle ACET).

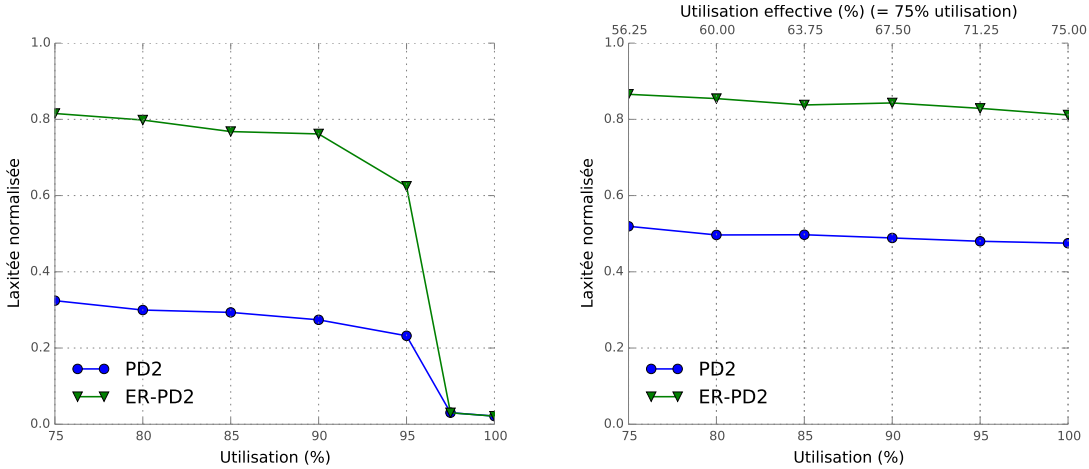


(a) Avec WCET. (b) Avec ACET.

FIGURE 5.33 – Somme du nombre de préemptions et migrations en fonction du taux d'utilisation pour un système composé de 50 tâches et 8 processeurs.

Laxité normalisée

Naturellement, les temps de réponse sont également largement améliorés. La figure 5.34 montre la laxité normalisée en fonction du taux d'utilisation avec les modèles WCET et ACET.



(a) Avec WCET.

(b) Avec ACET.

FIGURE 5.34 – Laxité normalisée en fonction du taux d'utilisation pour un système composé de 50 tâches et 8 processeurs.

Enfin, notons que les gains dépendent aussi du nombre de tâches et de processeurs comme c'est le cas pour les préemptions et migrations. Ainsi, plus il y a de tâches et moins il y a de différences entre les deux politiques. Et au contraire, plus il y a de processeurs et plus le gain est important.

Conclusion

La variante work-conserving de PD2 n'altère pas les conditions d'ordonnançabilité et ne complexifie pas l'algorithme. Expérimentalement, nous observons un gain parfois très important à la fois sur le nombre de préemptions et migrations mais aussi sur la laxité normalisée (et donc sur les temps de réponse). Rappelons qu'une plus grande laxité limite les risques de dépassement d'échéance en cas d'erreur d'estimation du WCET. Les gains augmentent avec l'augmentation du nombre de processeurs et la diminution du nombre de tâches. De plus, lorsque nous simulons en utilisant le WCET des tâches et que le taux d'utilisation est supérieur à environ 95%, ER-PD2 ne fait pas mieux que PD2. Un dernier élément a une importance forte dans la différence entre ces deux politiques : le choix du quantum. En effet, si on regarde la figure 5.32, on comprend aisément qu'avec un quantum 10 fois plus petit, nous aurions pour cet exemple 10 fois plus de préemptions.

5.7.2 Algorithme d'ordonnancement NVNLF

NVNLF se présente comme une variante *work-conserving* de LLREF [FKY08]. Mais comme LRE-TL qui a été publié plus tard, NVNLF se dispense de trier les tâches. Afin de nous concentrer sur l'aspect *work-conserving*, les résultats seront donc confrontés à ceux de LRE-TL. Les auteurs annoncent une réduction importante du nombre de préemptions et migrations. Son fonctionnement est brièvement décrit dans la section 1.5.2. En théorie, NVNLF doit permettre d'éviter l'interruption trop hâtive de travaux en cours d'exécution

en leur permettant de s'exécuter plus longtemps. Et ceci sans augmenter le nombre de préemptions ni pénaliser les autres travaux. De plus, les travaux ayant bénéficié de temps supplémentaire se terminent plus vite ce qui offre alors plus de temps pour les autres tâches, réduisant alors le nombre de préemptions et les temps de réponse.

Les auteurs comparent E-TNPA (NVNLF) à TNPA (assimilé LRE-TL), EKG et EDZL sur une architecture comprenant 16 processeurs. Le nombre de tâches varie en fonction de l'utilisation totale du système de par leur méthode de génération de tâches. Les résultats obtenus par les auteurs sont rappelés par la figure 5.35.

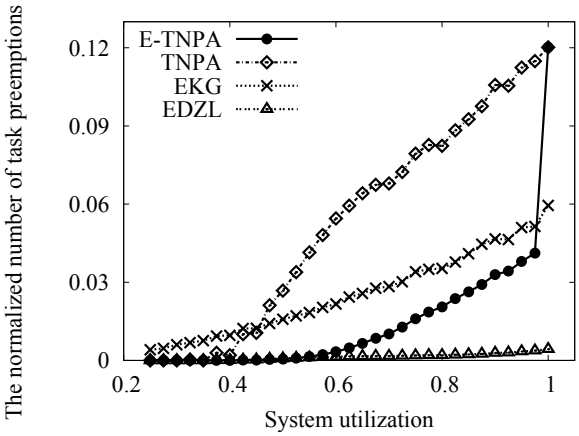


FIGURE 5.35 – Nombre de préemptions (incluant les migrations ici) en fonction du taux d'utilisation du système [FKY08].

À partir de nos données de simulation, nous sommes en mesure de reproduire une évaluation similaire. Cependant, nous n'utilisons pas la même méthode de génération de tâches ce qui nous permet d'isoler l'influence de l'augmentation de l'utilisation de celle de l'augmentation du nombre de tâches.

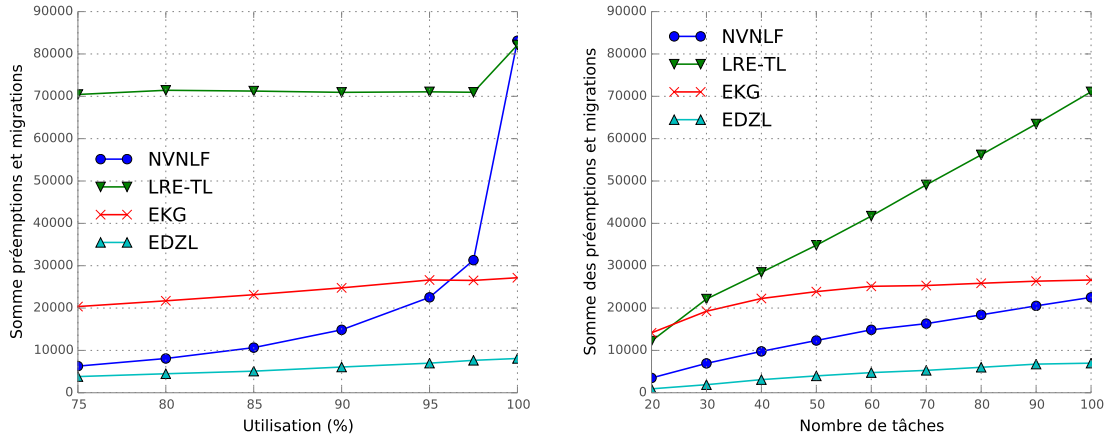
Préemptions et migrations

La figure 5.36 montre le nombre de préemptions et migrations en fonction de l'utilisation totale du système et du nombre de tâches. Ce type de courbe est représentatif du comportement observé sur les autres configurations testées.

Le nombre de préemptions pour LRE-TL ne semble pas dépendre fortement du taux d'utilisation du système jusqu'à ce qu'il devienne très chargé. NVNLF profite des temps d'inactivité que LRE-TL laisserait aux processeurs pour réduire le nombre de préemptions et migrations. Par conséquent, nous voyons comme attendu que le nombre de préemptions tend à converger vers celui de LRE-TL lorsque l'utilisation devient importante.

Tout comme LRE-TL, le nombre de préemptions pour NVNLF augmente avec l'augmentation du nombre de tâches. À l'aide des données supplémentaires concernant des simulations avec moins de processeurs, nous pouvons constater que les résultats ne dépendent pas du nombre de processeurs.

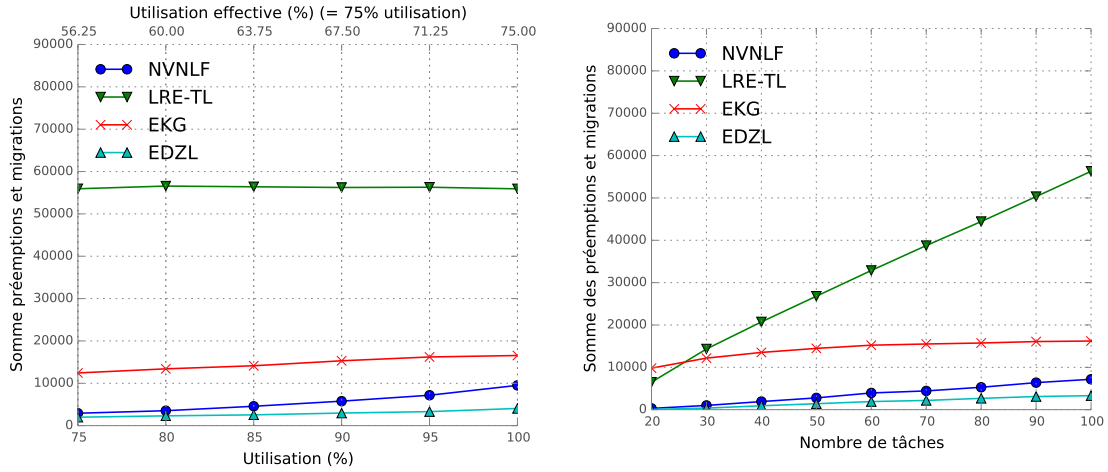
La figure 5.37 donne les résultats obtenus pour les mêmes systèmes mais en utilisant cette fois le modèle ACET. Nous constatons que NVNLF est encore plus avantageux lorsque la durée d'exécution des tâches est aléatoire et inférieure au WCET ce qui provoque l'apparition de périodes d'inactivité plus longues que NVNLF peut exploiter.



(a) En fonction de l'utilisation totale du système avec 100 tâches.

(b) En fonction du nombre de tâches pour un taux d'utilisation du système de 95%.

FIGURE 5.36 – Somme des préemptions et migrations en fonction de l'utilisation du système et du nombre de tâches. Le système est composé de 16 processeurs.



(a) En fonction de l'utilisation totale du système avec 100 tâches.

(b) En fonction du nombre de tâches pour un taux d'utilisation du système de 95% (d'après le WCET des tâches, 71.25% effectif).

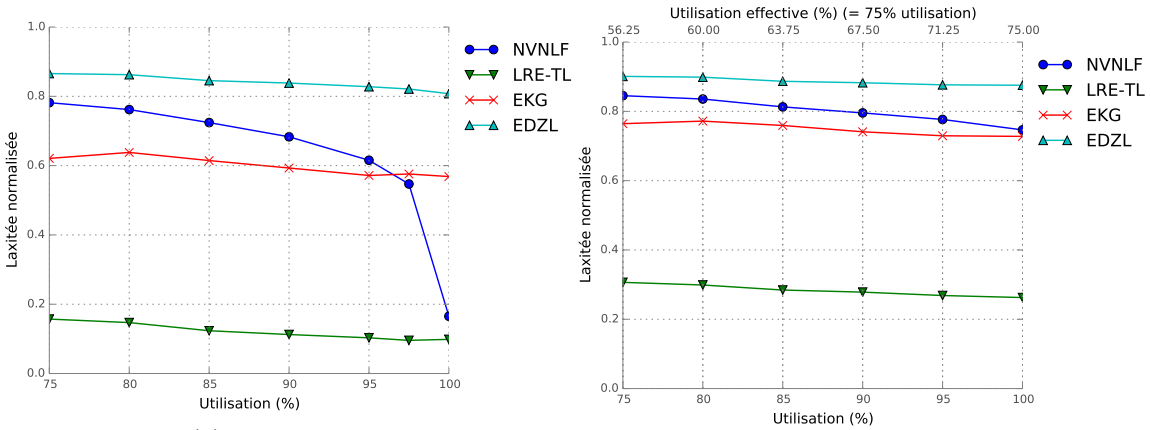
FIGURE 5.37 – Nombre de préemptions et migrations en fonction de l'utilisation du système et du nombre de tâches en utilisant le modèle ACET. Le système est composé de 16 processeurs.

Laxité normalisée

La figure 5.38 montre une réduction importante du temps de réponse des travaux lorsque le système n'est pas chargé à 100%. Lors de l'utilisation du modèle ACET, NVNLF se place entre EDZL et EKG en matière de laxité.

Conclusion

L'algorithme NVNLF permet aux tâches de prendre de l'avance par rapport à leur exécution fluide tout comme ER-PD2 le permet. Nos expérimentations confirment cette propriété. Ainsi, NVNLF sera plus efficace sur des systèmes avec relativement peu de tâches et une utilisation totale inférieure à 90-95%. Pour autant, NVNLF engendre toujours un



(a) Avec WCET.

(b) Avec ACET.

FIGURE 5.38 – Laxité normalisée en fonction du taux d'utilisation du système. Le système comprend 100 tâches et 16 processeurs.

nombre important de préemptions et migrations en comparaison de EDZL.

En utilisant le modèle ACET, il existe des périodes d'inactivité que NVNLF sait utiliser. Ainsi, sur un système réel où les travaux s'exécutent en pratique pendant une durée inférieure au WCET, les résultats sont encore plus marqués.

5.7.3 Combiner G-EDF à des politiques non conservatives

Les résultats obtenus avec NVNLF et ER-Fair laissent entendre que rendre des politiques *work-conserving* permet de réduire de façon importante le nombre de préemptions et migrations en plus d'améliorer les temps de réponse. Nous pourrions donc légitimement espérer des améliorations substantielles en rendant les politiques RUN et U-EDF *work-conserving* ou du moins en évitant autant que possible de laisser des processeurs inactifs lorsqu'il existe un travail prêt en attente[9]. Au lieu de chercher à améliorer le cœur des algorithmes RUN et U-EDF, nous proposons de combiner simplement G-EDF à ces politiques de manière simple et générique (qui pourrait s'appliquer à toute politique). Puis nous évaluons les gains apportés par cette technique.

Principe

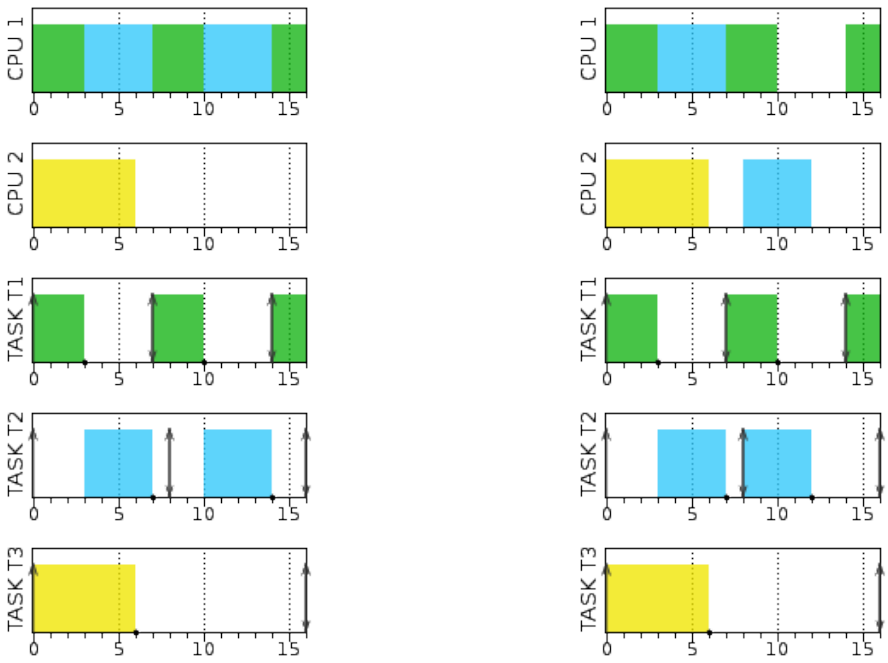
Nous souhaitons éviter le cas où un travail pouvant s'exécuter reste en attente alors qu'un processeur est disponible. Cette situation ne peut se produire que suite à certains évènements précis :

- Fin d'exécution d'un travail (un processeur devient inactif) ;
- Activation d'un nouveau travail (nouveau travail prêt en attente) ;
- Décision d'ordonnancement qui laisse un processeur inactif.

Lors de ces évènements, nous souhaitons faire qu'un processeur n'est pas laissé inactif si un travail peut s'exécuter dessus. Ainsi, lors de ces trois évènements, nous pouvons

9. Il existe de nombreux exemples qui mettent en évidence que ces algorithmes ne sont pas *work-conserving*. L'exemple le plus simple est l'ordonnancement de deux tâches identiques et dont le taux total d'utilisation est inférieur à 1, sur deux processeurs.

ordonnancer les tâches prêtes en attente selon G-EDF sur les processeurs inactifs. La figure 5.39 illustre ceci à travers l'exemple de l'ordonnancement de trois tâches sur deux processeurs.



(a) Ordonnancement U-EDF. (b) Ordonnancement U-EDF couplé à G-EDF.

FIGURE 5.39 – Différence entre un ordonnancement U-EDF et sa variante *work-conserving*.

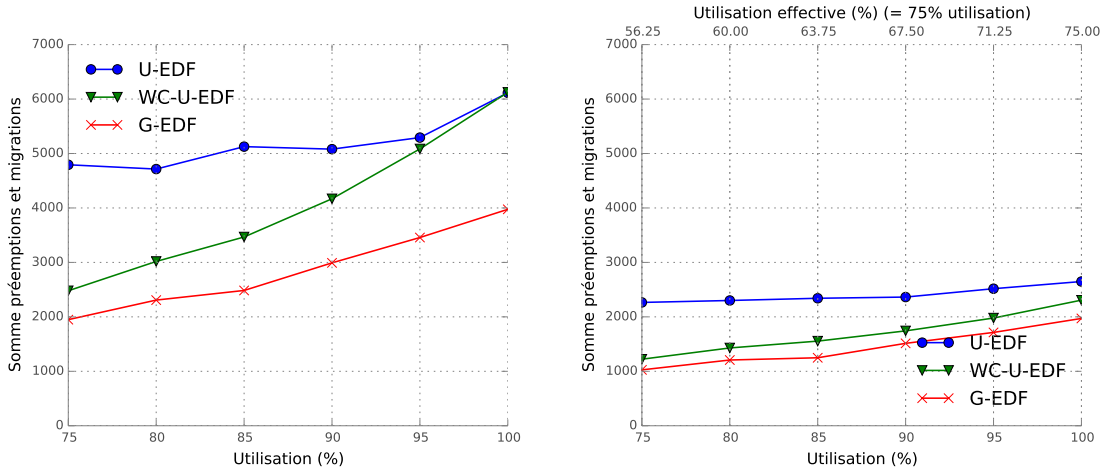
Malheureusement, ceci n'a pas eu les effets attendus en matière de préemptions et migrations. En effet, ceci a provoqué de nombreuses situations d'exécution très brèves de travaux. Autrement dit, au lieu de laisser des processeurs inactifs de très courts instants, des travaux se sont exécutés brièvement avant d'être préemptés. Dans le cas de l'utilisation du modèle ACET, cette situation est encore plus fréquente. Dans le cas de ER-PD2 et NVNLF, la réduction est surtout liée à la suppression d'évènements inutiles et générateurs de préemptions.

Nous avons donc décidé de ne pas appliquer G-EDF lors des évènements de type fin d'exécution à l'exception des cas où cela conduit à une prise de décision d'ordonnancement de l'algorithme initial.

Résultats

Cette stratégie a été appliquée aux politiques U-EDF et RUN. Nous appellerons WC-U-EDF et WC-RUN leurs variantes utilisant G-EDF pour ordonnancer les tâches restantes Pour ces deux politiques, les temps de réponse sont sensiblement améliorés.

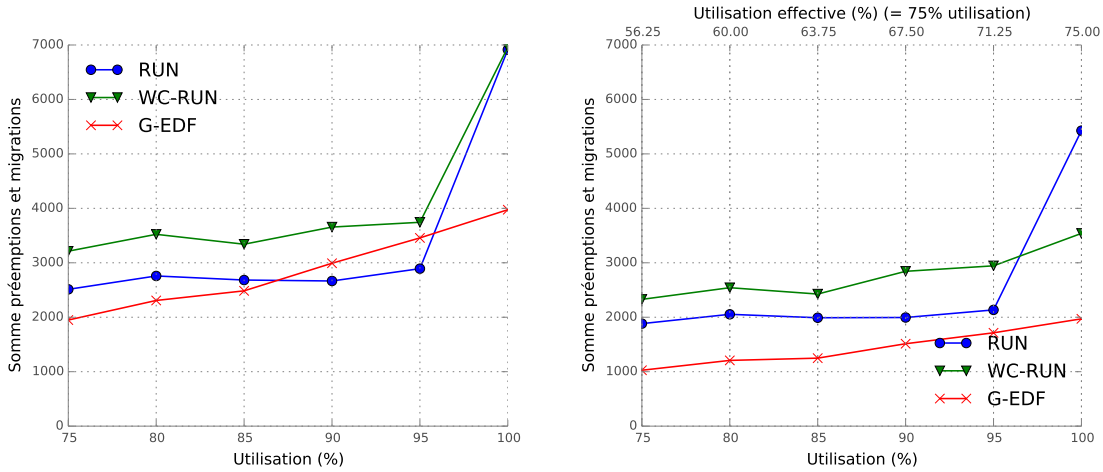
La figure 5.40 montre le nombre de préemptions et migrations pour U-EDF, WC-U-EDF et G-EDF en fonction de l'utilisation du système et en utilisant les modèles WCET et ACET. Nous pouvons voir que le nombre de préemptions et migrations est divisé par 2 dans cette configuration lorsque le taux d'utilisation du système est de 75% et ce pour les deux modèles (WCET et ACET). Logiquement, ce gain diminue avec l'augmentation du taux d'utilisation jusqu'à devenir identique pour des systèmes où il n'y a aucune période d'inactivité des processeurs. Les résultats obtenus pour WC-U-EDF sont relativement proches de ceux de G-EDF lors de l'utilisation du modèle ACET.



(a) Avec WCET. (b) Avec ACET.

FIGURE 5.40 – Somme des préemptions et migrations en fonction de l'utilisation du système. Le système est composé de 50 tâches et 8 processeurs.

Concernant RUN, les performances ne sont pas améliorées comme le montre la figure 5.41. En effet, de nombreuses migrations sont ajoutées par ce mécanisme qui ne sont pas toujours compensées par la réduction des préemptions.



(a) Avec WCET. (b) Avec ACET.

FIGURE 5.41 – Somme des préemptions et migrations en fonction de l'utilisation du système. Le système est composé de 50 tâches et 8 processeurs.

Conclusion

À travers cette expérimentation, nous avons souhaité montrer qu'il était simple d'utiliser SimSo pour mettre en œuvre et évaluer simplement des variantes. Avant d'investir du temps dans une idée, il est ainsi possible de vérifier expérimentalement que les résultats sont en accord avec notre intuition.

Au sujet de la technique que nous venons de présenter, nous avons vu que rendre une politique *work-conserving* ou du moins essayer d'utiliser plus souvent les processeurs inactifs, n'engendre pas nécessairement une réduction du nombre de préemptions et migrations. Nous n'avons pas amélioré RUN et l'amélioration de U-EDF s'est faite en n'agissant qu'au

moment des prises de décision de U-EDF. Mais les résultats obtenus montrent une réduction significative du nombre de préemptions et migrations.

Les résultats concernant U-EDF ont été partagés avec G. Nelissen, l'auteur principal de cet algorithme. Cet échange a permis d'évoquer une autre proposition qui consisterait à utiliser la durée restante d'exécution (le WCET moins la durée exécutée) à la place d'une stratégie de type EDF (basée sur l'échéance la plus proche) pour choisir les travaux à exécuter. Selon lui, cette proposition aurait pour avantage de réduire le nombre de travaux prêts à chaque instant en favorisant leur terminaison au plus vite. Cependant, aucune mise en œuvre n'a été faite.

5.8 Conclusion

Au cours de ce chapitre, un ensemble d'observations ont pu être établies à partir des simulations menées avec SimSo. À travers ces évaluations, nous avons montré qu'il était possible d'utiliser SimSo pour conduire une large campagne d'évaluation des politiques. De plus, toutes ces expérimentations ont été réalisées dans des conditions identiques ce qui autorise les comparaisons entre les algorithmes d'ordonnancement.

Tous les résultats obtenus n'ont cependant pas pu être détaillés pour des raisons évidentes de place. Nous avons donc sélectionné ceux qui illustrent le mieux les tendances générales qui se dégagent. Afin de faciliter la lecture, les résultats ont été regroupés par thème. Notons que ces résultats supposent des coûts système nuls et se basent sur des modèles de durée d'exécution simples.

Algorithmes de type G-EDF

L'algorithme d'ordonnancement G-EDF donne en pratique des résultats intéressants bien que les tests d'ordonnançabilité ne permettent pas de prouver l'ordonnançabilité de nombreux systèmes. Ses variantes améliorent le taux de systèmes ordonnançables sans pour autant nuire aux performances de l'ordonnancement en termes de préemptions, migrations et temps de réponse. Notons par exemple les bons résultats de G-FL qui ne nécessite qu'un changement trivial de G-EDF au niveau du calcul de la priorité des travaux. Ou encore G-MLLF qui est capable d'ordonnancer un grand nombre de systèmes. G-MLLF provoque cependant une augmentation du nombre de préemptions et migrations mais ce nombre reste raisonnable lorsque le nombre de tâches est faible ou le nombre de processeurs important.

Ordonnancement partitionné

Il a été montré que l'algorithme d'affectation *Decreasing First-Fit* permet de partitionner plus d'ensembles de tâches que *Decreasing Worst-Fit*, cependant, ce dernier provoque moins de préemptions pour des systèmes modérément chargés. Par ailleurs, le nombre de systèmes ordonnançables est important, en particulier si le nombre de tâches est grand. Par exemple, quasiment tous les systèmes dont le taux d'utilisation est inférieur à 95% ont pu être correctement ordonnancés avec au moins 50 tâches. Les algorithmes P-EDF et G-EDF n'ont pas pu être clairement départagés du point de vue du nombre de préemptions et migrations et des temps de réponse, mais P-EDF semble être en mesure d'ordonnancer un plus grand nombre de systèmes, en particulier si la charge est faible ou lorsqu'il y a de nombreux processeurs. Cependant, l'algorithme G-FL a pu exécuter sans dépassement d'échéance un plus grand nombre de systèmes que P-EDF, en particulier pour une charge

forte. Le clustering n'a pas été étudié ici mais pourrait être un compromis intéressant puisque le taux de systèmes ordonnançables par les politiques de type G-EDF se dégrade avec l'augmentation du nombre de processeurs.

Politiques DP-Fair

LLREF engendre un nombre de préemptions plus important que nécessaire en effectuant un tri inutile des travaux à exécuter. Ce problème a été corrigé dans les politiques LRE-TL et DP-WRAP ce qui leur permet de réduire le nombre de préemptions. Les auteurs de LRE-TL annoncent une réduction du nombre de préemptions et migrations par rapport à LLREF de l'ordre du nombre de tâches. Cependant, nos résultats indiquent que la réduction est proche du nombre de processeurs et que le nombre de tâches n'influe pas sur les résultats. Les politiques LRE-TL et DP-WRAP utilisent des algorithmes pour la distribution temporelle des dotations très différentes et pourtant elles produisent des nombres de préemptions et migrations très proches. Ces politiques génèrent cependant un nombre important de préemptions et migrations en comparaison des politiques de type G-EDF ou P-EDF.

Comparaison U-EDF, RUN et EKG

Les résultats obtenus par Nelissen *et al.* pour illustrer les bonnes performances de leur algorithme U-EDF en le comparant à RUN et EKG ont été analysés. Les résultats que nous obtenons ne contredisent pas ceux présentés par Nelissen *et al.* mais les conditions expérimentales retenues correspondent à des configurations où EKG se comporte mal. Nous avons donc élargi les configurations testées et montré que les performances des algorithmes U-EDF et RUN sont très bonnes comparativement à celles de EKG, dans le cas d'un système chargé à 100%.

De manière plus globale, en regroupant les résultats issus des autres évaluations, les politiques U-EDF et RUN engendrent le moins de préemptions et migrations et donnent de bons temps de réponse, parmi les politiques théoriquement optimales. Les politiques de type G-EDF ainsi que P-EDF sont cependant meilleures sur ces critères mais ne permettent pas d'ordonnancer des systèmes aussi chargés.

Utilisation du WCET

Nous avons discuté de la pertinence de l'utilisation du WCET dans les évaluations concernant le nombre de préemptions, migrations et temps de réponse. En effet, des politiques comme G-EDF et U-EDF sont capables de tirer profit de durées d'exécution plus faibles que prévues contrairement à des politiques qui se basent beaucoup sur le WCET comme RUN. Nous avons également vu que certaines politiques sont capables d'ordonnancer des systèmes pour lesquels la charge totale dépasse les capacités du système. En pratique, cela est possible car il est peu probable que les travaux utilisent tout leur WCET en même temps.

Politiques conservatives

Les politiques *work-conserving* ont montré qu'elles permettent de réduire de façon significative le nombre de préemptions et migrations pour des systèmes qui ne sont pas chargés à 100%. De plus, dans la pratique, ces politiques sont capables de tirer profit de durées d'exécution inférieures au WCET ce qui augmente d'autant plus leur attrait. Un travail pour utiliser au mieux les temps d'inactivité des processeurs est nécessaire pour améliorer les performances des politiques RUN et U-EDF.

Troisième partie

Simulation avec impact des caches

CHAPITRE 6

Modèles de cache

Ce chapitre présente un ensemble de modèles statistiques pour l'évaluation de la durée d'exécution des tâches tout en tenant compte des caches. Le besoin d'estimer rapidement les défauts de cache engendrés par les programmes est né des travaux visant à améliorer les performances des programmes ou de mieux dimensionner les systèmes. Les modèles présentés ici ne sont donc pas issus du domaine temps réel et leur objectif est d'approximer le nombre réel de défauts de cache. Dans le domaine temps réel, les caches peuvent être pris en compte dans l'évaluation du WCET et donc dans une optique d'estimation du pire cas. L'information produite par ces modèles est donc complémentaire : l'estimation du WCET est indispensable pour prouver l'ordonnançabilité d'un système temps réel dur alors que l'estimation de la durée en tenant compte des caches va nous permettre de simuler un système en nous rapprochant d'un comportement réel. Les modèles qui sont présentés dans ce chapitre reposent sur une partie des métriques qui ont été présentées dans la partie 2.5.

Nous étudions ici la possibilité d'utiliser ces modèles dans la simulation telle que nous l'abordons dans ce travail de thèse afin de simuler la durée d'exécution des travaux. Ce chapitre se divise ainsi en trois parties. Dans un premier temps, nous rappelons les données d'entrée que nous avons à notre disposition, ce que l'on veut obtenir et nous formulons quelques hypothèses. Ensuite, nous présentons les modèles qui ont retenu notre attention et nous expliquons comment nous pourrions les utiliser. Dans la troisième partie de ce chapitre, nous effectuons un certain nombre d'expérimentations pour étudier le comportement des programmes vis-à-vis des caches et évaluer les modèles retenus.

6.1 Contexte

Les modèles que nous présentons et évaluons dans ce chapitre doivent nous permettre de simuler de manière réaliste le comportement des tâches lorsque les caches sont pris en compte. Le terme « réaliste » signifie seulement que nous souhaitons produire lors de la simulation des comportements cohérents avec ce que l'on peut observer lors de l'exécution réelle de programmes. En effet, il est très important de comprendre que notre objectif n'est pas de reproduire de manière fidèle le comportement d'un programme en particulier mais d'étudier de manière statistique le comportement d'ordonnanceurs en les soumettant à des scénarios les plus proches possibles d'exécutions réelles et de dégager ensuite des tendances générales.

Afin de juger de la pertinence des modèles destinés à abstraire l'influence des mémoires caches sur l'exécution des tâches et d'évaluer leur degré de précision, nous les confrontons ici à des exécutions réelles. En effet, plus ces modèles sont précis et plus ils permettront de reproduire les différents types de comportement observés.

6.1.1 Besoin pour la simulation

Le composant de SimSo particulièrement concerné par la prise en compte de l'influence des caches est le modèle de temps d'exécution [1] (ou ETM présenté en 4.2.4). En effet, il doit être en mesure de déterminer, pour un travail :

- le nombre d'instructions exécutées sur un intervalle de temps donné ;
- la durée d'exécution restante d'un travail à tout moment.

Ces grandeurs peuvent être déterminées simplement si l'on dispose de la connaissance du CPI des tâches (Cycles Par Instruction, voir section 2.5). Dans la section 6.3, nous montrerons comment évaluer ce CPI.

6.1.2 Entrées disponibles

Pour évaluer le CPI des tâches, les modèles présentés ci-après utilisent les métriques suivantes (voir leur présentation détaillée dans la section 2.5) :

- le nombre d'accès moyen à la mémoire par instruction (API) ;
- le nombre d'instructions (n) ;
- le *Stack Distance Profile* (SDP) ;
- le CPI sans prendre en compte les pénalités liés aux accès mémoire (voir section 6.2.1 pour son calcul) ;
- les caractéristiques des caches (taille, associativité, pénalités) et leur hiérarchie.

Toutes ces métriques sont supposées connues et fixées. Nous reviendrons dans la partie 6.3 sur les outils pour les obtenir ainsi que sur leur analyse.

Nous rappelons que le calcul du SDP d'un programme prend en compte le nombre d'ensembles présents dans le cache [2]. Par conséquent, le SDP d'une tâche dépend de l'architecture ciblée (taille du cache et associativité). Cela impose de disposer de plusieurs profils de SDP pour une même tâche si nous souhaitons faire varier les paramètres de l'architecture. Cela n'est malheureusement pas souhaitable pour mener de larges expérimentations.

Ce problème peut être contourné soit en supposant un cache entièrement associatif, soit en synthétisant le SDP pour un certain nombre d'ensembles à partir du SDP calculé pour un seul ensemble (cache entièrement associatif). Cela peut se faire en considérant que les distances sont divisées par le nombre d'ensembles et en regroupant ces nouvelles distances par valeur entière (voir figure 6.1). L'impact de ces simplifications seront étudiées en détail dans la troisième partie de ce chapitre.

1. Le chapitre 7 présente les détails concernant l'intégration dans SimSo des modèles qui sont présentés dans ce chapitre.

2. Le nombre d'ensembles est égal à la taille du cache en nombre de lignes divisée par l'associativité.

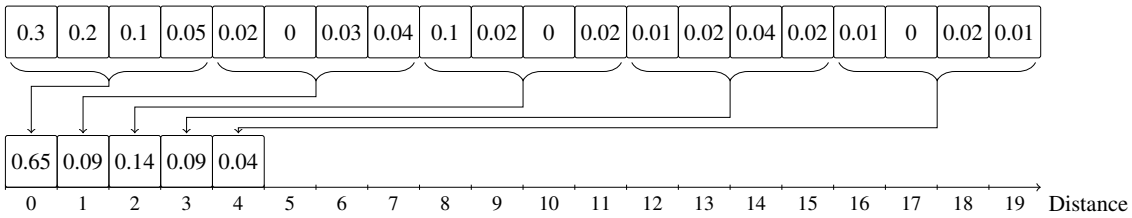


FIGURE 6.1 – Création d'un SDP pour un nombre d'ensembles de 4.

6.1.3 Hypothèses

Nous nous limitons à la seule prise en compte de caches associatifs (par ensembles ou entièrement) avec algorithme de remplacement LRU. Nous ne considérons que les caches de données, le coût lié au chargement des instructions sera pris en compte dans le CPI de base (*base_cpi*) et nous supposerons qu'il existe un cache d'instructions dédié.

Nous supposerons également que les caches sont hiérarchiques et inclusifs. Dans le cas de caches inclusifs, les données présentes dans le cache de premier niveau sont également présentes dans le cache situé au niveau supérieur.

Nous négligerons les coûts liés aux protocoles de cohérences. Il est cependant important de souligner que ce coût devient de moins en moins négligeable avec l'augmentation du nombre de cœurs partageant un unique cache. Ainsi nos travaux se limitent à des architectures dotées de quatre cœurs au maximum.

6.2 Présentation des modèles

Nous présentons ici des modèles qui permettent d'estimer la durée d'exécution d'un travail en évaluant notamment le taux de défauts de cache pour chaque niveau de cache. Dans la première section, nous expliquons comment nous pouvons déterminer le CPI d'un travail à partir des taux de défauts de cache, mais aussi à partir des pénalités associées et du CPI sans considérer les pénalités liées aux accès mémoire. Ce CPI permettra dans la simulation de déterminer, pour un intervalle de temps, le nombre d'instructions exécutées. Les sections suivantes présentent des modèles permettant d'évaluer le taux de défauts de cache dans le cas d'une tâche seule, d'un cache partagé, ou suite à une préemption ou migration.

6.2.1 Évaluation du CPI

Le nombre moyen de cycles par instructions (CPI) peut être calculé à partir des taux de défauts de cache sur les différents niveaux de cache et des pénalités associées à ces défauts de cache. Par la suite, toutes les durées sont exprimées en nombre de cycles processeur.

Notons qu'il existe plusieurs façons de prendre en compte les pénalités liées aux défauts de cache en fonction des entrées disponibles et des hypothèses sur l'architecture [FSS06, KS04, TSW08, HP12, EBSH11]. Ces hypothèses peuvent être l'inclusivité, la prise en compte des défauts de cache pour les instructions, etc.

a) Calcul du CPI

Soient $mr_{\tau,Lx}$ le taux de défauts de cache de la tâche τ au niveau de cache Lx et mp_{Lx} la pénalité associée à un défaut de cache au niveau Lx (c'est-à-dire la différence entre le temps pour accéder au niveau $L(x+1)$ par rapport au temps d'accès du niveau Lx) et mp_0 le temps d'accès au premier niveau de cache.

La pénalité moyenne associée à un accès mémoire (P_τ) est :

$$P_\tau = mp_0 + \sum_{x=1}^{L} mr_{\tau,Lx} \cdot mp_{Lx}, \text{ avec } L \text{ le nombre de niveaux de cache.} \tag{6.1}$$

L'hypothèse de caches inclusifs implique ici qu'un défaut de cache sur un niveau de cache provoquera également des défauts de cache sur les niveaux supérieurs.

À l'aide du taux d'accès au cache par instruction (API_τ), de la pénalité moyenne pour un accès mémoire (P_τ) et du CPI sans prendre en compte les accès mémoire ($base_cpi_\tau$), on peut en déduire le CPI d'une tâche τ, cpi_τ :

$$cpi_\tau = base_cpi_\tau + API_\tau \cdot P_\tau \tag{6.2}$$

Par conséquent, à partir du nombre d'instructions (n_τ) et du CPI, nous pouvons en déduire la durée d'exécution C_τ de la tâche τ :

$$C_\tau = cpi_\tau \cdot n_\tau \tag{6.3}$$

La valeur de $base_cpi_\tau$, nécessaire au calcul du CPI, peut être calculée en exécutant la tâche en isolation et en collectant la durée d'exécution (C_τ), le nombre de défauts de cache par niveau de cache ($mr_{\tau,Lx}$), les pénalités associées (mp_{Lx}), le nombre d'instructions exécutées (n_τ) et le taux d'accès au cache par instruction (API_τ). Le calcul se base sur les équations 6.2 et 6.3 :

$$base_cpi_\tau = \frac{C_\tau}{n_\tau} - API_\tau \cdot P_\tau \tag{6.4}$$

b) Exemple

Soit un système constitué d'un cache L1 par processeur dont le temps d'accès est de 1 cycle, un cache L2 commun à tous les processeurs avec un temps d'accès de 10 cycles et enfin une mémoire avec un temps d'accès de 130 cycles (voir figure 6.2).

Pour cet exemple, la pénalité mp_0 est égale au temps d'accès au niveau L1 (1 cycle), mp_{L1} est égale au temps d'accès au niveau L2 moins celui d'accès au niveau L1 (9 cycles) et mp_{L2} est égale au temps d'accès à la mémoire moins le temps d'accès au niveau L2 (120 cycles). La formule de calcul du CPI est donc :

$$cpi_\tau = base_cpi_\tau + API_\tau \cdot (1 + 9 \cdot mr_{\tau,L1} + 120 \cdot mr_{\tau,L2}) \tag{6.5}$$

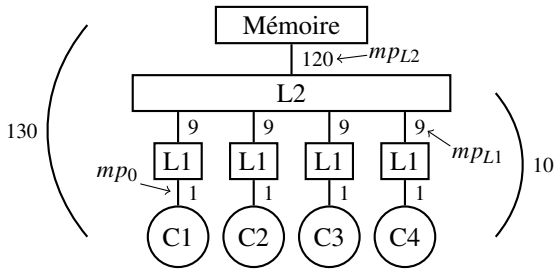


FIGURE 6.2 – Représentation de l'architecture mémoire servant d'exemple.

6.2.2 Défauts de cache pour une tâche isolée

Dans la partie 2.5.4, nous avons présenté le *Stack Distance Profile* (SDP). Cette métrique sera à la base des modèles présentés par la suite afin de déterminer le taux de défauts de cache. Nous voyons ici comment utiliser le SDP afin de déterminer le taux de défauts de cache dans le cas d'un cache qui n'est pas partagé.

On note $sdp(i)$ le nombre d'accès mémoire se faisant à une distance de i, divisé par le nombre d'accès total. La probabilité que i lignes distinctes de cache soient accédées entre deux accès consécutifs à une même ligne est $sdp(i)$. Pour un cache LRU, la proportion des accès dont la *stack distance* est supérieure à l'associativité du cache, correspond exactement au taux de défauts de cache (équation 6.6).

$$mr_{\tau,Lx} = 1 - \sum_{i=0}^{A_{Lx}-1} sdp(i), \text{ avec } A_{Lx} \text{ l'associativité du cache Lx.} \tag{6.6}$$

Cette équation suppose l'utilisation d'un SDP calculé avec un nombre d'ensembles identique à celui du cache utilisé. Malheureusement, ceci impliquerait de disposer d'un SDP pour tout nombre d'ensembles possibles ce qui pose des difficultés dans notre contexte. Nous pourrons donc soit calculer un nouveau SDP tel qu'expliqué dans la section 6.1.2, soit nous limiter au calcul pour des caches entièrement associatifs (SDP calculé pour un seul ensemble). Dans les deux cas, ceci engendre une erreur mais qui devrait être moins importante avec la simple hypothèse d'un cache entièrement associatif. Nous évaluons cette erreur dans la partie 6.3.4. L'équation devient alors :

$$mr_{\tau,Lx} = 1 - \sum_{i=0}^{S_{Lx}-1} sdp(i), \text{ avec } S_{Lx} \text{ la taille du cache Lx en nombre de lignes.} \tag{6.7}$$

En utilisant les équations 6.1, 6.2 et 6.3 pour le calcul du CPI, nous pouvons en déduire le nombre d'instructions exécutées sur un intervalle de temps.

6.2.3 Défauts de cache pour une tâche avec un cache partagé

Le partage d'un cache entre plusieurs cœurs n'est pas toujours équitable et l'exécution de deux tâches en même temps peut provoquer un ralentissement difficile à quantifier a priori. Nous nous intéressons donc ici aux modèles qui permettent de déterminer la vitesse d'exécution d'un programme en prenant en considération l'impact des autres programmes

qui utilisent un même cache partagé. Dans cette partie, nous présenterons les modèles FOA, SDC et Babka.

Nous avons rencontré dans la littérature d'autres modèles que nous n'avons pas retenus car ils étaient difficilement utilisables dans le contexte de la simulation que nous envisageons. Ceci est le cas par exemple des modèles présentés par Sandberg *et al.* [SBSH12, SBSH11] ainsi que Xu *et al.* [XCDM10a] qui se basent sur des données observées au moment de l'exécution que nous ne sommes pas en mesure de fournir. Le modèle Prob de Chandra *et al.* [CGKS05] ou le modèle StatCC de Eklov *et al.* [EBSH11] ont été étudiés, mais nous ne les avons pas retenus car trop complexes d'un point de vue calculatoire pour une simulation efficace. De plus, dans son approche, le modèle de Babka qui sera présenté est similaire à ces deux derniers modèles.

a) Modèle FOA

Le partage du cache entre plusieurs tâches n'est généralement pas équitable, et on comprend intuitivement que si une tâche fait beaucoup d'accès mémoire, elle pourra alors occuper plus d'espace en cache et même empiéter sur les performances des autres tâches.

C'est précisément cette intuition qui est à la base du modèle *Frequency of Access* (FOA) [CGKS05]. Ce modèle prend pour hypothèse que chaque travail dispose d'un espace effectif en cache proportionnel à sa fréquence d'accès. Dans ce cas, le comportement observé serait donc proche de celui où chaque tâche possède son propre cache mais d'une taille plus petite et tel que la somme des tailles corresponde à la taille du cache partagé. Il devient alors possible d'appliquer les mêmes formules que dans le cas de l'exécution seule d'une tâche (équation 6.7) mais avec des caches plus petits.

La fréquence d'accès pour une tâche correspond au taux d'accès à la mémoire divisé par le CPI. Le modèle FOA précise que le CPI utilisé ici correspond au CPI calculé pour une exécution seule. On note Af_τ la fréquence d'accès de la tâche τ :

$$Af_\tau = \frac{API_\tau}{cpi_alone_\tau} \tag{6.8}$$

Soit S la taille du cache et Nb le nombre de tâches en concurrence. L'équation 6.9 permet de calculer S_τ, la taille virtuelle du cache pour la tâche τ.

$$S_\tau = \frac{Af_\tau}{\sum_{i=1}^{Nb} Af_i} \cdot S \tag{6.9}$$

Pour chaque tâche, une fois la taille du cache virtuel obtenue, il est possible d'utiliser le SDP de la tâche afin de déterminer son nombre de défauts de cache à l'aide de l'équation 6.7 (une interpolation linéaire permet de traiter le cas où S_τ n'est pas une valeur entière). L'associativité du cache peut être prise en compte en remplaçant S par l'associativité dans l'équation 6.9 et en utilisant l'équation 6.6. Notons que le modèle FOA peut être appliqué pour des caches non partagés puisque l'équation 6.9 donne dans ce cas $S_\tau = S$.

La figure 6.3 illustre à travers un exemple le calcul du CPI en tenant compte du partage d'un unique cache. Dans le cas d'une architecture plus complexe dotée de plusieurs niveaux de cache, le calcul du taux de défauts de cache doit se faire sur chaque niveau avant de calculer le CPI (équation 6.1).

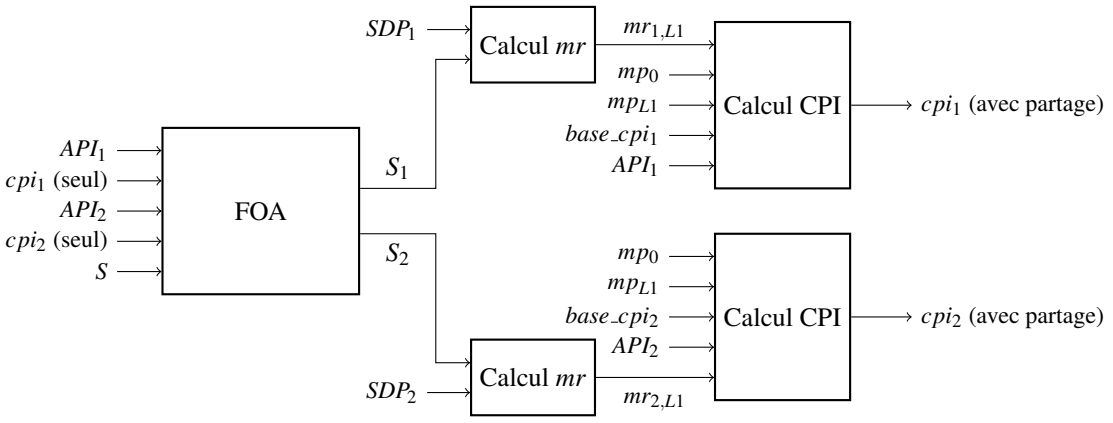


FIGURE 6.3 – Calcul du CPI de deux tâches avec partage d'un cache L1 en utilisant le modèle FOA.

Le point fort de ce modèle est sa simplicité de mise en œuvre et surtout sa vitesse de calcul. Cependant, il ne prend pas en compte le profil (SDP) des tâches dans le partage. Les résultats expérimentaux fournis par les auteurs donnent des résultats très bons lorsque le cache partagé n'est pas utilisé de manière très intensive ($< 5\%$ d'erreur) et en moyenne donne une erreur inférieure à 20%. Des expérimentations supplémentaires seront nécessaires afin de déterminer si le modèle est suffisamment précis pour notre cas d'utilisation (voir section 6.3.5).

b) Modèle SDC

Chandra *et al.* proposent un autre modèle appelé *Stack Distance Competition* (SDC) qui se base sur les *stack distances* [CGKS05]. L'objectif est encore de déterminer la taille de cache effective pour chaque tâche mais cette fois l'algorithme va privilégier la tâche qui a une forte réutilisation de ses lignes.

Pour cela, l'algorithme construit une distribution de *stack distances* à partir des *stack distances* des tâches en concurrence (en se limitant aux distances inférieures à l'associativité du cache). La figure 6.4 montre la fusion de deux SDP pour un cache d'associativité 8.

Initialement, un pointeur par SDP pointe sur la première valeur de la distribution. À chaque itération, l'algorithme choisit parmi les valeurs pointées la plus grande. La valeur est alors ajoutée au SDP en cours de construction et le pointeur est incrémenté. Au bout d'un nombre d'itérations correspondant à l'associativité, l'algorithme se termine.

La proportion de distances appartenant à une tâche correspond alors à la proportion effective du cache que la tâche détient. Dans la distribution résultante de l'exemple de la figure 6.4, trois distances appartiennent au programme n° 1 et cinq distances appartiennent au programme n° 2. Par conséquent, le modèle SDC attribue une taille de cache de trois huitièmes pour le premier programme et de cinq huitièmes pour le second.

La figure 6.5 montre l'utilisation du modèle SDC pour calculer le CPI. Nous voyons que son utilisation est similaire à celle de FOA mais en utilisant des entrées différentes.

D'après les résultats fournis par les auteurs, la qualité du modèle SDC est assez similaire au FOA mais avec une erreur maximale bien inférieure. En effet, dans leurs expérimentations il existe un cas où FOA donne des résultats avec une erreur de 264% alors que SDC

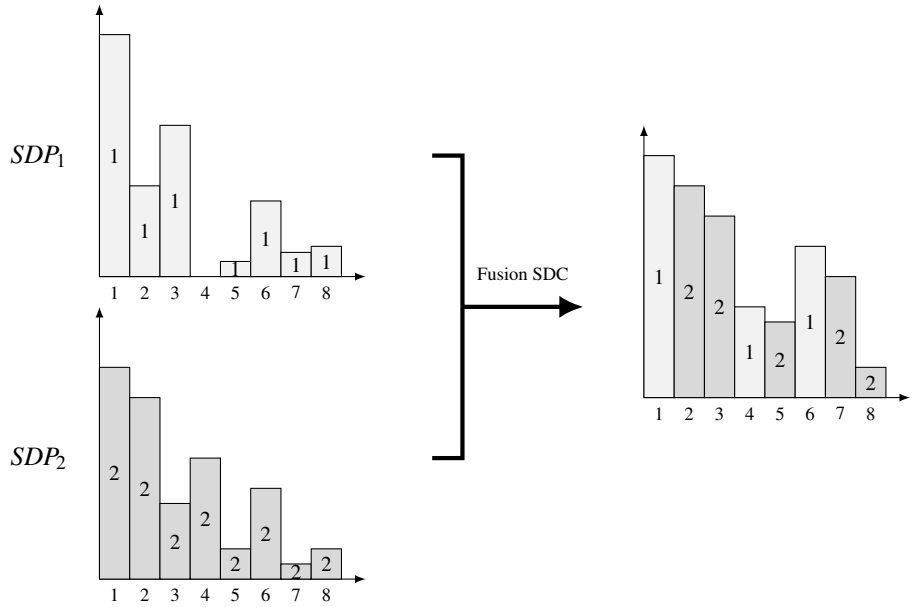


FIGURE 6.4 – Illustration de l'algorithme de fusion de deux SDP calculés pour un cache d'associativité 8.

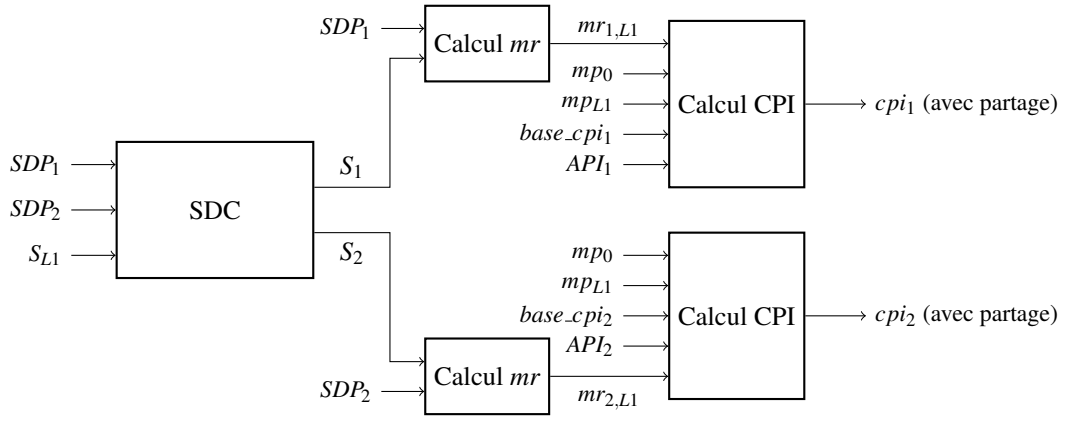


FIGURE 6.5 – Calcul du CPI des tâches avec partage d'un cache L1 en utilisant le modèle SDC.

ne génère une erreur que de 25% sur cet exemple. Les benchmarks utilisés sont issus de SPEC2000.

c) Modèle Babka

Vlastimil Babka a proposé un modèle pour estimer l'influence d'un cache partagé sur l'exécution des travaux [BLMT12]. Son modèle, que nous appellerons Babka par la suite, prend en entrée, pour chaque tâche, le SDP, l'API et le CPI. En sortie, un nouveau SDP et CPI sont déterminés en prenant en compte la concurrence sur le cache partagé.

Le SDP d'une tâche qui prend en compte l'effet des autres travaux qui partagent le même cache est calculé en trois étapes. Une fois le nouveau SDP calculé, le CPI de la tâche est réévalué afin de prendre en compte l'effet des autres travaux qui s'exécutent en compétition. Or, si le programme est plus lent (augmentation du CPI), alors le programme perturbera

moins les autres. Ainsi, une nouvelle itération du calcul est réalisée avec le nouveau CPI pour chaque tâche et ainsi de suite jusqu'à convergence.

Ces étapes consistent à modéliser le comportement « réel » du cache, à savoir que des lignes de différents travaux sont entrelacées dans le cache partagé. Ceci a donc pour effet d'augmenter les *stack distances*. Les trois étapes, appliquées à chaque distance d du SDP sont :

1. Calcul du temps moyen (en cycles processeur) entre deux accès consécutifs à une même ligne avec une distance de d. On note ce temps r_d.
2. Estimation du nombre de lignes distinctes accédées par les autres travaux durant ce temps r_d.
3. Augmentation de la stack distance d'origine par la distribution de probabilité estimée lors de l'étape 2.

Étape 1 : calcul du temps moyen La première étape consiste à calculer le temps moyen (en cycles processeur) entre deux accès consécutifs à une même ligne avec une distance de d. En d'autres termes, on cherche le temps moyen pour que $d-1$ lignes de cache distinctes soient accédées avant d'accéder à nouveau à la même ligne. Ceci est le temps moyen pour passer d'une occupation de cache de 0 à d lignes distinctes.

L'occupation de d lignes par une tâche peut être modélisée par une chaîne de Markov avec $d+1$ états, chaque état représentant le nombre de lignes de cache occupées par la tâche (Figure 6.6).

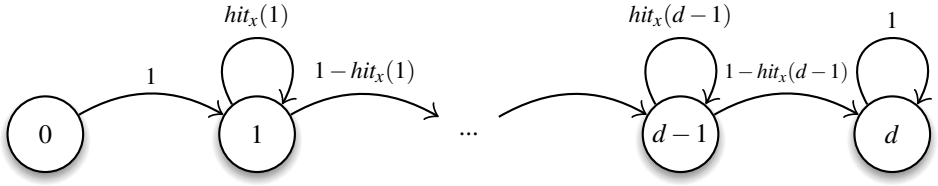


FIGURE 6.6 – Chaîne de Markov modélisant l'occupation d'un cache de d lignes.

La probabilité que le nombre de lignes distinctes n'augmente pas est égale à la probabilité de faire un accès à une ligne déjà présente. On note, $hit_\tau(i)$ la probabilité d'un succès de cache quand i lignes sont déjà occupées par la tâche τ :

$$hit_\tau(i) = \sum_{j=0}^{i-1} sdp_\tau(j) \tag{6.10}$$

On note $\mathbf{P}_{d,\tau}$ la matrice de transition associée à cette chaîne de Markov. L'état d étant absorbant ($P_{d,\tau}(d,d) = 1$), il est possible de calculer $t_{d,\tau}$ l'espérance du nombre d'étapes pour atteindre cet état à partir de l'état initial. Cette valeur correspond à moyenne de la première ligne de la matrice fondamentale.

On en déduit directement le temps moyen de réutilisation pour une distance d d'une tâche τ :

$$r_{d,\tau} = \frac{t_{d,\tau}}{API_\tau / CPI_\tau} \tag{6.11}$$

Étape 2 : Estimation du nombre de lignes distinctes accédées Pour calculer l'interférence causée par une autre tâche (notée tâche 2) sur la tâche 1, il convient d'estimer le nombre moyen d'accès de la tâche 2 pendant le temps de réutilisation de la tâche 1 :

$$a_{d,2} = r_{d,1} \cdot \frac{API_2}{CPI_2} \tag{6.12}$$

Un vecteur de probabilité d'utilisation des lignes du cache, suite à $a_{d,2}$ accès, est calculé à partir de la matrice de transition $\mathbf{P}_{A,2}$, avec A l'associativité du cache. On note $\mathbf{D}(a_{d,2})$ ce vecteur de probabilité :

$$\mathbf{D}(a_{d,2}) = \mathbf{u}\mathbf{P}_{A,2}^{a_{d,2}} \text{ , avec } \mathbf{u} = \{1, 0, ..., 0\} \tag{6.13}$$

Le vecteur $\mathbf{D}(a_{d,2})$ donne la distribution de probabilité du nombre de lignes accédées par la tâche 2 sur une durée $r_{d,1}$. Pour les valeurs non-entières de $a_{d,2}$, une interpolation linéaire est utilisée entre les vecteurs $\mathbf{D}(\lfloor a_{d,2} \rfloor)$ et $\mathbf{D}(\lceil a_{d,2} \rceil)$.

Ce calcul est réalisé pour l'ensemble des tâches en cours d'exécution et qui partagent le même cache.

Étape 3 : Augmentation de la stack distance Un profil de distances partielles $pdp'_{d,\tau}$ décrit pour une distance initiale d de la tâche τ, la nouvelle distribution en distance de cette tâche en prenant en considération le partage de cache. Il est calculé en distribuant la valeur initiale $sdp_\tau(d)$ sur les distances comprises entre d et $d + A$ proportionnellement aux valeurs de $\mathbf{D}(a_{d,y})$, $y \neq \tau$. C'est-à-dire que la distance d est augmentée d'une valeur égale aux distances évaluées par $\mathbf{D}(a_{d,y})$. Les accès ayant une distance supérieure à A étant forcément des échecs, ils sont simplement cumulés à la distance $A + 1$.

Calcul du nouveau SDP : En appliquant les trois étapes à chaque distance d, $0 \leq d \leq A$, le profil de distance de la tâche 1 avec le partage de cache, sdp'_1, est évalué en additionnant les distances partielles $pdp'_{d,1}$.

Calcul du nouveau CPI : Pour calculer le nombre de cycles d'horloge par instruction de la tâche 1 en tenant compte du cache partagé, cpi'_1, il faut introduire le ratio d'échecs d'accès au cache. On commence par calculer la différence de taux de succès avec le cache partagé :

$$\Delta mpa_1 = hit_1(A) - hit'_1(A) \tag{6.14}$$

ce qui permet d'évaluer le coût supplémentaire sur le CPI en intégrant la pénalité MP d'échec d'accès au cache :

$$\Delta cpi_1 = \Delta mpa_1 . API_1 . MP \tag{6.15}$$

La valeur du cpi'_x résultant est alors :

$$cpi'_1 = cpi_1 + \Delta cpi_1 \tag{6.16}$$

Les mêmes calculs doivent être réalisés pour les autres tâches qui s'exécutent en parallèle. Cependant, l'évaluation des équations (6.11) et (6.12) a été faite pour les valeurs des CPI sans partage de cache. Il est donc nécessaire de recommencer l'ensemble des calculs à partir des nouveaux CPI calculés. Le critère d'arrêt se fait sur une simple évaluation d'une valeur ϵ entre les CPI de deux solutions consécutives.

La figure 6.7 résume les entrées et sorties du modèle.

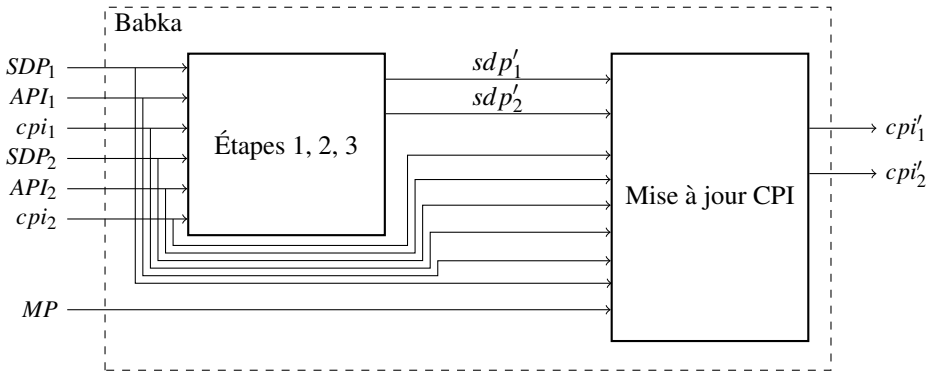


FIGURE 6.7 – Entrées et sorties du modèle de Babka.

Remarques Les étapes 1 et 2 nécessitent la manipulation de matrices dont la taille peut atteindre l'associativité du cache. Les manipulations sur ces matrices peuvent être très lourdes et dépendent là encore de l'associativité. Par conséquent, pour des raisons de temps de calcul, il est impossible d'appliquer ce modèle avec des SDP calculés pour des caches entièrement associatifs pour lesquels l'associativité peut dépasser des milliers de lignes.

6.2.4 Estimation des coûts de préemption

Cette partie s'intéresse à la prise en compte des coûts de préemption. La figure 6.8 illustre la perturbation d'un cache suite à une préemption. Nous voyons que pour évaluer cette préemption nous avons besoin d'estimer le nombre de lignes présentes dans le cache avant l'interruption et le nombre de lignes présentes au moment de la reprise. La différence nous indique la perturbation δ que le cache a subi. De cette perturbation, nous en déduisons le nombre de défauts de cache supplémentaires causés par la préemption et qui n'auraient pas eu lieu sinon.

Afin d'estimer le nombre de lignes présentes dans le cache avant et après la préemption, nous avons besoin d'évaluer le nombre de lignes utilisées par un programme en fonction du nombre d'accès mémoire (que nous connaissons à l'aide de API, CPI et de la durée d'exécution). Pour cela, nous avons dans un premier temps essayé d'estimer le chargement du cache selon une loi exponentielle mais les résultats ne furent pas bons et nous en expliquons donc la raison. Nous présentons alors un second modèle qui se base sur la chaine de Markov introduite par le modèle de Babka.

Une fois la perturbation δ évaluée à partir de l'évaluation du nombre de lignes présentes dans le cache, nous calculons le nombre de défauts de cache qui en résultent. Nous présentons succinctement le modèle de Liu *et al.* [LGS+08] et nous proposons deux autres modèles beaucoup plus rapides à calculer.

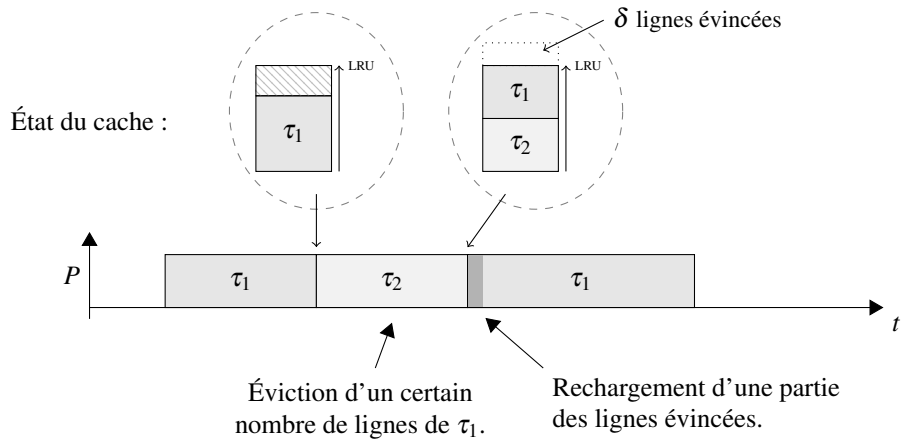


FIGURE 6.8 – Exemple de préemption.

a) Estimation du nombre de lignes

Estimation du chargement sans SDP

Nous avons initialement tenté d'estimer le chargement du cache sans nous baser sur le SDP mais uniquement à partir du nombre de lignes différentes utilisées par la tâche et du nombre d'accès mémoire au total. Pour ce faire, nous avons supposé un chargement basé sur l'hypothèse que la probabilité de faire un *cold miss* suit une loi exponentielle. Malheureusement, ces seules informations ne sont pas suffisantes. D'une part, la distribution des *cold misses* au cours du temps varie beaucoup d'une tâche à une autre (voir section 6.3.2). Mais surtout, si on se place dans le cas de l'exécution d'une partie arbitraire de la tâche, ces deux seules informations ne nous donnent pas d'indications sur la quantité de lignes déjà utilisées antérieurement qui seront réutilisées. En effet, deux tâches faisant le même nombre d'accès mémoire et utilisant au total le même nombre de lignes peuvent avoir des comportements très différents sur ce point.

La propension à réutiliser des lignes qui sont apparues plus tôt est cependant contenue dans le SDP.

Utilisation du modèle de Babka

En se basant sur la chaîne de Markov introduite dans la présentation du modèle de Babka (voir figure 6.6), nous proposons un modèle capable d'estimer le nombre de lignes de cache différentes utilisées après n accès mémoire en partant d'un cache vide.

En utilisant la matrice $\mathbf{P}_{A,x}$ élevée à la puissance n, nous obtenons sur la première ligne la distribution de probabilité du nombre de lignes utilisées (vecteur $\mathbf{D}(n)$). Il suffit alors de calculer la moyenne pondérée de $\mathbf{D}(n)$ pour obtenir le nombre moyen de lignes de cache utilisées après n accès :

$$l(n) = \sum_{m=0}^{A} m.\mathbf{D}_m(n) \tag{6.17}$$

L'écart type est donné par :

$$\sigma(n) = \sqrt{\sum_{m=0}^{A} \left(m.\mathbf{D}_m(n) - l(n)\right)^2} \tag{6.18}$$

Malheureusement, nous ne sommes pas en mesure d'évaluer le nombre de lignes pour un cache qui n'est pas initialement vide. Par conséquent, l'utilisation de ce modèle impose de prendre pour hypothèse que le cache est entièrement vide au début de l'exécution (ou de la reprise d'exécution).

b) Estimation des défauts de cache causés par une perturbation

Modèle de Liu

Dans la partie 2.4.2, nous avons retranscrit la distinction faite par Liu *et al.* des deux types de défauts de cache consécutifs à une préemption [LGS+08]. Lors de la reprise d'exécution de l'application, un certain nombre de lignes du cache ont été remplacées par une autre application, il est donc nécessaire d'en recharger une partie, les auteurs désignent ces défauts de cache par l'expression « *replaced context switch misses* ». Cependant, à la reprise de l'exécution de l'application, celle-ci va exclure ses propres données en ajoutant de nouvelles lignes dans le cache. Ce cas est désigné par l'expression « *reordered context switch misses* ». Ces deux types de défauts de cache sont pris en compte par le modèle de Liu *et al.* présenté dans les grandes lignes ci-dessous.

Description du modèle : La perturbation subie par une tâche utilisant un cache d'associativité A sera modélisée par un décalage δ. En effet, toutes les données utilisées par la tâche et qui sont présentes dans le cache vont *vieillir* et donc sortir du cache progressivement. Ce décalage peut prendre les valeurs de 0 à A inclus. Les δ premières valeurs dans le cache, juste après la perturbation, sont appelés des « trous » (voir figure 6.9). Ces trous vont naturellement se déplacer pour sortir lorsque l'application fera des accès mémoires.

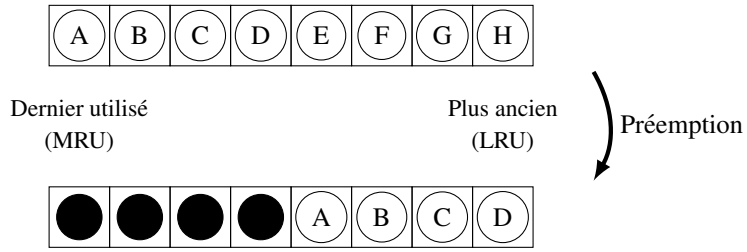


FIGURE 6.9 – État du cache à la reprise d'exécution, après une préemption.

L'état du cache est modélisé par le couple de valeurs (c, h) où c est le nombre de lignes appartenant à l'application considérée et h est la position du trou le plus récemment utilisé (indice de départ à 1). Suite à la préemption, nous avons donc $h = 1$ et $c \in [\![0, A]\!]$. Lorsque les trous ont été sortis du cache, ou s'il n'y a pas eu de perturbation, on a $h = A + 1$.

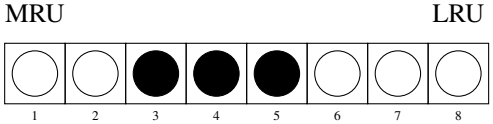


FIGURE 6.10 – État du cache modélisé par le couple $(5, 3)$.

Le modèle s'intéresse à la probabilité que l'état du cache soit (c, h) après n accès mémoires, pour toutes les valeurs de c et h possibles. Chaque état du cache implique une certaine probabilité de faire un défaut de cache. La probabilité d'atteindre un certain état du cache est calculé après une modélisation sous la forme d'une chaine de Markov.

La formule récursive suivante est obtenue à partir de la chaine de Markov. Le lecteur intéressé par les étapes intermédiaires qui ont mené à ce résultat est invité à lire l'article de Liu *et al* [LGS+08].

$$P(c,h,n) = \begin{cases} 1, & \text{si } c = A-\delta, h = 1, n = 0 \\ P(c,h,n-1) \times P_{stay}(h) + P(c,h-1,n-1) \times P_{shift}(h) & \\ \quad + P(c-1,h-1,n-1) \times P_{shift}(h), & \text{si } c \geq A-\delta, h = c+1, n > 0 \\ P(c,h,n-1) \times P_{stay}(h) + P(c,h-1,n-1) \times P_{shift}(h), & \text{si } c \geq A-\delta, h < c+1, n > 0 \\ 0 & \text{dans les autres cas} \end{cases} \tag{6.19}$$

Avec $sdp(i)$, la probabilité d'accès à la i-ème ligne du cache, P_{stay}, la probabilité que les trous restent à leur position :

$$P_{stay} = \sum_{j=1}^{h-1} sdp(j) \tag{6.20}$$

Et P_{shift}, la probabilité qu'un accès mémoire crée un décalage :

$$P_{shift} = sdp(> A) + \sum_{j=h-1}^{A} sdp(j) \tag{6.21}$$

La probabilité de faire un défaut de cache est la somme des probabilités de faire des accès à des positions strictement supérieures à c (puisqu'il ne reste que c lignes de l'application dans le cache).

$$P_{condmiss}(c) = \sum_{j=c+1}^{\infty} sdp(j) \tag{6.22}$$

Le nombre de défauts de cache est donc de :

$$TotMisses(N) = \sum_{c=A-\delta}^{A} \sum_{h=1}^{A+1} \sum_{n=0}^{N-1} (P(c,h,n) \times P_{condmiss}(c)) \tag{6.23}$$

Si on s'intéresse uniquement au nombre de défauts de cache qui résultent du changement de contexte, il faut soustraire le nombre de défauts de cache qui seraient intervenus sans la perturbation :

$$CSMisses(N) = TotMisses(N) - N \times sdp(> A) \tag{6.24}$$

Commentaires : Le temps de calcul de ce modèle peut être long pour une utilisation dans notre simulateur. En effet, d'après les équations 6.23 et 6.19, le calcul est de l'ordre de $\mathcal{O}(n^2 \times A \times \delta)$ avec n le nombre d'accès mémoire, A l'associativité du cache et δ le décalage. Sachant que n peut valoir plusieurs millions.

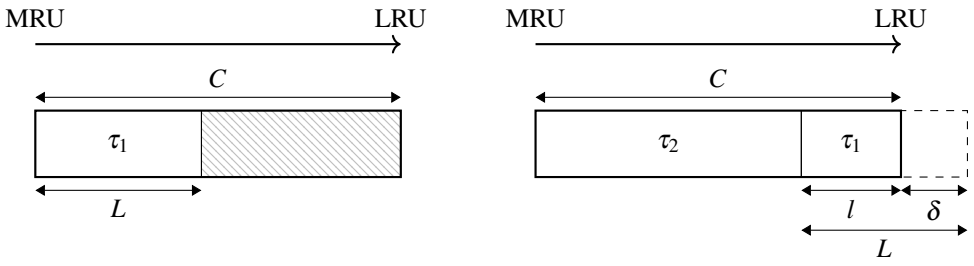
Autres modèles

Nous proposons ici deux modèles qui se basent sur des hypothèses très simplificatrices pour estimer le nombre de défauts de cache causés par la perturbation d'un cache suite à une préemption. Ces modèles sont évalués dans la seconde partie de ce chapitre. Afin de simplifier les explications, nous supposerons que le cache est entièrement associatif. La généralisation à des caches associatifs par ensemble est triviale.

La figure 6.8 montre une tâche τ_1 s'exécuter puis se faire préempter par la tâche τ_2 qui s'exécute puis laisse la tâche τ_1 se terminer. Pendant l'exécution de la tâche τ_2, un certain nombre de lignes du cache ont pu être « poussées » en dehors du cache. L'évaluation du nombre de lignes doit nous permettre de déterminer l'état du cache avant et après la préemption. Nous pourrons en déduire le nombre de lignes de τ_1 qui ont été éliminées, que nous appelons la perturbation δ.

Nous supposons que le cache contient, avant la préemption, L lignes de cache appartenant à la tâche τ_1 et que ces lignes sont les dernières utilisées. La figure 6.11a illustre l'état du cache au moment où la tâche τ_1 se fait préempter.

En estimant le nombre de lignes utilisées par la tâche τ_2, on peut en déduire le nombre de lignes l appartenant à la tâche τ_1 qui sont encore présentes dans le cache au moment de la reprise de l'exécution de la tâche τ_1. On en déduit δ le nombre de lignes de la tâche τ_1 qui ont été évincées du cache. La figure 6.11b montre l'état du cache au moment de la reprise de l'exécution de la tâche τ_1.



(a) Avant préemption. (b) Après préemption.

FIGURE 6.11 – État du cache avant et après l'exécution de la tâche τ_2.

Ces δ lignes ne sont pas nécessairement toutes encore utiles pour la tâche τ_1, ainsi, nous ne pouvons pas en déduire que le nombre de défauts de cache engendrés par la préemption est égal à δ. Les deux modèles que nous proposons visent à approximer le nombre R de défauts de cache.

De plus, si après la reprise d'exécution il y a n accès mémoires, seuls $m = n \times (\sum_{i=l}^{L} sdp(i))$ accès au maximum sont des accès visant des lignes exclues par la tâche τ_2. On en déduit le nombre de défauts de cache pour n accès mémoires suite à l'éviction de δ lignes du cache :

$$min\left(R, n \times \sum_{i=l}^{L} sdp(i)\right) \tag{6.25}$$

Modèle n°1 : La probabilité d'utiliser une des lignes qui a été exclue du cache correspond à la probabilité de faire un accès à une ligne se situant à une distance comprise dans $[\![l, L]\!]$. Cette probabilité varie rapidement avec les accès au cache, mais nous supposons ici que cette probabilité reste inchangée (cette hypothèse pourrait être la source d'erreurs,

notamment pour des caches entièrement vidés). On en déduit R, une estimation du nombre maximum de défauts de cache induits par ce changement de contexte.

$$R = \delta \times \sum_{i=l}^{L} sdp(i) \tag{6.26}$$

Modèle n°2 : Le second modèle proposé consiste à évaluer la distance moyenne des accès pouvant provoquer un défaut de cache. Ceci se fait à l'aide d'une moyenne pondérée :

$$R = \sum_{i=l}^{L} (i \times sdp(i)) \tag{6.27}$$

6.2.5 Estimation des défauts de cache suite à une migration

Nous nous plaçons maintenant dans le cas où une tâche reprend son exécution sur un processeur différent. Afin d'évaluer le nombre de défauts de cache causés par la migration, nous avons deux possibilités : soit nous estimons de la même manière le nombre de lignes dans les caches pour en déduire la perturbation ; soit nous supposons que la reprise d'exécution sur un autre processeur implique que les caches ont été entièrement vidés.

La première possibilité est motivée d'une part par le fait que si un cache est partagé entre les deux processeurs concernés par la migration, alors il n'y a aucune différence avec une préemption. D'autre part, dans le cas de migrations fréquentes, des lignes appartenant à la tâche sont peut être encore disponibles dans un cache privé.

Mais nous pourrions aussi prendre pour hypothèse que le cache du processeur qui accueille la tâche ne contient aucune ligne de celle-ci. En effet, même s'il reste des lignes, si celles-ci sont trop anciennes, elles sont probablement devenues inutiles. Ceci va donc dépendre de l'ancienneté de la dernière exécution sur ce processeur.

La stratégie adoptée sera décidée en fonction du résultat des évaluations proposées dans la seconde partie de ce chapitre.

6.3 Évaluation des modèles

Dans cette seconde partie, nous présentons les résultats obtenus à travers plusieurs expérimentations prenant en compte le comportement des caches. L'objectif de ces évaluations est de déterminer la précision des estimations produites par les différents modèles en comparant leurs prévisions à des exécutions sur un simulateur d'architecture matérielle. De cette étude dépendra le choix des modèles que nous intègrerons dans SimSo pour prendre en compte les caches dans la durée d'exécution des travaux.

Pour mener ces études, nous avons rassemblé un ensemble de programmes issus de plusieurs benchmarks principalement orientés temps réel ou systèmes embarqués. Ensuite nous avons mis au point des outils pour extraire de ces programmes les métriques qui nous intéressent à partir de traces d'exécution. Au cours de cette partie, nous étudions le profil des programmes à travers leur SDP mais aussi la position des *cold misses*. Nous verrons quelle influence a l'associativité sur les taux de défauts de cache et si à partir

d'un SDP généré pour un cache entièrement associatif nous pouvons regénérer un SDP équivalent pour un nombre d'ensembles arbitraire. Enfin, nous étudierons le cas de caches partagés et les effets des préemptions.

6.3.1 Choix des outils et benchmarks

Le comportement vis-à-vis des caches varie fortement d'un programme à un autre. Ainsi, nous avons besoin d'une suite de programmes représentatifs des différents types de comportement.

Pour étudier ces programmes, nous avons également besoin d'outils pour obtenir les durées d'exécution ou taux de défauts de cache par exécution, et collecter les métriques qui servent d'entrée aux modèles (SDP, API, CPI et nombre d'instructions).

a) Benchmarks

Dans le but d'étudier l'impact des caches sur l'exécution des programmes, il est nécessaire de disposer d'un ensemble de programmes aux comportements variés. Dans le domaine non temps réel, l'ensemble de benchmarks le plus utilisé semble être celui du Standard Performance Evaluation Corporation CPU (SPEC CPU). Mais dans le domaine temps réel, aucun benchmark ne s'est réellement imposé au sein de la communauté. Cela est d'autant plus difficile que les critères de performances sont mal définis [3] : overhead, ordonnancement, charge, etc. Dans l'annexe A.5 nous listons un ensemble de benchmarks. Certains sont orientés métier, d'autres ont été spécialement créés par la communauté scientifique pour l'évaluation des outils et méthodes.

Afin de mener les expérimentations qui suivent, nous avons sélectionné des programmes parmi les suites MiBench et Mälardalen. Nous pensons que ces programmes sont plus représentatif du type de calcul que nous pourrions faire dans un système embarqué temps réel que ceux de SPEC CPU. MiBench offre une large sélection d'opérations concrètes, cependant, certains d'entre eux ne compilent plus avec des compilateurs trop récents et d'autres ne s'exécutent pas correctement sur le simulateur d'architecture gem5. Certains programmes sont trop lents ou au contraire ils ne font pas assez d'accès mémoire pour être intéressants. Nous avons retenu une douzaine de programmes de MiBench. Concernant les programmes du benchmark Mälardalen, leur principal défaut est qu'ils ont été développés dans l'optique de faire de l'évaluation de WCET et pour des raisons pratiques les programmes sont très courts. Nous avons donc retenu une dizaine d'entre eux pour lesquels nous avons augmenté la durée de calcul (augmentation de la taille des données manipulées). Certains programmes ont été compilés avec des options d'optimisation différentes. Enfin, nous avons ajouté le programme *edge* afin de couvrir un comportement supplémentaire.

Le tableau 6.1 donne la liste des programmes qui ont servi pour les évaluations.

b) Exécution d'un programme

Nous avons fait le choix d'utiliser un simulateur d'architecture matérielle afin d'exécuter des programmes tout en maitrisant les caractéristiques matérielles du système (nombre

3. N. Weiderman a proposé dès 1989 dans [Wei90] des critères pour évaluer les systèmes temps réel, mais cela ne semble pas avoir eu d'écho.

Nom	Description
qsort-small	Tri d'une liste de 10000 mots avec qsort
Susan	Traitement d'image [SB97]
FFT-small	Transformée de Fourier rapide de longueur 8192
FFT-large	Transformée de Fourier rapide de longueur 32768
CRC	Contrôle de redondance cyclique sur un fichier de 26 Mo
SHA-small	*Secure Hash Algorithm* sur une chaîne d'environ 300 ko
SHA-large	*Secure Hash Algorithm* sur une chaîne d'environ 3 Mo
Blowfish	Chiffrement symétrique sur une chaîne d'environ 300 ko
JPEG	Compression JPEG d'un fichier d'environ 200 ko
Dijkstra-large	Plus court chemin sur un graphe dense de 100 par 100
Patricia	Manipulation d'une structure de type *Patricia trie* (arbre radix)
Say	Synthèse vocale
StringSearch	Cherche un mot dans une phrase (environ 1300 fois)
UD	Calcul matriciel (3 boucles imbriquées)
FIR	Filtre à réponse impulsionnelle finie
JFDCTINT	Transformée en cosinus discrète
ADPCM	*Adaptive Differential Pulse Code Modulation*
COVER	Une boucle avec beaucoup de blocs *switch*
CNT	Compte les nombres positifs dans une matrice (200 par 200)
COMPRESS	Compression de données
NSICHNEU	Simule un réseau de Petri
NS	Recherche dans un tableau multi-dimensionnel
MATMULT	Multiplication de 2 matrices de 80x80
UD-O2	Idem que UD mais compilé en O2
MATMULT-O2	Idem que MATMULT mais compilé en O2
JFDCTINT	Idem que JFDCTINT mais compilé en O2
ADPCM-O2	Idem que ADPCM mais compilé en O2
COVER-O2	Idem que COVER mais compilé en O2
CNT-O2	Idem que CNT mais compilé en O2
COMPRESS-O2	Idem que COMPRESS mais compilé en O2
edge	Détection de contour sur une image png

Tableau 6.1 – Liste des programmes servant de base pour les évaluations.

de cœurs, hiérarchie mémoire et taille des caches entre autres). Dans l'annexe A.3, une liste d'outils de ce type est présentée. Nous avons retenu gem5 qui a pour avantage d'être très utilisé dans la communauté de l'évaluation de performances des plateformes, d'être toujours développé et de disposer d'une communauté active. Gem5 est disponible librement et son développement a reçu le soutien de AMD, ARM, Intel, IBM ou encore Sun pour ne citer qu'eux.

Gem5 dispose de deux modes de fonctionnement : *full system* et *syscall emulation*. Le premier mode nécessite un système d'exploitation tandis que le second simule les principaux appels système et permet d'exécuter directement les programmes binaires au format ELF compilés statiquement. Nous avons opté pour ce second mode pour simplifier les évaluations et éviter d'être parasité par les services du système d'exploitation.

Après chaque exécution, gem5 génère un fichier nommé `stats.txt` et qui contient des informations sur la simulation. On y retrouve en particulier la durée de la simulation, le nombre d'instructions exécutées, le nombre d'accès aux différents caches ou encore le nombre d'accès au cache réussis et le nombre de défauts.

c) Mesure de *Stack Distance Profile*

Afin de mesurer les SDP, nous avons initialement utilisé MICA (voir annexe A.4). MICA est un outil qui instrumente l'exécution d'un programme et qui permet d'obtenir des informations comme les adresses accédées en lecture et écriture. Un profil proche du SDP était disponible, mais n'était pas suffisant [4], nous avons donc contribué à MICA en proposant un profil plus complet. Malheureusement, le SDP mesuré par MICA ne prend pas en compte les instructions exécutées par le noyau du système d'exploitation. C'est notamment le cas de tous les appels système. Par conséquent, le SDP généré avec MICA ne prenant pas en compte les mêmes instructions que gem5, il nous était difficile de comparer les résultats.

Nous avons donc abandonné MICA et décidé de nous baser sur gem5 pour construire le SDP. Pour cela, nous avons modifié le logiciel afin de journaliser les accès mémoire. Le fichier qui contient le journal est ensuite filtré pour en extraire la séquence des accès mémoires en lecture sur le cache L1 qui sont réalisés par le programme. Un second programme prend en entrée cette séquence et calcule le SDP en simulant un cache LRU de taille infinie. Toutes ces étapes, orchestrées par un script Bash, sont résumées par la figure 6.12.

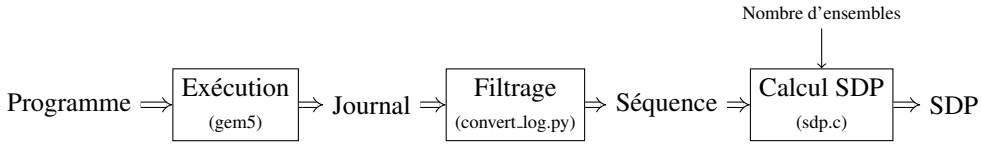


FIGURE 6.12 – Étapes permettant de générer le SDP d'un programme.

Le programme de calcul du SDP prend en argument le nombre d'ensembles à utiliser si l'on souhaite générer des SDP pour des caches associatifs par ensembles.

d) Simulateur de cache

Gem5 est capable de simuler le comportement fonctionnel et temporel des caches. Cependant, pour certaines études, il n'est pas nécessaire de prendre en compte les aspects temporels. Ainsi, au lieu d'utiliser gem5 qui a l'inconvénient d'être assez lent et difficile à manipuler, nous avons développé un simulateur de cache LRU. Ce simulateur prend en entrée une trace des accès mémoire produite par gem5 (celle utilisée pour calculer les SDP) ainsi que les caractéristiques du cache (taille et associativité). À partir d'une seule trace, nous pouvons facilement faire varier les caractéristiques du cache.

Ce simulateur se présente sous la forme d'un ensemble de fonctions en C que nous appelons en fonction de ce que nous cherchons à expérimenter. À tout moment, nous pouvons obtenir le taux de défauts de cache ou le nombre de lignes présentes dans le cache. De plus, grâce à cet outil, nous sommes également en mesure d'évincer des lignes pour simuler la perturbation d'une préemption et en voir les effets.

Le nombre de défauts de cache calculé par ce simulateur est identique à celui obtenu à l'aide de gem5.

4. Le profil généré ne comportait des valeurs que pour des puissances de deux.

6.3.2 Caractérisation du comportement mémoire des programmes

a) *Stack Distance Profile* des programmes

Certains travaux laissent entendre que le *Stack Distance Profile* d'un programme ressemble à une distribution géométrique car selon le principe de localité d'accès aux données, la probabilité de faire des accès à faible distance est plus fort (voir figure 6.13). Ceci permettrait alors d'approximer facilement un SDP avec une distribution exponentielle, et la génération de profils synthétiques serait contrôlée par peu de paramètres.

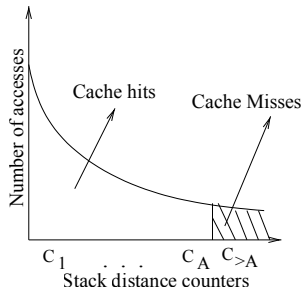


FIGURE 6.13 – *Stack Distance Profile* tel que présenté par Chandra *et al.* [CGKS05].

Malheureusement, ceci est probablement trop simplifié car il est facile de construire un programme qui effectue beaucoup d'accès à une distance donnée. Par exemple, si un programme entre dans une longue boucle qui provoque la séquence suivante : <`... A B C D A B C D A B C D ...`>, alors il y aura un pic dans la distribution à la distance 3.

Nous avons donc visualisé graphiquement la distribution des SDP des programmes que nous analysons afin de mieux comprendre les résultats que nous obtenons ensuite.

Protocole

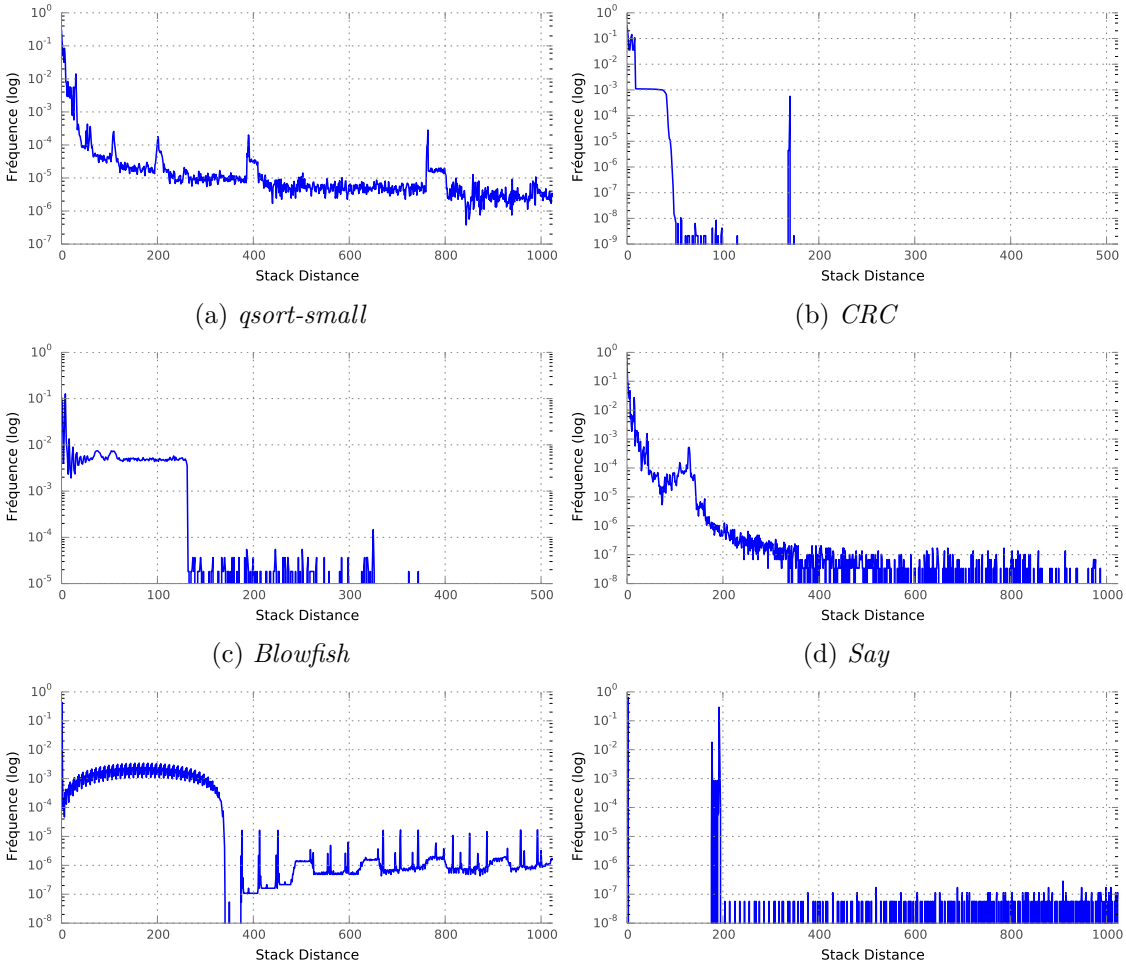
Nous mesurons ici le SDP des programmes du tableau 6.1. Pour cela, nous utilisons notre programme de calcul du SDP présenté dans la partie 6.3.1 et qui se base sur une trace des accès mémoire produit par gem5.

Le calcul du SDP ne dépend que de cette trace et du nombre d'ensembles de cache. Nous nous limitons ici à des SDP générés pour un seul ensemble, les courbes sont similaires avec plusieurs ensembles. Nous étudions les conséquences de l'utilisation d'un SDP pour cache entièrement associatif pour l'évaluation du taux de défauts de cache pour un cache associatif par ensemble dans la partie 6.3.3.

Résultats

Les profils mesurés varient énormément d'un programme à un autre. Pour des raisons de place, il n'est pas possible de fournir toutes les courbes ici, ainsi, nous en proposons une sélection. La figure 6.14 regroupe une sélection de six SDP choisis de sorte à montrer la diversité des profils. Chaque courbe indique la proportion d'accès pour chaque distance. Attention, l'axe des ordonnées est logarithmique.

Pour les programmes *qsort-small* et *Say*, le comportement est relativement proche d'une distribution exponentielle. Mais, parmi l'ensemble des SDP générés, ce comportement n'est pas majoritaire. Nous retrouvons pour certains des plateaux (*CRC*, *Blowfish* et *UD-O2*) pour lesquels il est préférable que la taille du cache soit une valeur plus grande que la distance de la fin du plateau. Enfin, sur certains SDP, nous pouvons observer des pics et



(a) *qsort-small* (b) *CRC*

(c) *Blowfish* (d) *Say*

(e) *UD-O2* (f) *MATMULT-O2*

FIGURE 6.14 – Sélection de six *Stack Distance Profile*.

là encore, une variation de la taille du cache autour de cette valeur peut engendrer un nombre de défauts de cache très différent.

Conclusion

De par le principe de localité, on peut voir pour la plupart des courbes une probabilité plus forte pour que la stack distance soit faible. Cependant, contrairement à ce qu'on pourrait croire, la distribution des distances ne suit pas une distribution classique et le profil des applications est très varié. Nous pouvons notamment voir sur certaines courbes des plateaux et des pics ce qui est synonyme de variations brusques du nombre de défauts de cache lors de variations de la taille du cache (taille physique ou effective à cause d'un partage).

b) Apparition des *cold misses*

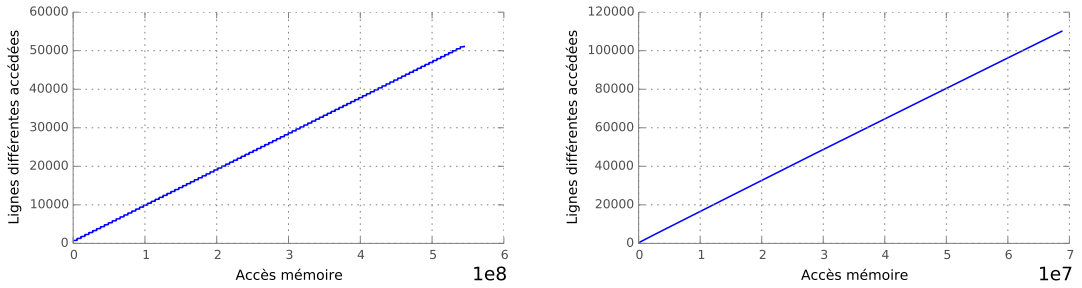
Une source de défauts de cache est liée à l'utilisation de mémoire jamais accédée auparavant. C'est ce qu'on appelle des *cold misses*. Nous voudrions savoir si l'apparition de ces défauts de cache est homogène, regroupée principalement au début du programme, ou si la situation est plus complexe et pour quelle raison.

Protocole

À l'aide de gem5, une trace de toutes les adresses mémoire accédées par chaque programme a été collectée. En utilisant cette trace, il est facile de générer la courbe donnant le nombre de lignes différentes accédées depuis le début du programme en fonction du nombre d'accès mémoire.

Résultats

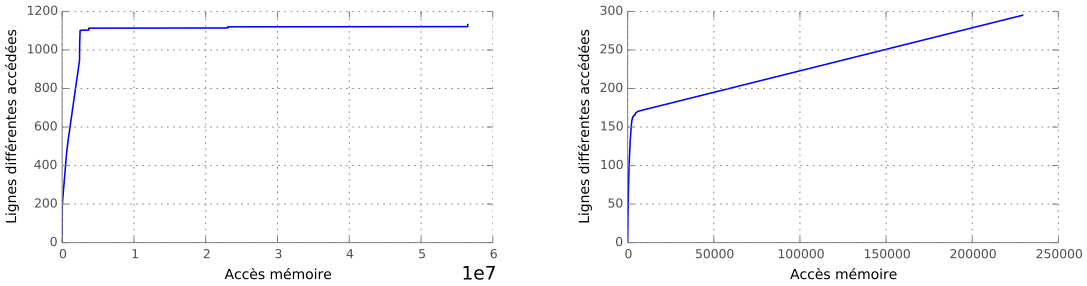
Les programmes qui réutilisent très peu leurs lignes consomment naturellement beaucoup de lignes différentes. On retrouve ainsi des courbes linéaires comme le montre la figure 6.15.



(a) *Susan* (b) *Patricia*

FIGURE 6.15 – Chargement du cache linéaire.

Dans d'autres cas, nous pouvons observer deux phases bien distinctes. La première est courte et correspond sans doute à l'initialisation des données ce qui provoque beaucoup de *cold misses*, puis la seconde phase correspond au comportement normal du programme. La figure 6.16 illustre ceci à travers les courbes obtenues à partir des programmes *qsort-small* et *CRC*. Beaucoup de programmes se comportent de manière similaire à *qsort-small*, avec souvent une première phase encore plus courte.



(a) *qsort-small* (b) *CRC*

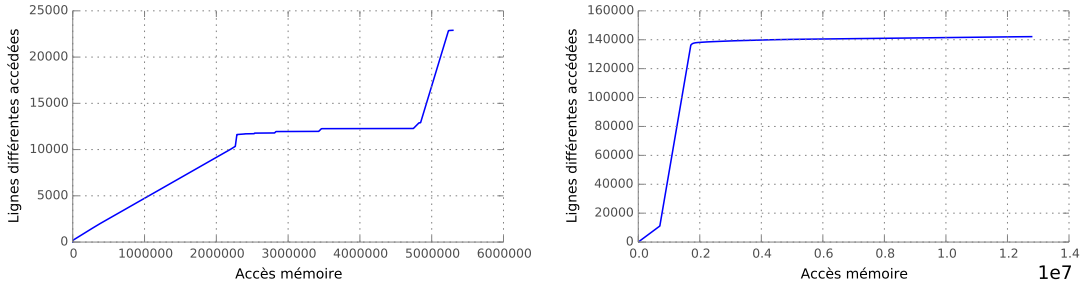
FIGURE 6.16 – Chargement du cache en 2 phases.

Enfin, nous retrouvons parfois une troisième phase (voir figure 6.17) et nous pourrions en avoir davantage. Une analyse de certains programmes nous permet de retrouver dans le code les différentes phases observées.

Conclusion

Les courbes indiquent que l'apparition de *cold misses* varie fortement d'un programme à un autre. Pour certains, l'apparition de *cold misses* est homogène, tandis que pour d'autres, une fois les lignes utilisées chargées au début du programme, il n'y a plus de *cold misses*.

Nous avons aussi remarqué des changements de comportement dans les courbes, liés au changement de la nature du programme pendant l'exécution. Cependant, ces phases



(a) *qsort-small* (b) *COMPRESS-O2*

FIGURE 6.17 – Chargement du cache en 3 phases.

sont détectables et nous pourrions ainsi envisager de découper les programmes en sous-programmes s'exécutant de manière consécutive si cela permet d'obtenir des comportements plus prévisibles et donc plus facilement estimables par les modèles. Cette possibilité n'a pas été mise en œuvre mais fait partie des améliorations pouvant être facilement ajoutées.

c) Commentaire sur les expérimentations utilisant LITMUS^{RT}

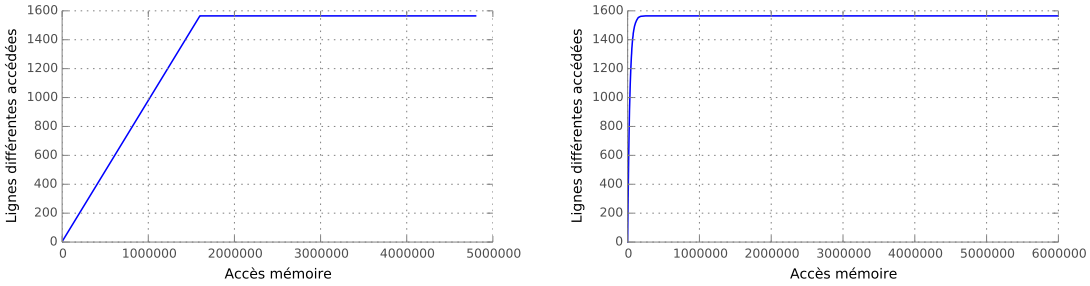
Des études sur des stratégies d'ordonnancement *cache-aware* ont été réalisées sur LITMUS^{RT}. Ces études reposent sur l'exécution de tâches simulant des accès mémoire. Deux types de profil d'accès ont été utilisés pour les évaluations [Cal09] : séquentiel et aléatoire. Ces programmes réalisent des accès mémoire sur un tableau de WSS octets (modélisant le *Working Set*).

Dans le cas d'accès séquentiels, le programme accède à toutes les cases d'un tableau d'entiers successivement puis recommence à trois reprises. Lors du premier parcours du tableau, un défaut de cache de type *cold miss* est provoqué tous les 64 octets. Lors des deux autres parcours du tableau, les défauts de cache font maintenant place à des accès se faisant à une distance égale au nombre de lignes utilisées par le tableau. En prenant en considération le code de la boucle *for* qui est à l'origine d'une grande part des accès mémoire réalisés, on obtient expérimentalement un SDP pour lequel plus de 99% des accès se font à une distance strictement inférieure à 3.

Dans le cas d'accès aléatoires, le programme qui s'exécute réalise $3 \cdot \text{WSS}$ accès aux cases du tableau de manière aléatoire. La distance entre les accès est donc uniformément aléatoire entre 0 et WSS. En raison de la boucle *for* et du code qui génère l'indice aléatoire, il y a encore de nombreux accès à une distance inférieure à 3. Le SDP que nous obtenons expérimentalement regroupe 90% des accès à des distances strictement inférieures à 3 puis le nombre d'accès aux WSS distances suivantes est uniforme.

La figure 6.18 montre la courbe de chargement des deux types d'accès. Dans le cas d'un accès séquentiel, les *cold misses* sont engendrés de manière bien régulière tout au long du premier parcours. Alors que dans le cas d'un accès aléatoire, les premiers accès ont une grande chance de concerner de nouvelles lignes puis cette probabilité décroit selon une loi exponentielle.

Les évaluations qui ont été conduites reposent donc sur des tâches aux comportements mémoire très singuliers. Intuitivement, nous comprenons que des tâches aux accès séquentiels ne sont pas très sensibles à un partage de cache notamment. Au contraire, les



(a) Accès séquentiels (b) Accès aléatoires

FIGURE 6.18 – Chargement du cache pour les tâches générées artificiellement.

conséquences d'un partage de cache entre deux tâches aux accès aléatoires ne présentent pas de changements brusques qui sont difficiles à caractériser. Certaines études évaluent notamment des ordonnanceurs qui tiennent compte du WSS des tâches pour prendre leur décision [ACD06], or le WSS est, dans ces études, un paramètre important pour la génération des tâches utilisées.

6.3.3 Caches N-Way et fully-associative

L'utilisation de caches associatifs par ensembles à N voies est un bon compromis entre l'utilisation d'un cache entièrement associatif qui devient trop lent dès que la taille du cache est trop importante et un cache direct qui provoque un taux de défauts de cache plus important. Il est fréquent de lire qu'un cache dont l'associativité est supérieure à 8 donne des résultats très proches de ceux obtenus avec un cache entièrement associatif. Agarwal *et al.* montrent dans leurs études que les effets bénéfiques de l'augmentation de l'associativité diminuent rapidement [AHH89].

L'objectif de cette étude est de voir si de tels résultats sont reproduits par nos programmes. Ceci nous permettrait de savoir si l'utilisation de modèles pour caches entièrement associatifs est pertinent pour des caches N-way, et à l'inverse si nous pouvons estimer les défauts de cache pour une associativité plus faible dans le but d'alléger les calculs.

Protocole

Nous utilisons ici le simulateur de cache présenté dans la section 6.3.1 et qui se base sur les traces des accès mémoire générés par gem5. Le nombre de défauts de cache a été collecté pour les programmes présentés dans le tableau 6.1 en faisant varier la taille du cache (16 Kio, 32 Kio, 64 Kio et 128 Kio) et l'associativité (1, 2, 4, 8 et entièrement associatif). La taille d'une ligne de cache est fixée à 64 octets.

Résultats

Pour des raisons de place, nous ne présentons pas l'ensemble des résultats. Cependant, les comportements sont assez similaires entre les différents programmes. Les résultats obtenus pour *Dijkstra-large* sont représentatifs du comportement observé sur la majorité des programmes que nous avons évalués. La figure 6.19 montre pour chaque associativité, le taux de défauts de cache en fonction de la taille du cache.

Nous constatons un gain important en passant d'un cache direct à un cache à deux voies, puis encore à nouveau un gain non négligeable lors du passage à quatre voies. Cependant, à partir de 4 voies, les résultats se confondent avec les résultats obtenus pour des associativités supérieures.

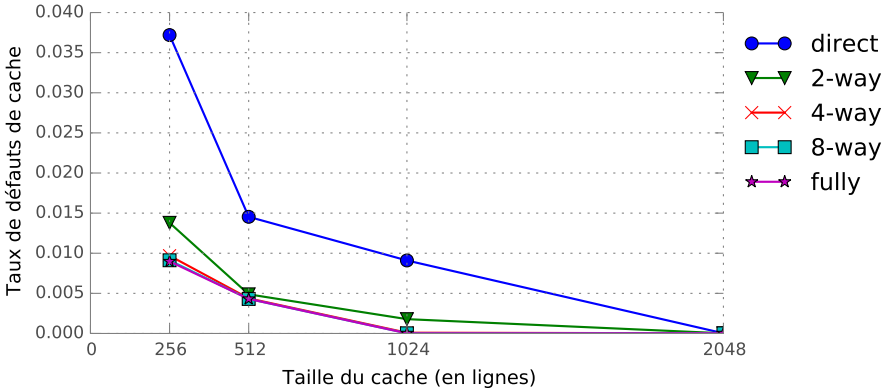


FIGURE 6.19 – Taux de défauts de cache pour le programme *Dijkstra-large* en fonction de la taille du cache, pour différentes valeurs d'associativité.

Pour certains programmes, les gains ne sont pas aussi visibles : c'est le cas en particulier lorsque les défauts de cache sont principalement des *cold misses* et que le cache est assez grand pour ne pas faire de *capacity misses*. Pour d'autres, un passage à 8 voies offre un gain supplémentaire intéressant lorsque le cache est petit. Mais dans tous les cas, nous n'avons pas repéré de différence majeure entre les taux de défauts de cache pour une associativité de 8 et pour un cache entièrement associatif.

Il existe malgré tout des cas plus rares où un cache direct permet de faire moins de défauts de cache qu'un cache entièrement associatif. Le programme *UD-O2* provoque en effet plus de défauts de cache avec un cache entièrement associatif lorsque le cache est suffisamment petit. Au delà d'une certaine taille de cache, la situation s'inverse. La figure 6.20 montre les résultats obtenus pour le programme *UD-O2*. Ce comportement a également été observé pour le programme *CNT* pour un cache de 2048 lignes. Ceci montre que l'algorithme LRU fait parfois de mauvais choix et qu'un remplacement moins évolué peut donner dans certains cas précis de meilleurs résultats.

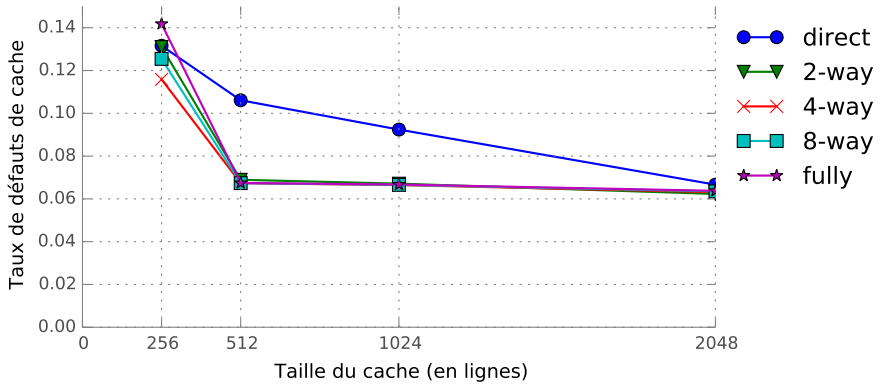


FIGURE 6.20 – Taux de défauts de cache pour le programme *UD-O2* en fonction de la taille du cache, pour différentes valeurs d'associativité.

Conclusion

Il n'est pas possible de dire que le passage d'un cache direct à un cache associatif permet de réduire significativement le nombre de défauts de cache. Généralement le passage à un cache associatif permet effectivement de réduire le nombre de défauts de cache, mais l'écart dépend des programmes et de la taille du cache. Nous avons également vu dans

des cas rares, qu'un cache direct peut provoquer moins de défauts de cache qu'un cache associatif.

En revanche, dans toutes les évaluations, nous n'avons pas remarqué de différences importantes entre les résultats obtenus pour des caches 8-ways et des caches entièrement associatifs. Ceci conforte donc l'idée répandue qu'il est inutile de fabriquer des caches dont l'associativité dépasse 8 ou 16 car les résultats sont les mêmes que pour un cache entièrement associatif.

Il est ainsi possible de simuler des caches entièrement associatifs à la place de caches dont l'associativité est plus faible, et inversement, sans changer fondamentalement les résultats.

6.3.4 Évaluation du comportement des tâches en isolation

a) Estimation des défauts de cache

Dans la partie 6.2.1, nous avons vu que le taux de défauts de cache peut être calculé à partir du SDP. Nous vérifions ici que les SDP que nous avons construit nous permettent effectivement de retrouver le taux de défauts de cache.

Nous comparons ici les résultats obtenus à l'aide de gem5 pour des caches 8-ways de différentes tailles, aux résultats calculés à partir des SDP des programmes. Nous utilisons ici des SDP calculés pour un seul ensemble (équation 6.7) et des SDP calculés pour un nombre d'ensembles qui correspond à celui du cache utilisé (équation 6.6).

Dans le cas où le SDP est calculé pour le même nombre d'ensembles que dans le cache utilisé, nous ne devrions pas avoir d'écart puisque la formule donne un résultat exact. En revanche, dans le cas où nous utilisons un SDP généré pour un seul ensemble, nous devrions trouver une erreur similaire à celle obtenue dans l'étude précédente sur l'associativité.

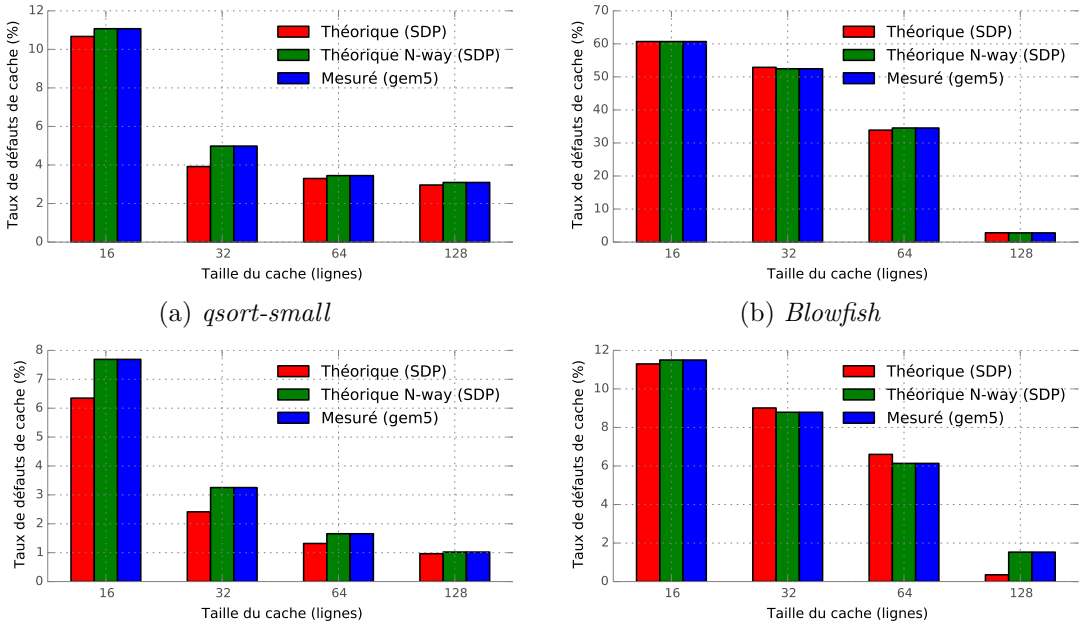
Protocole

L'ensemble des programmes ont été exécutés avec gem5 en faisant varier la taille des caches entre 16, 32, 64 et 128 lignes, avec une associativité de 8. Le nombre d'ensembles est donc respectivement de 2, 4, 8 et 16. L'utilisation d'un cache de 16 lignes (1 Kio) peut paraître très faible, mais permet d'augmenter le taux de défauts de cache et donc éventuellement mettre en difficulté nos modèles. Le nombre de défauts de cache est directement fourni par gem5.

Afin d'obtenir les résultats calculés à partir des SDP, nous avons réutilisé les SDP générés pour un seul ensemble et qui ont été présentés précédemment. Nous les avons complété par la génération de SDP calculés pour plusieurs ensembles (voir section 6.3.1).

Résultats

La formule que nous étudions étant exacte, il était attendu que les résultats obtenus à l'aide des SDP pour plusieurs ensembles soient identiques aux valeurs mesurées avec gem5. En revanche, nous observons un écart lorsque nous utilisons le SDP généré pour un seul ensemble alors que le cache réel est associatif par ensembles de 8 voies (voir figure 6.21).



(a) *qsort-small* (b) *Blowfish*

(c) *JPEG* (d) *Patricia*

FIGURE 6.21 – Défauts de cache en fonction de la taille du cache.

Conclusion

L'utilisation d'un SDP différent pour chaque taille de cache permet de déterminer précisément le nombre de défauts de cache. Cependant, le calcul d'un SDP nécessite la trace des accès mémoire d'un programme (jusqu'à plusieurs giga par trace dans notre cas) et il faut du temps pour calculer le SDP (entre quelques secondes et plusieurs heures en fonction de la distance moyenne des accès et de leur nombre). Or, l'un de nos objectifs est de pouvoir modifier facilement les caractéristiques des caches ce qui rentre en contradiction. Ainsi, nous avons voulu vérifier la possibilité d'utiliser un SDP généré pour un cache entièrement associatif.

Rappelons cependant que notre objectif n'est pas de reproduire fidèlement le comportement d'un programme mais simplement de reproduire des comportements possibles. Ainsi, même si les résultats indiquent qu'utiliser un unique SDP pour toutes les tailles de cache implique effectivement une erreur dans l'estimation, ceci n'est pas réellement un problème pour l'utilisation que nous souhaitons en faire.

b) Modèle CPI

Dans l'étude précédente, nous avons évalué le nombre de défauts de cache. Ce taux de défauts de cache est maintenant intégré dans le calcul du CPI puis multiplié par le nombre d'instructions pour obtenir la durée d'exécution (équations 6.2 et 6.3).

En plus de vérifier l'équation 6.3, nous voudrions notamment savoir si l'erreur liée à l'associativité que nous observons au niveau du taux de défauts de cache est atténuée en considérant maintenant la durée d'exécution des programmes.

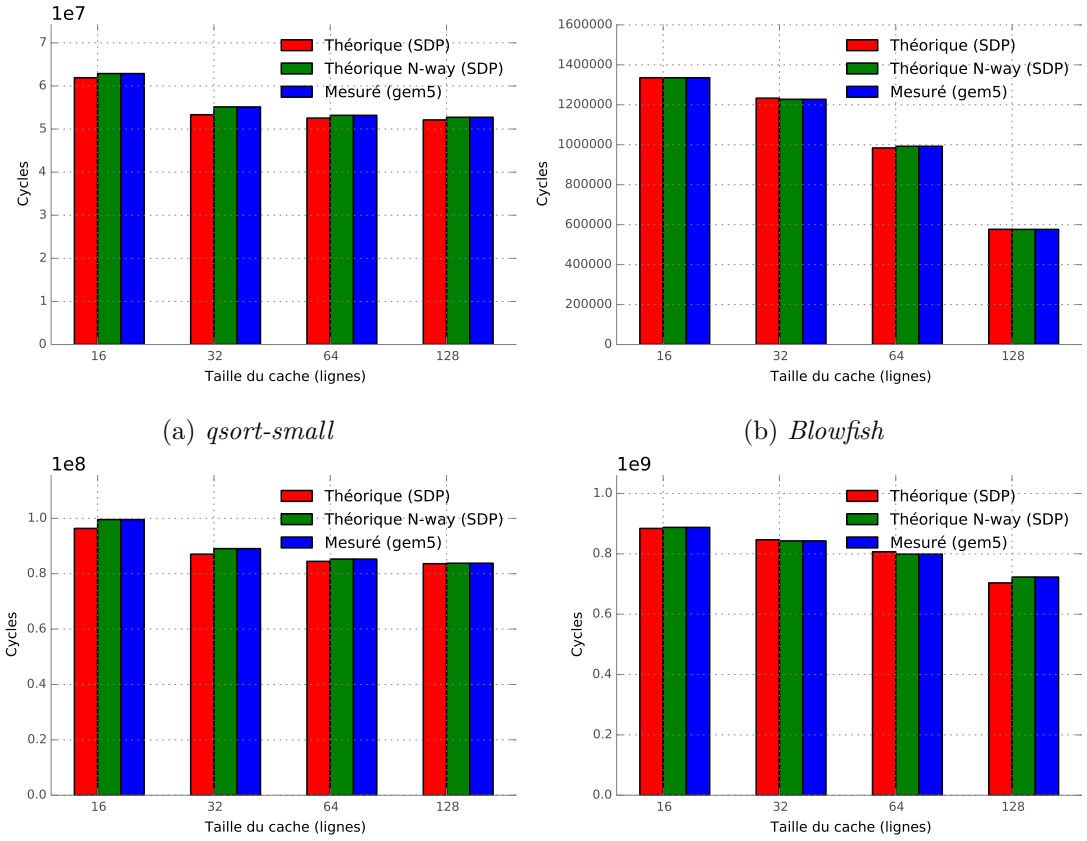
Protocole

Les expérimentations présentées ici se basent sur celles de l'étude précédente concernant les défauts de cache. Nous avons simplement collecté les durées d'exécution en plus.

L'équation 6.4 nous permet de déterminer la valeur de *base_cpi* à partir d'une exécution sur gem5. En utilisant cette valeur et le taux de défauts de cache calculé à partir des SDP pour un seul ensemble, nous en déduisons le CPI et donc la durée d'exécution.

Résultats

La figure 6.22 montre la durée d'exécution des programmes obtenus à l'aide de gem5, et l'estimation basée sur l'équation 6.2. Comme nous pouvons le voir, les estimations sont très proches des valeurs obtenues en pratique.



(a) *qsort-small* (b) *Blowfish*

(c) *JPEG* (d) *Patricia*

FIGURE 6.22 – Durée d'exécution en fonction de la taille du cache.

Conclusion

Ce qui nous importe réellement est l'estimation des durées d'exécution et non le nombre de défauts de cache. Nous voyons ici que les erreurs dans l'estimation du nombre de défauts de cache ont un impact atténué dans le calcul de la durée d'exécution des programmes. En examinant l'équation du CPI, on comprend que plus le nombre d'accès par instruction (API) est important et plus l'erreur d'estimation du nombre de défauts de cache aura d'impact sur la durée d'exécution.

6.3.5 Partage de cache

Nous évaluons maintenant les modèles FOA, SDC et Babka. Pour cela, nous exécutons en parallèle deux programmes sur un cache partagé puis nous collectons leur CPI. Les résultats obtenus sont alors comparés à ceux prédits par les modèles.

Protocole

Les programmes *UD-O2*, *CNT-O2*, *Dijkstra-large*, *COMPRESS-O2* et *edge* ont été exécutés en boucle pendant une durée fixe. Toutes les combinaisons de deux programmes ont été évaluées sur une architecture composée de deux cœurs, un cache L1 privé par cœur de 2 Kio et un cache L2 partagé de 16 Kio puis 32 Kio (respectivement 256 puis 512 lignes). L'associativité des caches est de 8.

Le nombre d'instructions exécutées est collecté pour chaque programme pour toutes les évaluations, et nous en déduisons le CPI.

Le CPI est également calculé à l'aide des modèles FOA, SDC et Babka pour pouvoir comparer les résultats. Le SDP utilisé a été calculé pour un seul ensemble.

Résultats

Les tableaux présentés ci-dessous montrent les CPI calculés par Gem5 et par les modèles FOA et Babka. Nous avons écarté SDC rapidement à cause de résultats systématiquement moins bons que ceux produits par FOA et Babka. Ainsi nous préférons nous concentrer sur les modèles ayant produit les meilleurs résultats.

Le tableau 6.2, donnant les résultats pour *UD-O2* partagé avec les autres programmes, correspond aux évaluations où le CPI a le plus varié et c'est aussi le cas où les écarts sont les plus marqués. L'erreur observée baisse lorsque les interférences sont plus faibles, on le voit par exemple pour *UD-O2* en augmentant la taille du cache (tableau 6.3). La figure 6.23 offre une représentation graphique du ralentissement des tâches tel qu'il est estimé par les modèles, par rapport au ralentissement observé à l'aide de gem5 (données du tableau 6.2).

UD-O2	Gem5	FOA		Babka	
exécuté avec	CPI	CPI	Écart Rel.	CPI	Écart Rel.
-	9.08	9.08	0.00 %	9.08	0.00 %
UD-O2	13.61	13.47	-1.04 %	12.74	-6.41 %
CNT-O2	9.39	11.08	17.97 %	10.44	11.21 %
Dijkstra-large	11.84	13.53	14.27 %	11.27	-4.82 %
COMPRESS-O2	11.60	13.00	12.07 %	12.48	7.54 %
edge	12.02	13.00	8.19 %	12.47	3.77 %

Tableau 6.2 – Exécution de *UD-O2* en parallèle avec un autre programme sur un cache partagé de 16 Kio.

Bien que sur cet exemple Babka produit des résultats meilleurs que ceux de FOA, ceci n'est pas toujours le cas. Le tableau 6.4 montre les résultats obtenus pour *Dijkstra-large* pour un cache de 32 Kio.

Lorsque les variations sont plus faibles, les résultats sont plus précis comme le montre le tableau 6.5 concernant *edge* pour un cache de 32 Kio.

UD-O2	Gem5	FOA		Babka	
exécuté avec	CPI	CPI	Écart Rel.	CPI	Écart Rel.
-	8.04	8.04	0.00 %	8.04	0.00 %
UD-O2	8.92	9.29	4.20 %	9.59	7.54 %
CNT-O2	8.06	8.04	-0.17 %	8.15	1.14 %
Dijkstra-large	8.10	9.72	19.94 %	8.12	0.22 %
COMPRESS-O2	8.19	8.89	8.57 %	9.46	15.50 %
edge	8.20	8.97	9.29 %	9.48	15.57 %

Tableau 6.3 – Exécution de *UD-O2* en parallèle avec un autre programme sur un cache partagé de 32 Kio.

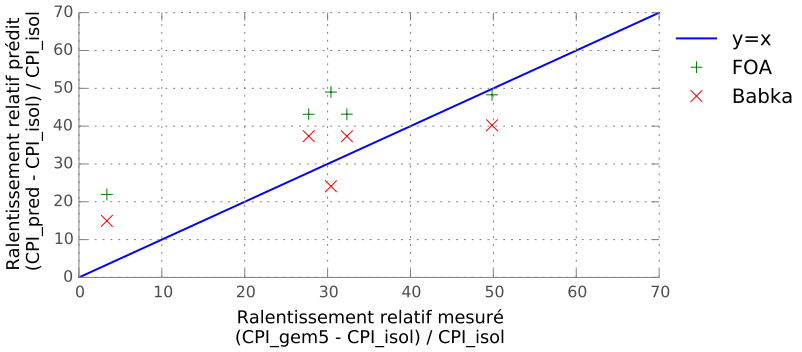


Figure 6.23 – Ralentissement relatif prédit en fonction du ralentissement relatif observé pour *UD-O2* sur un cache partagé de 16 Kio.

Dijkstra-large	Gem5	FOA		Babka	
exécuté avec	CPI	CPI	Écart Rel.	CPI	Écart Rel.
-	4.75	4.75	0.00 %	4.75	0.00 %
UD-O2	5.54	6.07	9.63 %	6.22	12.18 %
CNT-O2	4.81	4.83	0.27 %	5.30	10.04 %
Dijkstra-large	6.00	6.12	1.93 %	5.26	-12.34 %
COMPRESS-O2	5.33	5.74	7.70 %	6.30	18.22 %
edge	5.45	5.74	5.34 %	6.25	14.72 %

Tableau 6.4 – Exécution de *Dijkstra-large* en parallèle avec un autre programme sur un cache partagé de 16 Kio.

La figure 6.24 résume l'ensemble des valeurs mesurées pour un cache de 16 Kio. Nous pouvons voir que FOA a tendance à légèrement surestimer le ralentissement subi par les programmes alors que Babka sous-estime presque autant qu'il surestime.

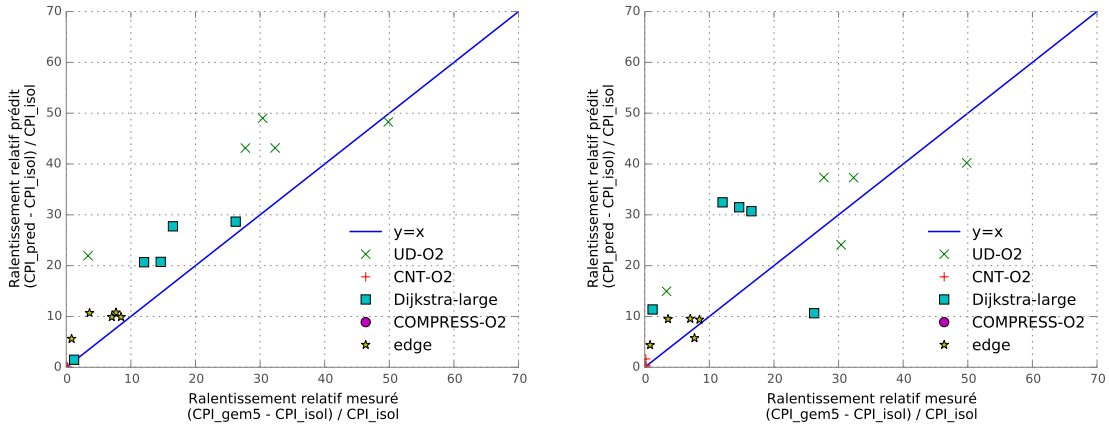
Conclusion

Les expérimentations n'ont pas permis de départager véritablement FOA et Babka, mais le calcul de Babka est beaucoup plus lent. Ainsi, pour la suite, nous avons décidé de retenir FOA. Dans les deux cas, même s'il y a des erreurs dans l'estimation, nous constatons que les effets des caches sur le CPI sont reproduits.

Nos résultats indiquent aussi que dès que le cache est suffisamment grand, les effets sur le CPI diminuent fortement. Ceci signifie que la somme des *Working Set Size* des programmes

edge	Gem5	FOA		Babka	
exécuté avec	CPI	CPI	Écart Rel.	CPI	Écart Rel.
-	6.74	6.74	0.00 %	6.74	0.00 %
UD-O2	6.99	7.46	6.83 %	7.38	5.67 %
CNT-O2	6.80	7.12	4.74 %	7.04	3.51 %
Dijkstra-large	7.26	7.47	2.86 %	7.13	-1.80 %
COMPRESS-O2	7.22	7.41	2.64 %	7.39	2.34 %
edge	7.31	7.41	1.30 %	7.38	0.83 %

TABLEAU 6.5 – Exécution de *edge* en parallèle avec un autre programme sur un cache partagé de 16 Kio.



(a) FOA (b) Babka

FIGURE 6.24 – Résumé de toutes les valeurs mesurées pour un cache de 16 Kio.

est inférieure à la taille du cache. Ces résultats sont donc à rapprocher de ceux que nous verrons dans la partie 6.3.6 et qui montre le WSS moyen des différents programmes.

6.3.6 Coûts des préemptions

Dans un premier temps, nous nous intéressons au chargement du cache par les applications dans le but de pouvoir estimer l'occupation des caches. Ensuite nous verrons l'impact d'une ou plusieurs préemptions sur le nombre de défauts de cache.

a) Chargement moyen

Dans la section 6.3.2, nous nous sommes intéressé à l'apparition de *cold misses*. Ces défauts de cache vont naturellement provoquer un remplissage du cache mais cela ne nous indique pas si les lignes chargées seront à nouveau utilisées et donc rechargées suite à une préemption. Nous cherchons donc ici à caractériser la quantité mais aussi la vitesse du rechargement d'un cache suite à une préemption. Pour cela, nous nous basons sur les modèles présentés dans la partie 6.2.4.

Les valeurs que nous mesurons ici correspondent aussi au *Working Set Size* moyen en fonction du nombre de références tel que défini par Denning (voir section 2.5.1).

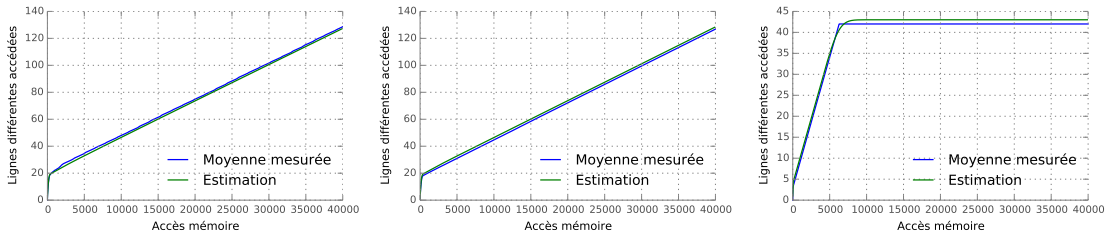
Protocole

La méthode utilisée ici est similaire à celle utilisée pour l'apparition des *cold misses* à la différence que les structures de données sont réinitialisées tous les 40 000 accès mémoire. Ceci nous permet donc d'évaluer le chargement du cache initialement vide à partir de nombreux points différents et pour une durée de 40 000 accès. Nous évaluons par la suite la moyenne obtenue à partir de tous ces chargements.

Le SDP des programmes est utilisé afin d'estimer le chargement à l'aide du modèle dérivé du modèle de Babka. Cependant, pour des raisons de performances, nous avons limité la taille de la matrice à 300x300 ce qui provoque un palier du nombre de lignes chargées à 300.

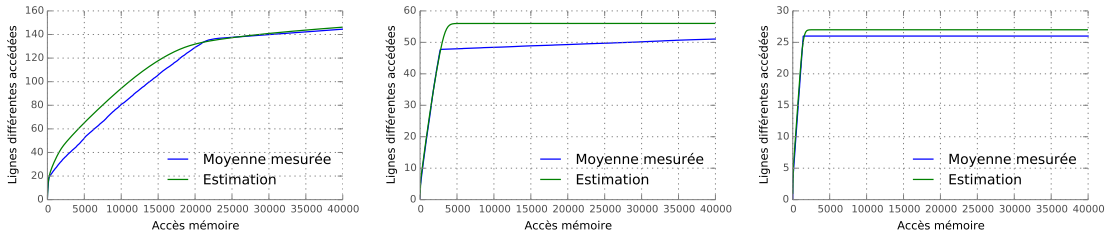
Résultats

Les résultats obtenus par le modèle de Babka sont très proches des valeurs moyennes mesurées. Nous pouvons le voir sur les figures 6.25 et 6.26. Dans d'autres cas, la taille limite de la matrice ne permet pas de suivre le chargement au complet (figure 6.27). Enfin, comme le montre la figure 6.28 il existe des cas, en nombre minoritaire ($\approx 10\%$), où les résultats obtenus par le modèle sont éloignés de ceux mesurés.



(a) *SHA-small* (b) *SHA-large* (c) *NS*

FIGURE 6.25 – Exemples où les résultats produits par le modèle coïncident parfaitement aux valeurs mesurées en moyenne.

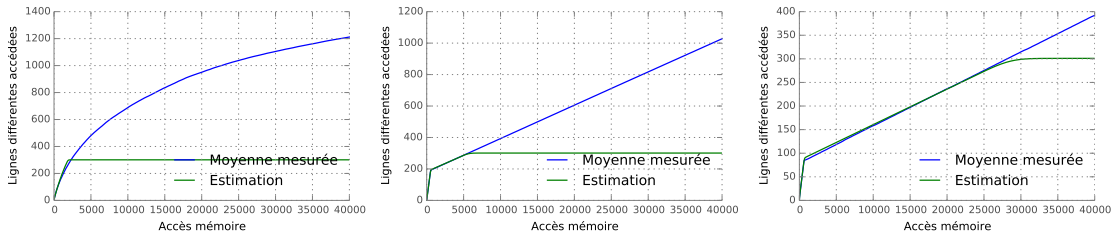


(a) *Say* (b) *FIR* (c) *COVER*

FIGURE 6.26 – Exemples où les résultats produits par le modèle coïncident bien aux valeurs mesurées en moyenne.

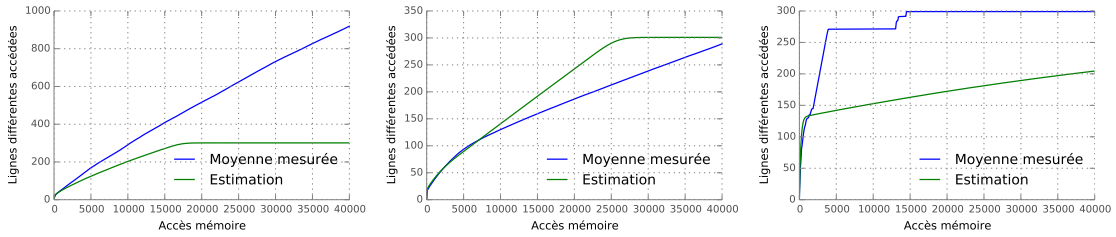
Le modèle de Babka n'est pas prévu pour des caches entièrement associatifs à cause de la taille de la matrice nécessaire. Sans augmenter la taille de la matrice, il est cependant possible d'augmenter facilement la limite supérieure en utilisant un SDP calculé pour plus d'un ensemble (voir section 6.1.2) puis de multiplier les résultats par le nombre d'ensembles.

La figure 6.29 montre les résultats obtenus en utilisant des SDP pour 4 ensembles. Nous pouvons voir que lorsque le nombre de lignes de cache utilisées est faible, une petite erreur est introduite mais reste raisonnable. En revanche, en augmentant davantage le nombre



(a) *edge* (b) *MATMULT-O2* (c) *MATMULT*

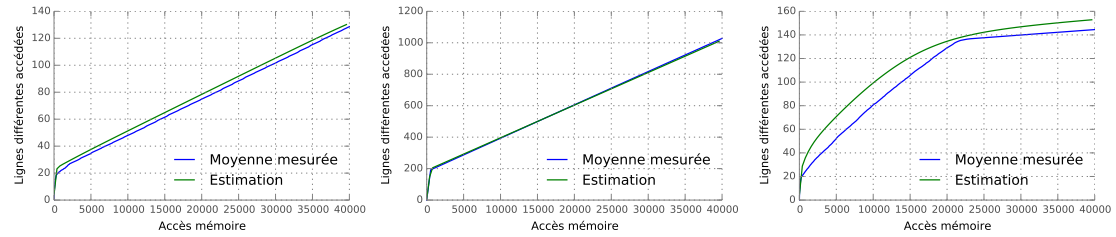
FIGURE 6.27 – Exemples où le modèle donne de bons résultats jusqu'à ce que le nombre de lignes atteigne 300.



(a) *qsort-small* (b) *FFT-small* (c) *Blowfish*

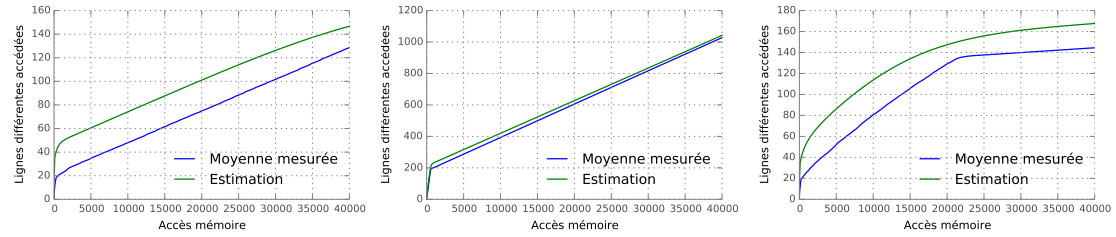
FIGURE 6.28 – Exemples où le modèle ne produit pas de bons résultats.

d'ensembles utilisés pour le calcul du SDP, l'erreur devient de plus en plus importante (voir figure 6.30). Ceci s'explique par la perte d'information causée par l'augmentation artificielle du nombre d'ensembles dans le SDP utilisé.



(a) *SHA-small* (b) *MATMULT-O2* (c) *Say*

FIGURE 6.29 – Résultats obtenus à base de SDP calculés pour 4 ensembles.



(a) *SHA-small* (b) *MATMULT-O2* (c) *Say*

FIGURE 6.30 – Résultats obtenus à base de SDP calculés pour 16 ensembles.

Conclusion

À travers ces expérimentations, nous avons montré qu'il était possible d'obtenir une bonne estimation du chargement moyen du cache en fonction du nombre d'accès mémoire. Les courbes présentées ci-dessus montrent bien le comportement très varié des programmes.

Ainsi, nos autres tentatives visant à évaluer le chargement à l'aide de distributions classiques (telle qu'une distribution exponentielle) ont échouées.

Bien que le modèle donne des résultats très proches de ceux obtenus expérimentalement, ceci n'est qu'une moyenne et en réalité la vitesse de chargement du cache fluctue fortement en fonction de la position dans le programme. Ainsi, il n'est peut-être pas réaliste de supposer un rechargement du cache identique à tout moment dans le programme. Pour une simulation, il serait peut-être plus judicieux de générer des rechargements aléatoires en tenant compte de l'écart type qui est également calculable à partir de la chaîne de Markov du modèle (équation 6.18).

b) Étude de l'impact du lieu de la préemption

Dans cette étude, nous simulons l'impact d'une unique préemption en matière de défauts de cache supplémentaires par rapport à une exécution sans préemption. Pour le moment, une seule préemption est faite mais nous nous intéressons à l'impact de cette préemption en fonction de sa position dans l'exécution du programme.

Protocole

Pour cela, nous simulons un cache entièrement associatif que nous vidons entièrement ou partiellement à un moment donné afin de simuler la perturbation causée par une préemption. Suite à cette perturbation du cache, le nombre total de défauts de cache augmente. La différence entre l'exécution sans préemption et avec une préemption correspond aux défauts de cache supplémentaires liés à la préemption.

Chaque programme est préempté une seule fois mais à une position différente à chaque fois. Plus précisément, nous expérimentons une préemption tous les $M/20$ accès au cache, avec M le nombre total d'accès. Nous réalisons cette expérimentation sur un cache de 512 lignes de 64 octets (32 Kio).

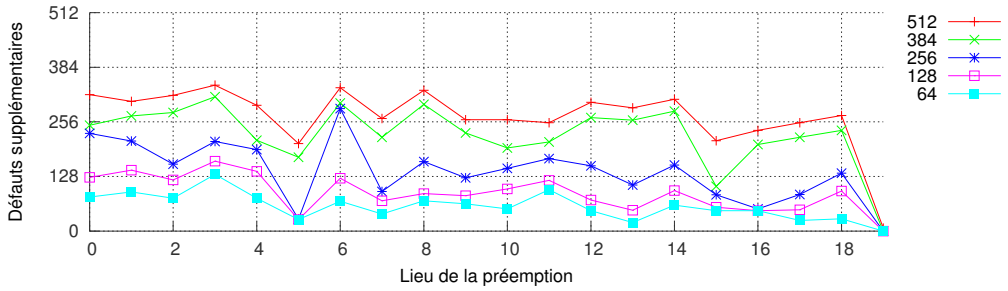
Résultats

La figure 6.31 présente les résultats obtenus pour les programmes *Dijkstra-Large*, *UD-O2* et *COMPRESS-O2*. Ces programmes ont été retenus pour montrer les différents comportements observés.

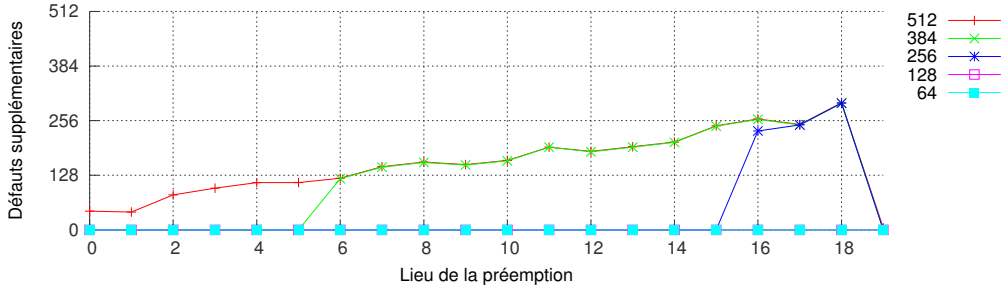
Dans le cas de *Dijkstra-Large*, nous constatons d'une part que plus la perturbation est importante et plus le nombre de défauts de cache augmente. Généralement le nombre de défauts de cache engendrés est inférieur à la taille de la perturbation mais il existe des cas où une perturbation de 128 ou 256 lignes provoque un nombre de défauts de cache supérieur (conforme aux explications de Liu). D'autre part, les conséquences d'une perturbation restent relativement stable en changeant le lieu de la préemption.

Au contraire, pour le programme *UD-O2*, nous constatons qu'une perturbation peut ne provoquer aucun défaut de cache supplémentaire si les lignes qui seront réutilisées ne sont pas touchées. Nous voyons également que le seuil à partir duquel une perturbation engendre des défauts de cache supplémentaire varie en fonction du moment où se produit la perturbation.

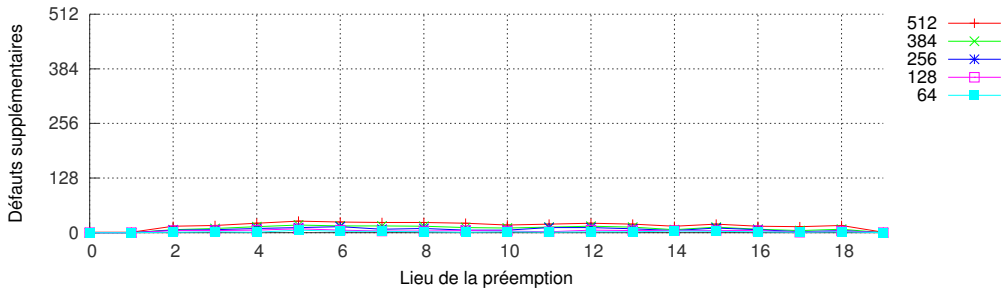
Enfin, dans le cas du programme *COMPRESS-O2* que les perturbations de cache ne provoquent pas beaucoup de défauts de cache supplémentaires. Ceci est le signe que soit *COMPRESS-O2* n'utilise pas beaucoup de lignes différentes, soit il y a dans tous les cas beaucoup de défauts de cache.



(a) *Dijkstra-Large*



(b) *UD-O2*



(c) *COMPRESS-O2*

FIGURE 6.31 – Défauts de cache supplémentaires suite à une préemption, en fonction du moment de la préemption.

Conclusion

Nous constatons que les effets d'une perturbation du cache suite à une préemption peut, pour certains programmes, varier énormément en fonction du moment où la perturbation est faite. Par conséquent, cette perturbation est difficile à estimer. Nous étudions dans la section suivante si nous sommes cependant en mesure d'évaluer le nombre de défauts de cache engendrés en moyenne.

c) Multiples préemptions

Dans cette étude, plusieurs préemptions sont faites, à des intervalles constants. L'objectif est de déterminer le coût moyen d'une préemption en fonction de la taille de la perturbation et en faisant varier le nombre de préemptions.

Protocole

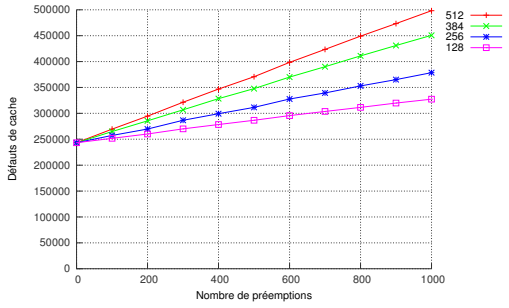
Pour ce faire, nous réutilisons les traces des accès mémoires des programmes ainsi que le simulateur de cache. Pour rappel, ce simulateur nous permet de perturber l'état du cache à tout moment.

Nous faisons varier le nombre de préemptions entre 0 et 1000. Nous obtenons donc pour chaque programme le nombre de défauts de cache en fonction du nombre de préemptions. Cette expérimentation est reproduite pour des perturbations différentes.

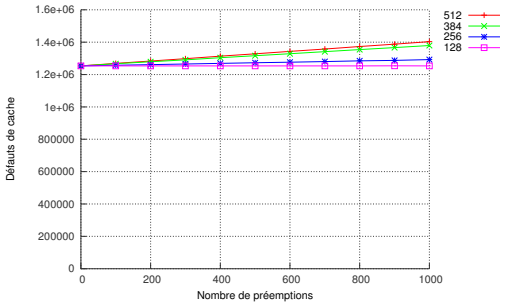
Le cache utilisé contient 512 lignes de 64 octets, soit 32 Kio.

Résultats

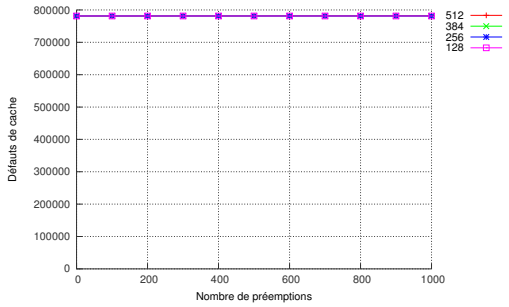
Un millier de préemptions représente, pour les programmes que nous manipulons ici, une préemption toutes les 0.1 à 1 ms (en fonction du programme). Ceci est suffisant pour laisser le cache se remplir à nouveau de la plupart des lignes que le programme préempté utilise. Par conséquent, le nombre de défauts de cache supplémentaires engendrés par les préemptions est proportionnel à leur nombre. Sur les courbes, ceci se traduit par une droite dont la pente est égale au nombre moyen de défauts de cache causés par une préemption (voir figure 6.32 et tableau 6.6). Ces résultats correspondent à la moyenne des valeurs obtenues au cours de la précédente étude.



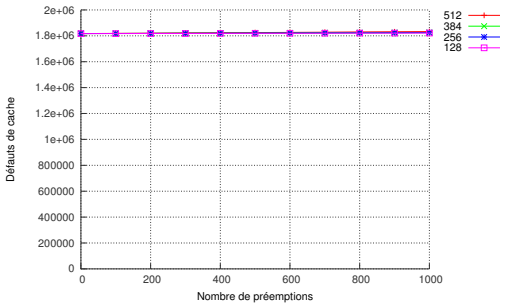
(a) *Dijkstra-Large*



(b) *UD-O2*



(c) *CNT-O2*



(d) *COMPRESS-O2*

FIGURE 6.32 – Nombre de défauts de cache en fonction du nombre de préemptions.

Perturbation	Pente			
	Dijkstra-Large	UD-O2	CNT-02	COMPRESS-O2
512	255.367	149.141	0.932	15.668
384	207.828	125.331	0	8.864
256	135.634	38.349	0	7.189
128	84.587	0.241	0	4.255

TABLEAU 6.6 – Nombre de défauts de cache moyen engendrés par une préemption, en fonction de la taille de la perturbation.

Les tableaux 6.7 et 6.7 donnent les estimations calculées à l'aides modèles 1 et 2 présentés dans la section 6.2.4. Le modèle proposé par Liu n'a pas été évalué car son temps de calcul est trop lent pour l'utilisation que nous souhaitons en faire dans SimSo. Nous constatons

que le modèle 1 se comporte particulièrement mal dans le cas où le cache est entièrement vidé. Ceci s'explique par l'invalidité de l'hypothèse selon laquelle la probabilité de faire un accès à une ligne qui s'est faite exclure reste invariée. Le second modèle est trop optimiste (jusqu'à 10 fois pour *Dijkstra-large.*) mais les tendances sont néanmoins respectées.

Perturbation	Pente			
	Dijkstra-Large	UD-O2	CNT-02	COMPRESS-O2
512	509.795	477.479	479.999	440.09
384	30.910	128.462	0	1.655
256	1.197	19.035	0	0.737
128	0.270	0.012	0	0.186

Tableau 6.7 – Estimation par le modèle n°1 du nombre de défauts de cache moyen engendrés par une préemption, en fonction de la taille de la perturbation.

Perturbation	Pente			
	Dijkstra-Large	UD-O2	CNT-02	COMPRESS-O2
512	25.820	84.599	0.937	5.221
384	13.727	70.729	0	1.390
256	1.760	21.250	0	1.115
128	0.951	0.043	0	0.657

Tableau 6.8 – Estimation par le modèle n°2 du nombre de défauts de cache moyen engendrés par une préemption, en fonction de la taille de la perturbation.

Conclusion

Grâce à cette expérimentation, nous voyons que les lignes qui sont utiles sont rapidement rechargées dans le cache et que les perturbations des préemptions successives sont alors presque indépendantes. Ainsi, le nombre de défauts de cache est proportionnel au nombre de préemptions (à perturbation identique), dans l'intervalle d'étude que nous avons considéré et qui nous semble raisonnable.

Nous constatons également que le nombre de défauts de cache engendrés en moyenne par une préemption dépend fortement des programmes et n'est pas proportionnel à la taille de la perturbation. En outre, certains programmes ne sont pas sensibles aux préemptions alors que d'autres au contraire subissent un nombre de défauts de cache significatif. Ceci est un résultat important pour l'ordonnancement puisque cela signifie qu'il vaut mieux préempter certaines tâches que d'autres.

Les deux modèles que nous proposons ont chacun leurs lacunes. Par conséquent, nous ne sommes pas en mesure d'évaluer simplement le nombre défauts de cache engendrés par une préemption.

d) Conclusion concernant les préemptions

Afin d'évaluer le nombre de défauts de cache engendrés par une préemption, nous évaluons le taux d'utilisation des caches par les programmes pour en déduire la perturbation δ subie. À partir de cette perturbation, et de la taille du cache, nous devrions être en mesure d'évaluer le nombre de défauts de cache causés par la préemption.

Cependant, nous nous sommes heurtés à plusieurs problèmes :

- Nous sommes capable d'évaluer précisément le chargement moyen pour la plupart des programmes mais ceci n'est qu'un chargement moyen qui peut beaucoup fluctuer pour certains programmes.
- Nous ne savons pas comment calculer le chargement depuis un cache non vide ce qui limite son utilisation.
- Les conséquences d'une perturbation dépend fortement du point de la préemption.
- Nous avons également des difficultés pour évaluer ne serait-ce que le nombre de défauts de cache moyen engendrés par une préemption en fonction de la taille de la perturbation.

Face à tous ces obstacles, nous renonçons pour le moment à évaluer le coût des préemptions pendant la simulation. Nous choisissons d'utiliser au contraire une pénalité fixe pour chaque programme. Cette pénalité pourra être éventuellement générée en utilisant le modèle n°2 au moment de la génération des systèmes, en fonction de la taille des caches et des programmes.

6.3.7 Conclusion

Au cours de ce chapitre, nous avons présenté un ensemble de modèles pour évaluer le CPI de tâches lorsqu'elles s'exécutent en isolation, avec un partage de cache ou avec des préemptions. Afin d'évaluer la qualité de ces modèles, nous avons comparé les estimations produites aux résultats obtenus par l'exécution de programmes sur un simulateur d'architecture matérielle. Cette étude nous a aussi permis d'approfondir notre compréhension des comportements des programmes vis-à-vis des caches.

Les comportements que nous avons observés sont très variés d'un programme à l'autre. Cependant, nous avons montré qu'il est possible de les caractériser à l'aide de leur API, CPI et SDP. Ces différents paramètres permettent bien de reproduire la sensibilité d'un programme aux perturbations induites par les autres quand il y a concurrence sur les caches. Nous retrouvons ainsi différents profils de comportement pour les programmes : un programme avec un taux d'accès elevé est plus robuste aux perturbations en gardant dans le cache les lignes les plus importantes et il peut potentiellement perturber davantage les autres programmes ; un programme qui fait des accès à une grande distance perturbe plus facilement les autres programmes, mais sera aussi plus facilement perturbé ; un programme qui fait beaucoup de *cold misses* occupe beaucoup de place dans les caches et risque donc de perturber les autres programmes.

Les évaluations réalisées montrent que ces effets sont difficiles à modéliser en particulier en ce qui concerne les préemptions. Cependant, notre objectif n'est pas de reproduire finement le comportement d'un programme particulier mais simplement de reproduire des comportements variés, maîtrisables et réalistes pour ensuite évaluer statistiquement le comportement des algorithmes en tenant compte des effets des caches. En cela, les résultats obtenus pour FOA et Babka sont satisfaisants pour évaluer le partage de cache. En ce qui concerne les préemptions, les variations sont telles qu'il nous paraît inapproprié d'essayer de les caractériser finement, en revanche il est possible de simuler leurs effets avec des pénalités déduites des SDP.

CHAPITRE 7

Prise en compte des surcoûts temporels dans SimSo

Ce chapitre présente la manière dont sont pris en compte, dans la simulation, les surcoûts temporels liés au système (ordonnancement et changements de contexte), ainsi que les effets des caches. Pour cela, nous avons identifié les endroits dans la simulation où nous devons ajouter des pénalités temporelles, et nous proposons un nouvel *Execution Time Model* (voir partie 4.2.4) pour simuler la durée d'exécution des travaux en tenant compte du partage des caches et des surcoûts liés aux préemptions et migrations.

Dans une seconde partie, nous verrons quatre expérimentations qui prennent en considération ces nouveaux aspects opérationnels pour évaluer les comportements de politiques d'ordonnancement en étudiant les taux d'utilisation et d'ordonnançabilité des systèmes.

7.1 Prise en compte des aspects opérationnels dans SimSo

7.1.1 Intégration des pénalités temporelles

SimSo a été développé pour permettre l'ajout de pénalités temporelles lors de certains évènements (voir chapitre 3). Ces pénalités sont prises en compte au niveau des objets *Processor* qui simulent l'exécution des travaux et le fonctionnement du système d'exploitation (voir section 4.2.3). Par exemple, si un processeur appelle une méthode de l'ordonnanceur, une attente sera ajoutée pour simuler le délai introduit par la décision d'ordonnancement et donc empêcher ce processeur de continuer à traiter des travaux.

La simulation réalisée par SimSo se base sur le moteur de simulation SimPy. Ce dernier permet de modéliser le système sous la forme de processus actifs ce qui facilite l'identification des moments où un surcoût doit être ajouté (voir section 4.2.3). De plus, SimPy offre une commande « hold » qui permet de mettre en attente un processus pendant un certain temps. Cette fonctionnalité est utilisée par les objets *Processor* pour leur permettre de passer du temps (au sens temps simulé) aux endroits où nous souhaitons simuler un surcoût temporel.

Actuellement, il est possible d'associer un tel délai lors d'un changement de contexte ou lors d'un appel à une méthode de l'ordonnanceur. Dans cette section, nous décrivons plus en détail la mise en œuvre de ces pénalités lors de ces différents évènements.

a) Changement de contexte

Un système préemptif repose fortement sur la notion de contexte. Le contexte d'une tâche en cours d'exécution correspond notamment à l'état de l'ensemble de ses registres. Lorsqu'une tâche est interrompue, son contexte doit être sauvegardé pour pouvoir être restauré au moment de la reprise de son exécution. Par conséquent, lorsque le système d'exploitation préempte une tâche A pour exécuter une tâche B, il doit interrompre le processeur sur lequel s'exécute la tâche A, sauvegarder son contexte puis charger le contexte de la tâche B.

La figure 7.1, déjà présentée dans la section 4.2.3, montre à quel endroit les pénalités de changement de contexte sont appliquées. L'exécution du travail ne commence réellement que lorsque ce temps est écoulé, et de manière analogue, le système d'exploitation ne peut utiliser le processeur qu'après un temps correspondant à la sauvegarde de contexte.

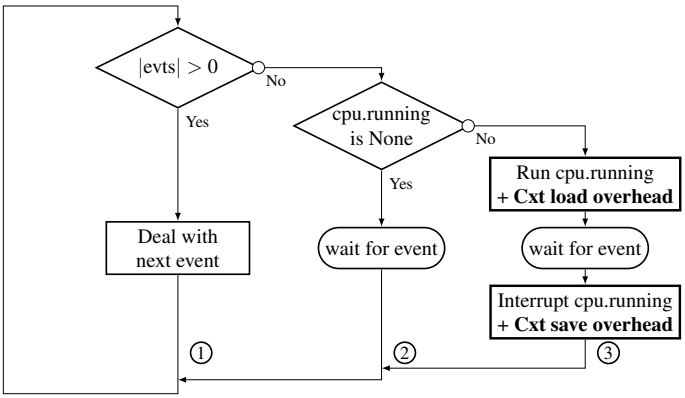


FIGURE 7.1 – Schéma simplifié de l'exécution d'un *Processor*. En gras les endroits où les pénalités sont insérées.

b) Appels à l'ordonnanceur

De la même manière, si l'évènement traité par l'un des objets de type *Processor* (branche 1 sur la figure 7.1) conduit à appeler l'une des méthodes de l'ordonnanceur (*schedule*, *on_activate* ou *on_terminated*), alors la commande « hold » est utilisée juste après l'appel.

La figure 7.2 donne un extrait du code qui gère l'appel à la méthode *schedule* et nous pouvons voir à la ligne 14 le moment où est appliqué la pénalité. Nous observons également dans cette fonction la journalisation de l'évènement au niveau du processeur et de l'ordonnanceur, ainsi que la gestion du verrou d'exclusion mutuelle (voir section 4.3.1). Ce verrou peut entraîner un coût supplémentaire en bloquant le processus tant qu'il n'est pas libéré.

Les pénalités associées à chaque méthode sont configurées de manière fixe dans la configuration du système. Cependant, l'ordonnanceur peut aussi modifier pendant l'exécution du système la valeur des différents surcoûts en modifiant ses attributs *overhead*, *overhead_activate*, *overhead_terminate*. Typiquement, ceci peut être fait dans la fonction concernée. Un ordonnanceur peut donc estimer le coût de l'exécution de la fonction et appliquer une pénalité temporelle variable (exemple : une fonction linéaire du nombre de tâches actives).

```
1      elif evt[0] == RESCHED:
2          # Log des évènements :
3          self.monitor.observe(ProcOverheadEvent("Scheduling"))
4          self.sched.monitor_begin_schedule(self)
5
6          # Gestion du Mutex si l'ordonnanceur ne peut pas être exécuté
7          # en même temps sur plusieurs processeurs :
8          yield waituntil, self, self.sched.get_lock
9
10         # Appel à l'ordonnanceur :
11         decisions = self.sched.schedule(self)
12
13         # Surcoût pour cet appel :
14         yield hold, self, self.sched.overhead
15
16         # Application des décisions d'ordonnancement
17          ...
18
19         # Libération du verrou :
20         self.sched.release_lock()
21
22         # Log fin d'ordonnancement :
23         self.sched.monitor_end_schedule(self)
```

FIGURE 7.2 – Extrait de la classe *Processor* montrant l'appel de la méthode *Schedule*

c) Autres pénalités temporelles

Il reste d'autres cas que nous aurions pu traiter tel que les coûts liés à la configuration d'un timer, à l'exécution de la méthode associée à un timer, au temps pour changer la fréquence d'un processeur ou encore aux délais supplémentaires pour les interruptions entre les processeurs. Notons que ces surcoûts ont pour point commun d'être des surcoûts liés au système et non aux tâches. En effet, il est aussi possible, comme nous allons le voir dans la section suivante, d'allonger la durée d'exécution des travaux suite à certains évènements.

7.1.2 Intégration de modèles de cache dans SimSo

Dans cette section, nous présentons la façon dont les modèles de cache retenus ont été intégrés dans SimSo. À savoir le modèle FOA pour simuler le partage de cache entre plusieurs programmes qui s'exécutent simultanément et des pénalités fixes pour les préemptions et migrations.

Dans un premier temps, nous détaillons les nouvelles entrées de la simulation nécessaires pour la simulation des caches. Puis, nous présentons le fonctionnement du module en charge de déterminer pendant la simulation la durée d'exécution des travaux (*Execution Time Model*).

a) Modélisation de l'architecture mémoire

Afin de pouvoir simuler l'influence des caches sur les durées d'exécution des tâches du système, nous avons besoin d'étendre les caractéristiques des tâches et de spécifier les caractéristiques des caches et de la hiérarchie mémoire.

Comportement mémoire des tâches

Le calcul du CPI avec le modèle FOA nécessite de connaître pour chaque tâche :

- le nombre moyen de cycles par instruction sans prendre en compte les pénalités liés aux accès mémoire (*base_cpi*) ;
- le nombre d'instructions exécutées par les travaux de la tâche (n) ;
- la proportion d'accès mémoire par instruction (API) ;
- le profil des accès mémoire pour un cache entièrement associatif (*Stack Distance Profile*) ;
- et le coût temporel d'une préemption ou migration.

Ces caractéristiques sont fixées par l'expérimentateur et peuvent être issues de vrais programmes ou être générées de manière artificielle. Notre précision dans la simulation ne nous permet pas de simuler exactement le comportement d'une tâche, ainsi ces données ne font que constituer un type de profil (réutilisation des données, *cold misses*, taux d'accès au cache, etc).

Caractérisation des caches

Rappelons que le modèle FOA est prévu pour des caches associatifs avec politique de remplacement LRU. Nous nous limitons aux caches de données (les effets des caches d'instruction sont pris en compte dans le *base_cpi*) et nous supposons que les caches sont entièrement associatifs.

Chaque cache est défini par :

- le nom du cache, utilisé pour y faire référence dans la hiérarchie mémoire ;
- la taille du cache en nombre de lignes ;
- le temps d'accès au cache pour un processeur.

Les caches sont organisés sous la forme d'une hiérarchie de caches inclusifs. Une liste de caches (exemple : [L1, L2, L3]) est ainsi associée à chaque processeur et les caches peuvent être éventuellement partagés tout en respectant la contrainte d'inclusivité. Les pénalités de défaut de cache seront déduites des temps d'accès en utilisant la hiérarchie mémoire.

La figure 7.3 montre l'exemple d'une hiérarchie mémoire composée de deux niveaux de cache dont le second est partagé entre deux cœurs. Les tableaux 7.1a et 7.1b montrent la façon dont cet exemple est modélisé pour SimSo, le temps d'accès à la mémoire est simplement ajouté dans l'onglet *General*. Chaque cœur contient la liste des caches qui lui sont associés (respectivement [L11, L2] et [L12, L2]) et les caractéristiques des trois caches sont renseignés séparément. Les valeurs de pénalité sont égales à la différence des temps d'accès entre les niveaux de la hiérarchie mémoire.

Name	Size	Access Time
L11	128	1
L12	128	1
L2	512	10

(a) Onglet *Caches*.

Name	Caches
C1	L11, L2
C2	L12, L2

(b) Onglet *Processors*.

Tableau 7.1 – Modélisation des caches dans SimSo.

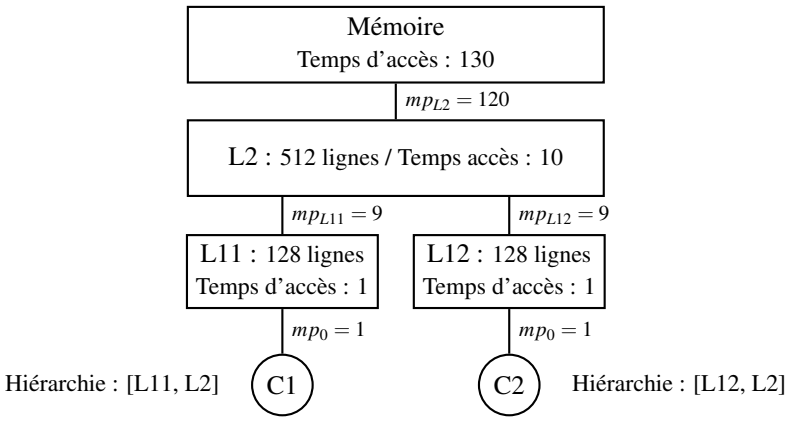


FIGURE 7.3 – Exemple de hiérarchie mémoire

b) Intégration dans SimSo des modèles de cache

La prise en compte des caches se fait en agissant sur la durée d'exécution des travaux simulés. La section 4.2.4 explique le fonctionnement des *Execution Time Models* (ETM) dans SimSo. Pour rappel, l'objet ETM est appelé par les travaux (objets *Job)* afin de connaître une borne inférieure de la durée d'exécution restante et pour informer l'ETM des évènements de la simulation. La figure 7.4 rappelle l'interface entre un ETM et un *Job*.

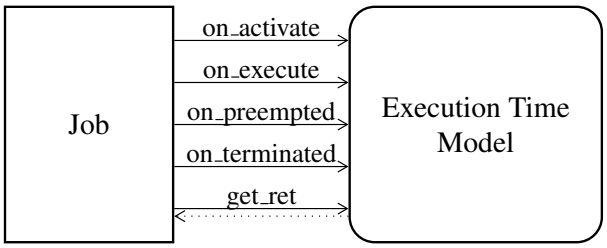


FIGURE 7.4 – Interface d'un *Execution Time Model*.

Prise en compte des pénalités de préemption et migration

Dans l'ETM, nous avons ajouté un premier dictionnaire *running* qui contient pour chaque processeur le travail en cours d'exécution, et un second dictionnaire *was_running_on* qui contient pour chaque travail le processeur sur lequel il s'est exécuté pour la dernière fois. Un troisième dictionnaire, *penalty*, permet de comptabiliser pour chaque travail la pénalité cumulée engendrée par les préemptions et migrations.

Lors de l'appel à la méthode *on_execute*, le dictionnaire *was_running_on* est consulté pour savoir si le travail s'est déjà exécuté (on ignore la première exécution), et si c'est le cas alors on regarde si le travail est resté sur le même processeur ou s'il a migré. Dans le cas d'une simple préemption, on regarde si un autre travail s'est exécuté sur ce processeur entre-temps. S'il s'agit d'une migration ou s'il s'agit d'une interruption avec exécution d'un autre travail sur le même processeur, alors on ajoute à la pénalité du travail la pénalité de migration. La figure 7.5 montre l'extrait du code de SimSo qui correspond à l'algorithme qui vient d'être présenté.

La pénalité calculée sera additionnée au temps d'exécution du travail dans la fonction *get_ret* que nous détaillons à la fin de cette section. Remarquons que ces pénalités sont considérées comme fixes, mais pourraient être variables en intégrant de nouveaux modèles.

```
def on_execute(self, job):
    # Calcul de la pénalité :
    if job in self.was_running_on:
        # Reprise sur le même processeur.
        if self.was_running_on[job] is job.cpu:
            # Est-ce qu'un autre travail s'est exécuté entre-temps ?
            if self.running[job.cpu] is not job:
                self.penalty[job] += job.task.preemption_cost
        else:  # migration.
            self.penalty[job] += job.task.preemption_cost

    # Mise à jour des dictionnaires :
    self.running[job.cpu] = job
    self.was_running_on[job] = job.cpu

    ...
```

FIGURE 7.5 – Extrait de la fonction *on_execute* pour le calcul de la pénalité de préemption et migration.

Prise en compte du partage de cache

Lorsque les caches sont pris en compte, chaque travail dans SimSo dispose d'un certain nombre d'instructions à exécuter. La durée nécessaire pour l'exécution de ces instructions dépend du CPI du travail. Ce CPI est calculé à l'aide du modèle FOA présenté dans le chapitre 6, à partir des autres caractéristiques du travail (API, SDP, base_cpi) ainsi que des caractéristiques des caches (taille, pénalité, hiérarchie).

À chaque changement d'état dans le système (exécution ou interruption d'un travail), le CPI de chaque travail est calculé pour l'intervalle défini entre le dernier évènement et la date courante. En divisant la durée de l'intervalle par le CPI des travaux, on obtient pour chaque travail le nombre d'instructions exécutées sur cet intervalle. Le nombre d'instructions exécutées depuis le début pour chaque travail est ainsi mis à jour (figure 7.6). Pour le calcul de FOA, la liste des travaux en cours d'exécution est nécessaire pour évaluer le partage des caches.

```
def _update_instructions(self):
    for job in self._running_jobs:
        # Calcul du nombre d'instruction pour now() - last_update
        instr = compute_instructions(job.task, self._running_jobs,
                                     self.sim.now() - self._last_update)
        # Mise à jour du nombre d'instructions pour ce travail
        self._instr_jobs[job] = self._instr_jobs.get(job, 0) + instr

    # Mise à jour de last_update
    self._last_update = self.sim.now()
```

FIGURE 7.6 – Fonction *_update_instructions* dans l'ETM pour mettre à jour le nombre d'instructions exécutées.

La méthode *on_execute* de l'ETM est complétée par une mise à jour du nombre d'instructions et de la liste des travaux en cours d'exécution (figure 7.7). De même, lorsqu'un travail est préempté (*on_preempted*) ou se termine (*on_terminated*), le nombre d'instructions est mis à jour puis le travail est retiré de la liste des travaux en cours d'exécution.

```
def on_execute(self, job):
    ...
    # Mise à jour du nombre d'instructions pour l'intervalle [last_update, now].
    self._update_instructions()
    # Ajout du travail dans la liste des travaux en cours d'exécution.
    self._running_jobs.add(job)
```

FIGURE 7.7 – Suite de la fonction *on_execute* pour la mise à jour du nombre d'instructions exécutées.

Calcul du temps restant d'exécution

La fonction *get_ret* retourne une borne inférieure du temps restant d'exécution. Cette durée est calculée à partir du nombre restant d'instructions à exécuter et du CPI calculé sans partage de cache entre tâches. La pénalité des préemptions et migrations, calculée dans la fonction *on_execute*, est ensuite ajoutée à cette durée. La figure 7.8 reproduit le code de cette méthode.

```
def get_ret(self, job):
    self._update_instructions()
    penalty = self.penalty[job]
    return (job.task.n_instr - self._instr_jobs[job]) \
        * job.task.get_cpi_alone() + penalty
```

FIGURE 7.8 – Fonction *get_ret* qui retourne une borne inférieure de la durée restante d'exécution.

7.2 Expérimentations

Dans cette seconde partie, nous montrons quelques exemples d'utilisation de SimSo avec prise en compte des pénalités temporelles et/ou des caches. L'objectif est de montrer que nous sommes effectivement en mesure de prendre en considération ces différents phénomènes.

7.2.1 Comparaison de G-EDF et P-EDF avec pénalités temporelles

Les politiques d'ordonnancement G-EDF et P-EDF se différencient en particulier par leur gestion des tâches prêtes. L'algorithme G-EDF utilise une unique liste de travaux prêts alors que P-EDF maintient une liste par processeur. Cette différence a pour conséquence pratique de créer une section critique pour l'ordonnanceur G-EDF alors que l'algorithme P-EDF peut s'exécuter simultanément sur plusieurs processeurs de manière indépendante.

Nous évaluons ici les algorithmes G-EDF et P-EDF avec des pénalités associées aux méthodes de l'ordonnanceur. Nous regardons plus particulièrement l'impact sur la charge du système d'un « verrou » reproduisant la section critique et interdisant l'exécution simultanée de deux fonctions *schedule* sur deux cœurs différents.

Protocole

Pour chaque nombre de processeurs allant de 1 à 8, nous générons 100 systèmes constitués de 20 tâches et avec un taux total d'utilisation de 0.8 fois le nombre de processeurs. Les

taux d'utilisation des tâches sont générés à l'aide de l'algorithme RandFixedSum et les périodes sont générées selon une distribution log-uniforme sur l'intervalle 2 à 100 ms. La durée de simulation est de 1000 ms.

Nous utilisons l'ETM qui exécute les travaux pour une durée égale à leur WCET. Le coût de l'appel à la méthode *schedule* de l'ordonnanceur est fixé à 0.01 ms et les autres pénalités temporelles sont fixées à 0.

L'ensemble de ces systèmes a été ordonnancé à l'aide des algorithmes d'ordonnancement G-EDF et P-EDF. Pour chaque expérimentation, nous extrayons le taux total d'utilisation correspondant aux appels à la méthode *schedule* (somme du temps passé dans la méthode *schedule* divisé par la durée de la simulation).

Résultats

Le nombre d'appels à la méthode *schedule* dépend principalement des tâches (leur nombre et leurs périodes). Ici, seul le nombre de processeurs varie, ainsi le nombre d'appels à la méthode *schedule* varie très peu (résultat obtenu lors des évaluations du chapitre 5 mais non montré).

La figure 7.9 montre le taux d'utilisation total correspondant à l'exécution de la méthode *schedule* (temps passé à exécuter la méthode *schedule* divisé par la durée de la simulation). Nous voyons clairement que le taux d'utilisation pour la méthode *schedule* reste constant avec P-EDF alors qu'il augmente en augmentant le nombre de processeurs avec G-EDF. Puisque dans les deux cas le nombre d'appels à l'ordonnanceur est constant, la différence est donc liée au verrou.

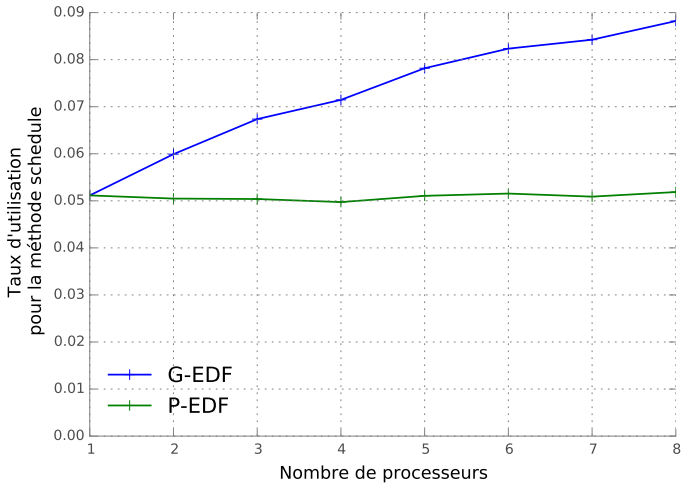


FIGURE 7.9 – Utilisation totale pour la méthode *schedule* en fonction du nombre de processeurs.

Conclusion

En conclusion, les surcoûts temporels liés aux appels à l'ordonnanceur augmentent avec le nombre de processeurs pour l'algorithme G-EDF alors que P-EDF conserve un coût fixe. Nous pouvons cependant nuancer ce résultat en remarquant que le coût relatif des appels à l'ordonnanceur décroît dans les deux cas avec l'augmentation du nombre de processeurs (figure 7.10).

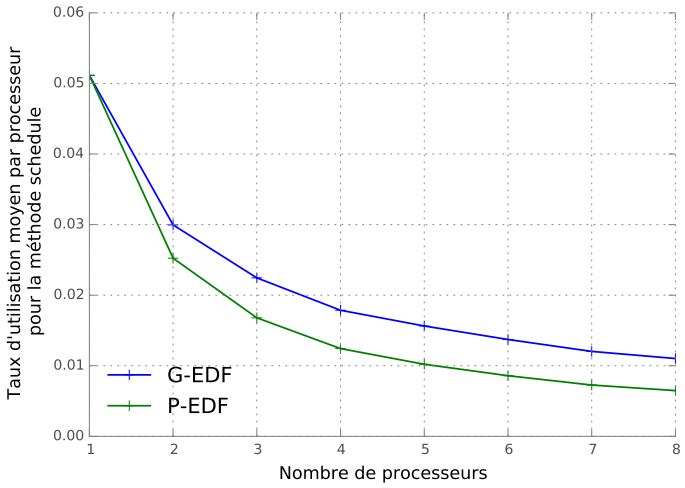


FIGURE 7.10 – Utilisation relative (utilisation totale divisée par le nombre de processeurs) pour la méthode *schedule* en fonction du nombre de processeurs.

Enfin, notons que le choix des périodes a probablement une influence dans les résultats. En effet, dans le cas de systèmes synchrones avec des périodes multiples voire identiques, la méthode *schedule* est appelée simultanément pour prendre plusieurs décisions. L'algorithme P-EDF est donc plus adapté à ce type de système grâce à l'absence de section critique (verrou). De plus, ici nous avons étudié une implémentation de G-EDF qui prend une seule décision d'ordonnancement à la fois mais il serait possible de prendre plusieurs décisions d'un coup pour réduire les coûts.

7.2.2 Variation des *offsets* avec prise en compte des caches

L'objectif de cette étude est de montrer l'influence des dates de première activation (*offset* O_i) sur les performances d'un système en prenant en compte les caches, pour un ordonnancement P-EDF. Pour cela, nous définissons un système multiprocesseur avec un cache partagé puis nous tirons aléatoirement des valeurs de O_i.

Protocole

Nous avons défini un système constitué de deux cœurs, un cache L1 privé par cœur et un cache L2 partagé (voir figure 7.11). Chaque cache L1 fait 1 Kio et le cache L2 a une taille de 16 Kio puis 32 Kio lors de la seconde évaluation. Les temps d'accès aux caches sont indiqués sur la figure.

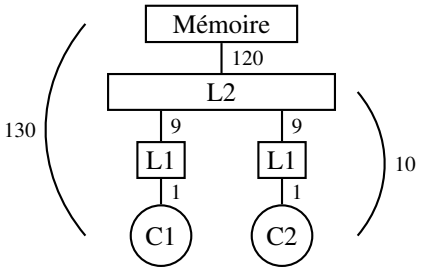


FIGURE 7.11 – Hiérarchie mémoire utilisée pour l'expérimentation sur l'influence des offsets

Concernant les tâches, nous avons utilisé les métriques extraites des benchmarks présentés au cours du chapitre 6 (SDP, API, Instructions) et nous avons défini les périodes et WCET manuellement. Le tableau 7.2 résume les caractéristiques des tâches utilisées.

Pour simuler la durée d'exécution des travaux, nous utilisons l'ETM qui simule les caches. Ainsi le paramètre WCET n'est utilisé que pour le partitionnement des tâches. Le WCET a été fixé de sorte à ce que dans toutes les expérimentations, les durées simulées pour chaque travail restent inférieures à leur WCET.

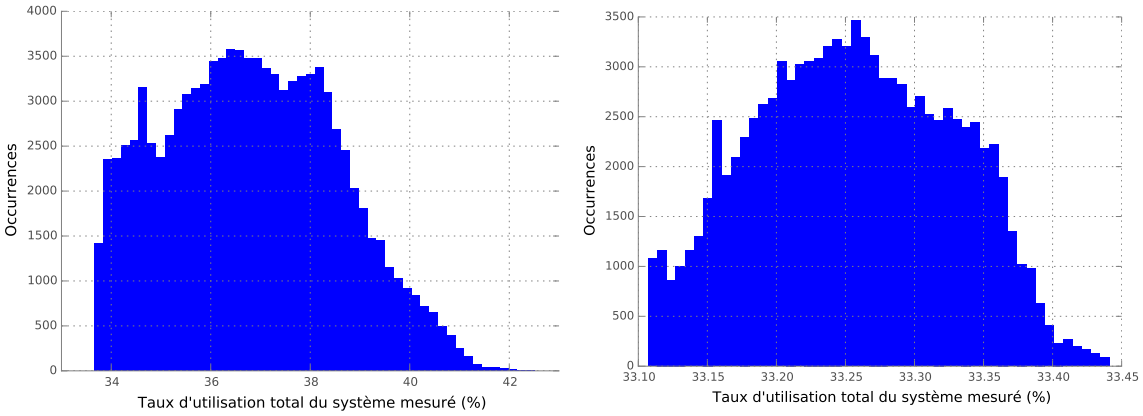
Nom	Période (ms)	WCET (ms)	Instructions	API
Dijkstra-large	40	9	188 603 755	0.30
MATMULT-O2	30	10	47 922 711	0.37
COMPRESS-O2	20	5	46 771 323	0.28
Patricia	30	8	204 293 931	0.34
CNT-O2	10	3	143 775 036	0.08

TABLEAU 7.2 – Caractéristiques des tâches ($\sum u_i = 1.35$).

Une fois le système défini, nous avons automatisé l'exécution de 100 000 simulations en générant les O_i aléatoirement sur $[0, T_i]$ pour chaque taille de cache L2 (16 Kio puis 32 Kio). La durée d'une simulation est de six hyper-périodes. Pour chaque simulation, nous collectons le taux d'utilisation mesuré du système (en pourcentage) sur les cinq dernières hyper-périodes.

Résultats

La figure 7.12 montre la distribution des taux d'utilisation que nous avons obtenu sur les 100 000 simulations en faisant varier les dates de première activation des tâches par un tirage aléatoire. Les résultats sont plus concentrés lorsque la taille du cache partagé est doublé car les effets du partage du cache sont largement réduits. Pour un cache L2 de 16 Kio, l'utilisation [1] varie de 33.66% à 42.51% soit une étendue relative ($\frac{x_{max}-x_{min}}{\bar{x}}$) de l'ordre de 24%. En doublant la taille du cache L2 (32 Kio), les valeurs se concentrent entre 33.11% à 33.44%, soit une étendue relative de l'ordre de 1%.



(a) Taille L2 : 16 Kio

(b) Taille L2 : 32 Kio

FIGURE 7.12 – Distribution des taux d'utilisation mesurés en faisant varier les dates de première activation et la taille du cache partagé L2.

1. Le terme utilisation correspond à $\sum \frac{C_i/T_i}{m}$ avec C_i la durée d'exécution simulée ($C_i < WCET_i$). Cette valeur est donc inférieure à $\sum u_i/m = 1.35/2 = 67.5\%$.

Nom	O_i
Dijkstra-large	19.2580
MATMULT-O2	24.9177
COMPRESS-O2	9.7568
Patricia	3.8064
CNT-O2	7.2040

Tableau 7.3 – Dates de première activation ayant conduit au taux d'utilisation le plus faible (33.66%)

Nom	O_i
Dijkstra-large	12.0989
MATMULT-O2	16.5196
COMPRESS-O2	9.0495
Patricia	12.9212
CNT-O2	6.5124

Tableau 7.4 – Dates de première activation ayant conduit au taux d'utilisation le plus fort (42.51%)

Les tableaux 7.3 et 7.4 indiquent les valeurs des dates d'activation ayant conduit aux taux d'utilisation le plus faible et le plus fort respectivement pour un cache L2 de 16 Kio.

Conclusion

Initialement, nous sommes partis de l'observation que, dans le cadre de l'utilisation de P-EDF, les mêmes tâches ont tendance à s'exécuter en même temps du fait de périodes multiples voire identiques. Dans le cas d'un système peu chargé comme celui que nous venons d'étudier, les décalages permettent alors d'exécuter certaines tâches pendant une période d'inactivité de l'autre processeur ce qui supprime la compétition sur le cache partagé. Dans le cas de systèmes plus chargés, nous avons aussi pu produire des résultats similaires car les tâches les plus consommatrices en cache peuvent s'exécuter en même temps que des tâches qui n'utilisent pas beaucoup de lignes dans le cache.

Cette évaluation a permis de montrer qu'un paramètre aussi simple que la date de la première activation pouvait conduire à une variation importante du taux d'utilisation effectif du système. Il est donc possible de choisir des dates d'activation qui minimisent l'utilisation d'un système avec l'algorithme d'ordonnancement P-EDF.

7.2.3 Impact du placement des tâches

Nous étudions maintenant l'impact sur le taux d'utilisation total du système du placement des tâches sur les processeurs avec l'algorithme P-EDF. Dans cette étude, nous étendons l'expérience précédente à l'ensemble des partitionnements possibles pour le système considéré.

Protocole

Nous réutilisons le système présenté dans l'étude précédente composé de 5 tâches et de 2 cœurs (voir tableau 7.2). Ce système comprend deux niveaux de cache : un premier niveau L1 privé de 1 Kio et un second niveau partagé de 16 Kio (voir figure 7.11).

Les partitionnements possibles sont générés à l'aide d'un script utilisant SimSo pour lire le système initial et pour générer les fichiers XML des systèmes. Chacun de ces fichiers XML est ensuite passé au script utilisé dans la section 7.2.2 pour générer les distributions à partir de 10 000 simulations en faisant varier les dates de première activation.

Résultats

Sur les 16 partitionnements possibles, seuls 10 systèmes sont ordonnançables selon le critère $\sum u_i \leq 1$. Le tableau 7.5 regroupe les médianes des différents systèmes. La figure 7.13 montre la distribution de trois de ces partitions :

- en bleu, {Dijkstra-large, COMPRESS-O2, CNT-O2} {MATMULT-O2, Patricia} ;
- en vert, {Dijkstra-large, MATMULT-O2, CNT-O2} {COMPRESS-O2, Patricia} ;
- en rouge, {Dijkstra-large, CNT-O2} {MATMULT-O2, COMPRESS-O2, Patricia} (utilisé dans l'étude précédente).

Partition	Utilisation (%)
{Dijkstra-large, MATMULT-O2, COMPRESS-O2} {Patricia, CNT-O2}	35.31
{Dijkstra-large, MATMULT-O2, Patricia} {COMPRESS-O2, CNT-O2}	35.38
{Dijkstra-large, MATMULT-O2, CNT-O2} {COMPRESS-O2, Patricia}	34.86
{Dijkstra-large, MATMULT-O2} {COMPRESS-O2, Patricia, CNT-O2}	36.70
{Dijkstra-large, COMPRESS-O2, Patricia} {MATMULT-O2, CNT-O2}	36.36
{Dijkstra-large, COMPRESS-O2, CNT-O2} {MATMULT-O2, Patricia}	38.30
{Dijkstra-large, COMPRESS-O2} {MATMULT-O2, Patricia, CNT-O2}	36.18
{Dijkstra-large, Patricia, CNT-O2} {MATMULT-O2, COMPRESS-O2}	37.15
{Dijkstra-large, Patricia} {MATMULT-O2, COMPRESS-O2, CNT-O2}	35.20
{Dijkstra-large, CNT-O2} {MATMULT-O2, COMPRESS-O2, Patricia}	36.75

Tableau 7.5 – Médianes des taux d'utilisation des systèmes.

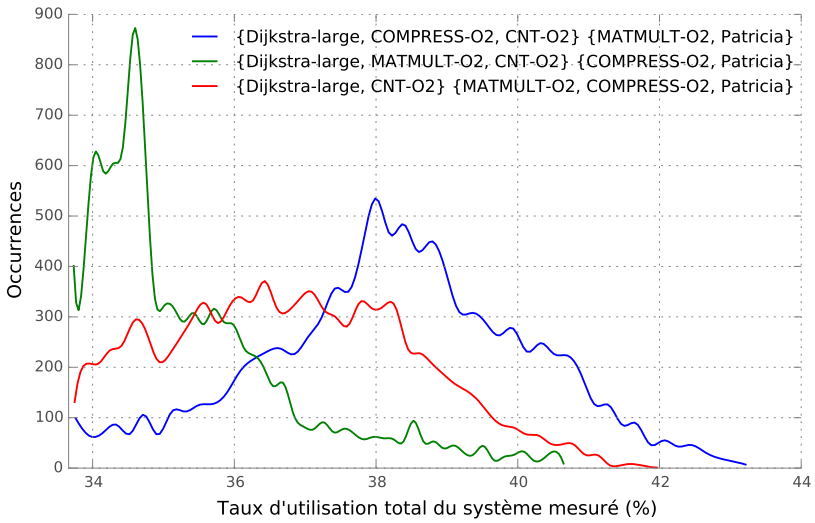


Figure 7.13 – Distribution des taux d'utilisation avec variation des dates de première activation pour trois placements de tâches différents.

Conclusion

À travers cet exemple, nous avons vu que le placement des tâches sur les processeurs peut avoir une influence sur les performances du système à travers une bonne utilisation des caches. Ainsi, lorsqu'il y a plusieurs partitionnements possibles pour un système, il serait pertinent de prendre en considération les caractéristiques mémoire des tâches.

7.2.4 Pénalités de préemption et migration

Dans cette expérimentation, nous mesurons la capacité des algorithmes G-RM, G-EDF, G-FL, P-EDF (*decreasing first-fit*) et P-EDF (*decreasing worst-fit*) à supporter des surcoûts supplémentaires qui n'avaient pas été pris en compte dans le WCET des tâches. Ceci concerne donc plus particulièrement le cas de systèmes temps réel souples.

Les algorithmes qui se basent fortement sur le WCET pour l'ordonnancement n'ont pas pu être évalués puisque ces algorithmes ne tolèrent généralement pas qu'un travail puisse nécessiter une durée d'exécution supérieure au WCET (budget alloué insuffisant au mieux, erreurs d'ordonnancement dans certains cas).

Protocole

Un ensemble de 301 systèmes composés de 12 tâches et 4 processeurs ont été générés en faisant varier le taux total d'utilisation entre 1 et 4 par pas de 0.01 (algorithme RandFixedSum). Ces systèmes sont ensuite simulés sur une durée de 1000 ms avec l'ETM « FixedPenalty » qui se base sur le WCET et permet d'appliquer des pénalités pour les préemptions et migrations. La pénalité associée à une préemption ou une migration a été fixée progressivement de 0 à 0.16 ms par pas de 0.01 ms. Ce sont donc au total 5117 configurations qui ont été simulées avec chacun des cinq algorithmes d'ordonnancement choisis.

Soient Q l'ensemble des taux totaux d'utilisation et $S(U, D)$ le résultat de la simulation du système pour un taux d'utilisation total U et pour un coût de préemption et migration D [BBA11]. $S(U, D)$ prend la valeur 1 si l'ordonnancement des tâches s'est bien déroulée et 0 sinon. À partir de ces résultats, nous calculons le taux d'ordonnançabilité pondéré par les taux d'utilisations :

$$W(D) = \frac{\sum_{U \in Q}(U \cdot S(U, D))}{\sum_{U \in Q} U} \tag{7.1}$$

Résultats

La figure 7.14 donne la valeur du taux d'ordonnaçabilité pondéré, $W(D)$, en fonction de la pénalité associée à une préemption ou migration. Nous constatons que P-EDF est capable d'ordonnancer un nombre de systèmes plus important que les autres algorithmes lorsqu'il n'y a pas de pénalité appliquée. En revanche, le taux d'ordonnançabilité pour P-EDF décroit rapidement, en particulier lorsque l'algorithme de placement *decreasing first-fit* est utilisé. Au contraire, la pente pour les algorithmes globaux est beaucoup plus faible.

L'algorithme P-EDF avec l'algorithme de placement *decreasing first-fit* cherche à minimiser le nombre de processeurs utilisés en chargeant au maximum les processeurs. L'inconvénient est alors que certains processeurs sont très chargés et ne peuvent donc plus ordonnancer

correctement les tâches si les coûts de préemption augmentent. Pour illustrer cela, il suffit de prendre l'exemple d'un système avec $U = 1$. L'algorithme *decreasing first-fit* va affecter toutes les tâches au premier processeur et laisser les trois autres processeurs libres. À la moindre augmentation d'une durée d'exécution, il y aura des dépassements d'échéance.

Dans le cas de l'utilisation de *decreasing worst-fit*, les tâches sont réparties plus équitablement sur les processeurs ce qui a deux avantages : le premier est la réduction du problème observé avec l'algorithme first-fit et le second est lié à la réduction du nombre de préemptions (voir section 5.3.2).

Les algorithmes globaux sont quant à eux plus flexibles grâce à la migration des tâches. Cela permet donc à une tâche de s'exécuter sur un autre processeur si d'autres tâches ont dépassé leur WCET. Cette différence permet à ces algorithmes de donner de meilleurs résultats que P-EDF. L'algorithme G-FL donne de meilleurs résultats que G-EDF qui améliore G-RM ce qui est en accord avec nos précédentes études.

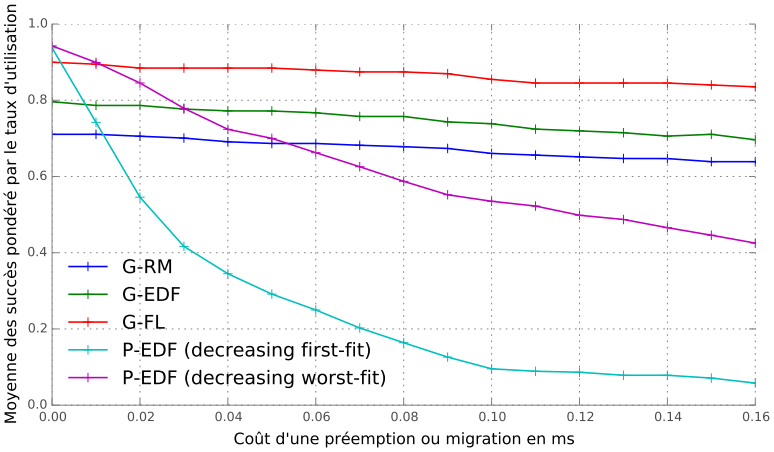


FIGURE 7.14 – Taux d'ordonnançabilité pondéré par les taux d'utilisations en fonction du coût d'une préemption ou migration.

Conclusion

L'algorithme P-EDF est l'algorithme permettant d'ordonnancer le plus de systèmes en l'absence de pénalité pour les préemptions. Mais pour des systèmes temps réel souples où les pénalités de préemption et migration n'ont pas été intégrés dans les durées d'exécution des travaux, les algorithmes globaux sont plus robustes grâce à la flexibilité apportée par les migrations qui permet de s'adapter à des comportements anormaux.

Cette étude pourrait être complétée en attribuant des pénalités de migration plus importantes que celles de préemption, en augmentant le nombre de processeurs ou encore en simulant sur des durées aléatoires. Tous ces changements seraient en faveur de P-EDF.

7.3 Conclusion

Dans la première partie de ce chapitre, nous avons montré comment nous avons intégré des surcoûts temporels dans SimSo. L'intégration de surcoûts système tels que les coûts de changement de contexte ou le temps d'exécution des méthodes de l'ordonnanceur a été simplifiée par la simulation à base de *processus*. Nous pourrions de la même manière ajouter

d'autres surcoûts, tels que ceux listés par Bastoni *et al.* [BBA11]. La prise en compte des caches s'est faite en modifiant la durée d'exécution simulée des travaux (*Execution Time Model*). La modularité de SimSo offre la possibilité d'utiliser d'autres modèles de cache si nécessaire. Ceci peut être fait dans le but d'améliorer la précision des résultats ou éventuellement prendre en compte les caches avec cette fois-ci un objectif d'étude du pire cas.

Nous avons ensuite illustré la prise en compte de ces surcoûts dans la simulation à travers quatre études. Nous avons fait le choix d'isoler les surcoûts que nous prenons en compte pour mieux montrer leurs effets. Ainsi dans un premier temps nous avons étudié la prise en compte du coût à l'appel de la méthode *schedule* avec et sans verrou. Puis nous avons étudié l'effet du partage de cache en modifant les dates de première activation et le placement des tâches avec l'algorithme d'ordonnancement P-EDF. Enfin, nous avons étudié les effets des préemptions sur l'ordonnancement dans un système temps réel souple.

La prise en compte dans la simulation de ces surcoûts permet d'envisager de nouvelles études sur les politiques d'ordonnancement, en tenant compte de certains aspects qui sont généralement mis de côté car trop complexes à évaluer. La prochaine étape consiste donc naturellement à réaliser des campagnes d'évaluation plus larges en tenant compte de ces surcoûts.

Conclusion

Bilan

L'état de l'art sur les politiques d'ordonnancement temps réel multiprocesseur nous a amené à plusieurs constats : (i) il est difficile d'avoir une vue globale des politiques existantes car la production scientifique dans ce domaine est très importante et les approches sont variées et théoriquement incomparables ; (ii) les évaluations et comparaisons des politiques d'ordonnancement disponibles dans la littérature ne permettent pas d'avoir une vue synthétique des résultats car ils sont souvent partiels, obtenus dans des conditions expérimentales mal définies et difficilement reproductibles ; (iii) certains aspects opérationnels tels que les surcoûts liés à l'ordonnancement ou l'influence de l'architecture matérielle sur les durées d'exécution sont souvent écartés par des hypothèses simplificatrices et sont rarement considérés dans les évaluations.

Le travail présenté dans ce manuscrit tente d'apporter des solutions – au moins partielles — à ces différents constats et de faciliter le travail actuel et futur d'évaluation des politiques d'ordonnancement temps réel avec une attention particulière pour celles dédiées aux architectures multicœurs.

Pour ce faire, nous proposons une approche empirique par simulation afin d'obtenir des tendances générales sur le comportement des algorithmes d'ordonnancement. Cette simulation prend en compte certains aspects opérationnels, en particulier la mémoire cache et les surcoûts du système, sans pour autant chercher à reproduire finement l'exécution réelle d'un système en particulier. En cela nous qualifions notre démarche de simulation à « grain intermédiaire ». Ceci se traduit par la mise à disposition d'outils et de méthodes pour faciliter la mise en place de larges campagnes d'évaluation qui soient facilement exploitables et reproductibles. La solution offerte ne nécessite pas de systèmes physiques spécifiques, ni des compétences en développement informatique trop pointues. En effet, la prise en main des outils doit être simple pour s'adresser au chercheur qui souhaite rapidement valider une nouvelle idée ou conduire une évaluation plus approfondie, mais aussi à l'étudiant qui découvre l'ordonnancement temps réel.

Pour mettre en place ces outils nous avons commencé par faire un état de l'art sur les politiques d'ordonnancement temps réel proposées jusqu'à aujourd'hui. Cela nous a permis de répertorier plus d'une cinquantaine d'algorithmes pour l'ordonnancement en ligne de tâches indépendantes sur des architectures multiprocesseurs et basées sur des stratégies diverses : partitionnement, approche globale, semi-partitionnement, clustering, etc. La connaissance de ces politiques nous a permis de concevoir l'architecture de base du simulateur et une majorité d'entre eux a été implémentée, mise à disposition avec le simulateur et évaluée lors de campagnes expérimentales.

Nous avons également étudié les éléments opérationnels pouvant avoir un impact sur le comportement des algorithmes d'ordonnancement. Plus particulièrement, nous nous sommes focalisés sur les coûts liés à l'exécution du code de l'ordonnanceur et sur les effets des caches sur les durées d'exécution des travaux. Afin d'introduire ces éléments dans la simulation, nous avons étudié différents modèles issus de la communauté scientifique autour de l'évaluation de performance des architectures matérielles. Ces modèles permettent de représenter l'influence du cache de manière statistique ce qui convient parfaitement à nos objectifs d'évaluation et au niveau d'abstraction de simulation souhaité. Afin de valider ces modèles, nous avons procédé à leur évaluation sur des ensembles de benchmarks et qualifié leur comportement dans des situations d'ordonnancement.

Ces différents éléments d'étude nous ont permis de mettre en œuvre un simulateur d'ordonnancement baptisé *SimSo*. Cet outil, développé en Python, est mis à la disposition de la communauté sous licence libre. Une bibliothèque de plus de vingt-cinq algorithmes d'ordonnancement l'accompagne ainsi qu'un ensemble d'outils et de scripts pour faciliter son utilisation dans le cadre de campagnes d'évaluation. La simulation repose sur le moteur de simulation à évènements discrets SimPy. Le degré d'abstraction offert par SimPy nous a permis d'exprimer de façon simple et naturelle le fonctionnement du système. L'intégration des modèles statistiques du comportement du cache a été réalisée à travers un module du simulateur dédié à la gestion des durées d'exécution des travaux, module pouvant être facilement modifié afin d'intégrer de nouveaux modèles ou d'étendre ceux déjà présents. Nous avons aussi pris en considération dans la simulation les surcoûts liés aux appels à l'ordonnanceur par des pénalités temporelles paramétrables.

Enfin, nous avons utilisé SimSo pour automatiser la simulation et l'exploitation de plus de cinq cent mille simulations. Nous avons pu ainsi évaluer comparativement les principaux algorithmes d'ordonnancement actuellement disponibles. Ces expérimentations ont été réalisées dans un premier temps sans prendre en compte les aspects opérationnels ce qui nous a permis de confirmer certains comportements présentés dans la littérature, mais aussi d'étendre certaines analyses en étudiant une plus grande variété de systèmes et d'établir de nouveaux résultats. Nous avons ensuite intégré les aspects opérationnels pour montrer l'impact sur le taux d'utilisation ou le taux de succès du partage des caches, des coûts de préemption et des coûts d'ordonnancement.

Perspectives

Le travail réalisé dans le cadre de cette thèse nous a permis de mettre à disposition de la communauté un environnement complet de simulation et d'expérimentation pour l'évaluation des politiques d'ordonnancement. Nous avons aussi montré qu'à l'aide de modèles plus ou moins sophistiqués, les aspects opérationnels peuvent être pris en compte et permettre ainsi de réaliser des campagnes expérimentales mettant en perspectives l'impact des aspects opérationnels sur les algorithmes. Cependant, de nombreux points restent encore à approfondir aussi bien en ce qui concerne l'outillage, que la conduite de nouvelles expérimentations ou bien plus fondamentalement sur l'utilisation des modèles exploités.

Diffusion et développement de SimSo

Tout d'abord, en ce qui concerne l'outil, nous souhaitons continuer à promouvoir SimSo et à le maintenir afin qu'il soit utilisé et enrichi par la communauté. Bien qu'il commence

à susciter de l'intérêt [BFO14, TSRB14], il semble évident qu'il faille continuer à faciliter son accessibilité. Cela doit passer par l'amélioration de la documentation et par la mise en ligne des réponses aux questions les plus fréquentes. Des exemples concrets pour l'automatisation des évaluations doivent également être fournis. Enfin, mettre en place un moyen pour permettre la contribution de personnes extérieures au projet semble une bonne idée afin de maintenir le développement de l'outil.

Dans un avenir proche, nous avons prévu d'enrichir la bibliothèque des ordonnanceurs par des algorithmes publiés récemment [MLR+14, NSG+14], d'autres que nous n'avons simplement pas pris le temps de mettre en œuvre [KYI09, KC11] ou encore des variantes que nous n'avons pas encore considérées [BNVT14, DK12] mais que nous souhaiterions évaluer. En complément de ce travail, il nous semble indispensable de bien documenter les choix d'implémentation qui ont été faits car à notre connaissance il n'existe pas de documents formalisant ce type d'informations. Cela se révèle être un véritable manque à partir du moment où l'on souhaite réellement mettre en œuvre une politique d'ordonnancement. En effet, certaines alternatives d'implémentation peuvent avoir un impact important sur les performances d'un algorithme, il faut donc être capable de les identifier et de les répertorier précisément.

Toujours concernant SimSo, nous souhaitons étendre les modèles de tâches disponibles pour notamment prendre en compte les dépendances entre les tâches, le partage de ressources ou encore les tâches « multithreadées ». Nous souhaiterions aussi étendre le modèle de l'architecture matérielle pour permettre de modéliser des systèmes temps réel plus complexes avec des architectures distribuées. Ce travail devrait être facilité grâce à l'architecture modulaire de SimSo et permettrait aussi de montrer les capacité d'extension de l'outil.

Dans ce travail de thèse, nous nous sommes concentré sur la prise en compte des effets des caches de manière statistique dans le but de reproduire des comportements réalistes. Il serait néanmoins possible de définir de nouveaux *Execution Time Models* visant à tenir compte de l'impact des caches cette fois-ci dans une optique d'évaluation en considérant les effets pire-cas.

Enfin, afin de montrer la capacité d'utilisation de SimSo dans un contexte différent de l'évaluation des politiques d'ordonnancement, nous envisageons de l'intégrer dans une chaîne de conception de systèmes embarqués et de l'utiliser comme un simple simulateur d'architecture logicielle. Pour ce faire, nous pensons proposer une transformation permettant de passer de la description d'une architecture logicielle dans un langage de haut niveau comme AADL ou UML MARTE vers le modèle interne de SimSo. Cette transformation permettra ensuite de jouer une simulation du système afin d'en visualiser le comportement temporel.

Validation des modèles et expérimentations

Les études sur l'influence des aspects opérationnels méritent d'être approfondies. Ce travail consisterait à travailler d'une part sur une étude plus extensive des modèles de comportement des caches et des surcoûts du système, et d'autre part sur des expérimentations plus larges pour étudier l'influence de ces comportements sur les performances des algorithmes. Ces deux points passent par une confrontation des résultats que nous obtenons avec SimSo avec ceux que pourraient produire une architecture réelle (ou du moins un simulateur d'architecture).

Les évaluations menées sur les modèles nous ont permis d'en sélectionner certains pour notre simulateur, mais de nombreuses questions restent ouvertes : peut-on évaluer les coûts de préemption ou de migration à partir des modèles d'accès au cache ? peut-on apporter plus de finesse dans l'évaluation du surcoût des appels système afin de mieux prendre en considération la variabilité temporelle induite par les algorithmes d'ordonnancement ? etc. Pour y répondre il est nécessaire de procéder à de nouvelles expérimentations pour comprendre plus finement le comportement des différents éléments du systèmes et ainsi apporter une plus grande précision dans notre simulateur.

Dans l'objectif d'évaluer les algorithmes en prenant en considération les aspects opérationnels, il conviendrait de mener de plus larges campagnes d'expérimentation en étudiant précisément l'influence des différents paramètres matériels (taille des caches, clusters de processeurs, etc.). Nous pourrions notamment étudier les algorithmes d'ordonnancement dits « cache-aware » (voir section 2.6), c'est-à-dire des politiques qui intègrent dans leur algorithme le comportement des tâches vis-à-vis du cache. Nous n'avons pas implémenté ces politiques car nous nous sommes focalisé sur la validation des modèles de cache dans la simulation, mais ce travail devient maintenant possible. Certains algorithmes nécessitent cependant l'accès à des informations supplémentaires que nous ne fournissons pas encore mais que nous devrions être en mesure de proposer (exemple : valeur des compteurs de performance).

Nouveaux apports théoriques

Enfin, d'un point de vue plus théorique, plusieurs continuations du travail de thèse sont envisageables. Au cours des évaluations des modèles de cache nous avons constaté que les programmes avaient des comportements pouvant fortement varier en fonction de « phases » (chargement des données, traitement, mise à jour, etc.). Cette problématique est connue et de nombreuses méthodes existent pour les détecter automatiquement [SPHC02, VBEC06] ce qui ouvre plusieurs perspectives. D'une part, nous pourrions envisager de modéliser le comportement des travaux en fonction de phases d'exécution afin de représenter les changements de comportement et d'autre part cette connaissance pourrait servir pour proposer de nouveaux algorithmes d'ordonnancement qui optimiseraient la réduction de la concurrence entre les travaux.

Nous nous sommes également limités aux caches de données dans les modèles. Une extension naturelle de ce travail de thèse serait de prendre aussi en compte les caches d'instructions dans la simulation. Ceci est d'autant plus important que certaines architectures embarquées ne disposent que d'un cache d'instructions. Pour cela, le travail sur les modèles nécessite d'être repris, étendu avec de nouveaux comportements et validé expérimentalement.

Pour finir, grâce aux études sur les caches que nous avons menées, il nous semble possible de proposer de nouvelles politiques d'ordonnancement qui prendraient en compte les caractéristiques statistiques du comportement des tâches (comme les SDP, API et CPI). Ces algorithmes consisteraient, par exemple dans le cas partitionné, à allouer les tâches sur les processeurs, voire à fixer les offsets ou les instants de préemption, en tenant compte des effets de la concurrence sur les caches dans le but de les réduire. De même, la « sensibilité » d'une tâche à la concurrence, c'est-à-dire la mesure de l'influence des autres tâches sur son comportement temporel, pourrait être un indicateur à prendre en considération dans des algorithmes en-ligne pour éviter des situations inadéquates.

ANNEXE A

Boite à outils pour l'évaluation

Cette annexe regroupe une liste d'outils en lien avec les travaux présentés dans cette thèse. Ces outils sont divisés en cinq catégories :

- les simulateurs d'ordonnancement ;
- les plate-formes permettant des évaluations sur système physique ;
- les simulateurs d'architecture qui simulent des architectures matérielles (processeurs, caches, etc.) ;
- les logiciels qui permettent de récolter des métriques concernant le comportement mémoire des programmes ;
- les suites de logiciels pour l'évaluation (*benchmarks*).

A.1 Simulation d'ordonnancement

Dans la partie 3.4, les outils MAST, Cheddar, STORM et Yartiss ont été présentés. Néanmoins, il existe d'autres outils que nous avons écarté parce que soit nous n'avons pas pu obtenir le code source, soit nous n'avons pas été en mesure de les faire fonctionner, soit ils ne correspondent pas à notre besoin.

A.1.1 STRESS et PERTS

Parmi les premiers simulateurs d'architecture temps réel, nous retrouvons PERTS et STRESS. L'outil PERTS a été publié en 1993 et permettait de changer l'ordonnanceur utilisé. Cependant, l'objectif n'est pas de tester de nouvelles approches dans la conception du système (en particulier l'ordonnancement), mais plutôt la génération de code applicatif [LLD+96]. STRESS, publié l'année suivante, est un simulateur mais c'est également le nom du langage dédié qui permet de spécifier la configuration du système. La spécification d'une politique d'ordonnancement est possible à travers ce langage [ABRW94].

Ces outils sont aujourd'hui introuvables et il n'est pas sûr qu'ils puissent encore s'exécuter.

A.1.2 GHOST et RTSIM

Quelques années plus tard, l'équipe du Retis Lab de l'École Supérieure Sant'Anna en Italie a développé GHOST, un outil qui se concentre davantage sur la simulation d'ordonnancement. Il permet de définir sa propre politique d'ordonnancement en respectant l'API du simulateur définie en C. Le principal défaut de cet outil est qu'il ne gère pas les architectures multiprocesseurs [SBA97].

La même équipe de recherche est à l'origine d'un autre simulateur, RTSIM, publié l'année suivante, qui vise les systèmes temps réel distribués [CBLL98]. Cet outil est disponible sous licence libre. La dernière version disponible sur leur cite internet [1], au moment de l'écriture de ce manuscrit, date de 2007. Au moment de l'évaluation des outils disponibles, il était impossible de compiler RTSIM et ses dépendances. Le projet semble regagner de l'activité depuis peu à travers le projet T-Res [2] mais le site web de RTSIM n'a pas été mis à jour.

A.1.3 FORTISSIMO

FORTISSIMO est un framework qui vise à faciliter le développement de simulateurs d'ordonnancement temps réel pour des architectures multiprocesseurs. Autrement dit, ce n'est pas un outil prêt à l'emploi, mais plutôt un point de départ pour le développement d'un simulateur. L'intérêt réside dans le fait de permettre le développement de simulateurs spécifiques plus facilement [KAK00].

À l'instar de STORM, l'ordonnanceur n'est pas un composant de l'outil mais un élément extérieur que devra fournir l'utilisateur. Mais là aussi, la simulation se fait pas à pas et l'ordonnanceur est appelé à chaque unité de temps.

Nous n'avons pas été en mesure de trouver le framework sur Internet et nous n'avons pas connaissance de travaux utilisant ce framework.

A.1.4 FORTAS

FORTAS [CG11] est un outil d'évaluation de politiques d'ordonnancement qui repose principalement sur des tests analytiques. Bien que ce ne soit pas sa principale utilité, il est également possible de simuler les systèmes.

FORTAS a été développé à l'ECE à Paris pour les besoins de recherche d'un étudiant en thèse qui a souhaité rendre disponible ce qu'il a produit. Une version est disponible mais sans le code source.

A.1.5 RealtssMP

RealtssMP [DROMA12] est un simulateur d'ordonnancement temps réel de tâches périodiques et indépendantes pour architectures multiprocesseurs développé à l'Institut Technologique de Mexicali. L'implémentation de nouvelles politiques peut se faire en tcl, C ou

1. Disponible à l'adresse : http ://rtsim.sssup.it/
2. Une version plus récente de RTSIM est disponible sur Github.com.

C++. Les résultats de simulations sont exportés dans un format utilisable par le logiciel Kiwi [3] qui permet la visualisation de traces. Un certain nombre d'ordonnanceurs sont disponibles telles que EDF et RM en global et partitionné, ainsi que des variantes simples (TkC, EDF-US, fpEDF, etc). Cet outil n'est cependant pas disponible sur Internet à notre connaissance.

A.2 Exécutions réelles

Cette partie liste les outils librement disponibles pour mener des évaluations sur un système réel.

A.2.1 Linux

Linux permet de traiter depuis longtemps des tâches temps réel, cependant, ce n'est que depuis mars 2014 que le support de EDF a été intégré au noyau, après plus de 5 ans de développement. Lelli *et al.* ont montré qu'il est possible de faire des expérimentations avec EDF (partitionné, global ou clustering) ou RM sur une plate-forme Linux [LFCL12].

Malheureusement, ceci ne permet pas la mise en œuvre simple de nouvelles politiques d'ordonnancement. De plus, rien n'est prévu de manière intégrée pour générer des ensembles de tests ou conduire des expérimentations.

A.2.2 LITMUS^{RT}

LITMUS^{RT} [CLB^{+}06] est une plate-forme d'expérimentation développée par l'équipe de James Anderson à l'Université de Caroline du Nord (University of North Carolina, UNC). Il se présente sous la forme d'une extension du noyau Linux pour permettre l'étude de politiques d'ordonnancement multiprocesseur.

Plusieurs travaux ont été menés en utilisant LITMUS^{RT} afin d'étudier le comportement de politiques d'ordonnancement sur une architecture réelle. LITMUS^{RT} intègre les politiques P-EDF, G-EDF, C-EDF, P-FP et PD^2. D'autres politiques ont été expérimentées sur cette plate-forme, mais n'y sont pas intégrées. C'est le cas par exemple des politiques semi-partitionnées EDF-fm, EDF-WM, NPS-F et RUN.

La prise en main de LITMUS^{RT} n'est pas aisée, la mise en œuvre d'une politique d'ordonnancement doit se faire en espace « kernel » dans Linux, bien que LITMUS^{RT} offre toutefois une abstraction qui permet de simplifier un peu la tâche.

A.2.3 RESCH

RESCH [ANK11], développé par Shinpei Kato dans le cadre du projet AIRS, permet le chargement d'ordonnanceurs temps réel pour Linux. L'ajout de politiques d'ordonnancement peut se faire sous la forme de plugins. Par ailleurs, quelques politiques globales,

3. Kiwi est un visualisateur de trace disponible à cette adresse :
http ://users.dsic.upv.es/grupos/ia/sma/tools/kiwi/index.html

partitionnées et semi-partitionnées sont fournies (G-EDF, EDF-US, G-FP, FP-US, P-FP, P-EDF, EDF-WM, FP-PM).

L'objectif principal de RESCH est de permettre la mise en œuvre d'un ordonnanceur pour Linux de manière simple, sans avoir à toucher directement aux sources du noyau Linux. À l'aide de cet outil, il est ainsi possible de développer et de tester en pratique sur un vrai système une politique d'ordonnancement. L'implémentation d'une politique se fait en langage C.

L'inconvénient principal de ce logiciel est le manque de documentation alors que son utilisation ne semble pas évidente.

A.2.4 ChronOS Linux

ChronOS Linux [DGR11] est un patch pour le noyau Linux développé par le groupe Systems Software Research à Virginia Tech. Les objectifs sont similaires à ceux de LITMUS^{RT} mais ChronOS se différencie par l'utilisation de PREEMPT/RT qui est un patch visant à rendre le noyau Linux temps réel dur.

Les ordonnanceurs monoprocesseur EDF, DASA, LBESA, HVDF et RMA ont été mis en œuvre, ainsi que les politiques multiprocesseur G-EDF, G-NP-EDF, G-FIFO, NG-GUA et G-GUA.

A.3 Simulateurs d'architecture

Les simulateurs d'architectures visent à la fois à émuler le comportement fonctionnel d'une architecture matérielle, mais aussi à simuler les aspects temporels.

A.3.1 Gem5

Gem5 est un simulateur d'architecture, fruit du rapprochement des outils M5 et GEMS [BBB$^+$11]. M5 fournit un environnement hautement configurable pour simuler différents modèles de processeurs avec des jeux d'instructions variés [BDH$^+$06]. GEMS apporte un système mémoire plus détaillé et flexible [MSB$^+$05]. Gem5 supporte les principaux jeux d'instructions : ARM, ALPHA, MIPS, PowerPC, SPARC et x86 (64 bits).

Gem5 vise à se rapprocher temporellement du comportement réel du système simulé [BGOS12]. Le niveau de précision de la simulation dépend des modèles de processeurs et de mémoire utilisés.

Il dispose de deux modes de fonctionnement : full system (FS) et syscall emulation (SE). Le mode FS simule un système complet jusqu'à permettre l'exécution d'un OS complet tel que Linux. Le mode SE évite de devoir modéliser les périphériques et de simuler un OS grâce à l'émulation de la plupart des appels systèmes. Ceci permet de simuler directement un binaire compilé en statique pour Linux.

Ce simulateur dispose d'une communauté importante dans le domaine scientifique et propose un environnement bien documenté.

A.3.2 Simics

Simics [MCE+02] est un simulateur d'architecture développé par Virtutech et racheté en 2010 par Wind River. Bien que sa licence ne soit pas libre, un programme universitaire donnant accès au produit existe.

Les architecture émulées sont nombreuses : x86, IA-64, MIPS, PowerPC, ARM, AMD64, etc. Et un ensemble d'outils accompagne le simulateur pour configurer les architectures, analyser les résultats etc. Enfin, il est possible d'étendre les outils via une API dédiée.

Simics n'est cependant pas *cycle-accurate*, il ne simule pas la micro-architecture. Chaque instruction ne prend qu'un seul cycle processeur à s'exécuter mais il est possible d'étendre Simics avec des bibliothèques pour simuler la micro-architecture, par exemple en ajoutant plusieurs niveau de mémoire cache. Des librairies étaient fournie pour la version 3 (version développée par Virtutech) et des extensions produites par des universitaires. La documentation Winder River sur la version actuelle (4.6) laisse supposer qu'aucun modèle n'est fourni avec.

A.3.3 SESC et ESESC

SESC (SuperESCalar Simulator) est un simulateur d'architecture *cycle-accurate* qui supporte le multicore mais uniquement l'architecture MIPS. Il est développé à l'origine par le groupe de recherche i-acoma de l'Université de L'illinois à Urbana-Champaign (UIUC) puis par d'autres groupes de recherche qui y ont ajouté de nouveaux modèles d'architure. SESC sépare la partie exécution du programme à l'aide d'une émulation de l'architecture MIPS de la partie temporelle. La partie temporelle permet de modéliser les piplines avec prédiction de branchement, les caches, les bus, et tous les éléments des processeurs modernes pour permettre une simulation précise. ESESC[AR13] est une version améliorée de SESC qui se base sur QEMU pour la partie émulation de l'ISA. ESESC vise les architectures ARM et intègre des modèles pour la puissance, la température et les performances.

SESC manque cependant d'une bonne documentation et son seul support de l'architecture MIPS est contraignant. Cependant, ESESC, qui a été publié en 2013, semble très prometteur.

A.3.4 MARSSx86 et PTLSim

PTLSim [You07] est un simulateur d'architecture x86 développé à l'origine par Matt T. Yourst à partir de 2001. Le logiciel a été placé sous licence libre et a ainsi pu être utilisé par des centaines de chercheurs dont en particulier l'équipe de recherche en conception de processeurs de la State University of New York à Binghamton. Contrairement aux autres simulateurs, PTLSim utilise de la virtualisation pour simuler les instructions de manière rapide. Un aspect temporel est ensuite rattaché aux instructions exécutées. PTLSim ne semble plus actif depuis quelques années.

MARSSx86 (Micro-ARchitectural and System Simulator for x86-based Systems) [PACG11], développé par la même équipe de recherche, se base sur PTLSim en l'améliorant et en ajoutant le support du multi-cœur.

A.3.5 Autres

SimpleScalar est un simulateur *cycle-accurate* qui a pour particularité d'être bien documenté et d'avoir été très utilisé dans des travaux de recherches ou pour l'enseignement. Cependant, seul le monoprocesseur est pris en charge et il n'est plus activement développé.

Dinero IV [EH98] développé par University of Wisconsin System permet de simuler des caches sur des architectures monoprocesseurs. Cependant, il n'y a pas de notion de temps associé aux caches.

Trimaran [CGH+05] est un compilateur et simulateur d'architecture dédié pour la recherche sur les architectures informatiques et l'optimisation des compilateurs. Il prend en compte un grand nombre d'architectures de processeurs pour l'embarqué mais le projet ne semble plus actif depuis 2007.

OVPsim est un émulateur d'architecture proposant une API pour spécifier ses propres processeurs, périphériques et modèles de plate-forme. Il est développé par Open Virtual Platforms (OVP) et comprend trois composants : un modèle ouvert, un simulateur et une API. Un grand nombre d'architectures sont prises en considération (ARM, MIPS etc.) avec des modèles déjà pré-configurés. Il n'est pas cycle-accurate.

Sniper [CHE11] est un simulateur d'architecture x86 qui annonce des temps de simulation rapides. L'outil n'est pas déterministe et moins flexible que ne peut l'être Gem5.

Slacksim [CAD09] est un simulateur pour CMP basé sur la notion de Slack. C'est un compromis entre vitesse et précision, ce n'est pas *cycle-accurate*. Développé à l'USC (University of Southern California). L'outil ne semble pas disponible facilement et il n'y a semble-t-il pas de communauté.

A.4 Mesure de SDP et analyse des caches

Plusieurs outils sont capables de calculer le *Stack Distance Profile* d'un programme. La première approche consiste à obtenir la trace des adresses mémoires accédées et d'effectuer le calcul. Dans cette catégorie, on retrouve des outils d'instrumentation basés sur Pin ou Valgrind, mais aussi des simulateurs capables d'exécuter un programme et de sortir une trace (voir outils présentés en section A.3). Ces méthodes souffrent cependant d'un problème de lenteur. Ainsi, il est possible d'effectuer des échantillonnages (StatStack) ou d'approximer les résultats en réduisant artificiellement la taille du cache et en observant le nombre de défauts de cache (Stressmark).

A.4.1 MICA

Intel développe un outil d'instrumentation appelé Pin [4] [LCM+05]. Ce dernier permet d'écrire des « Pin tools » afin d'instrumentaliser l'exécution d'un programme pour une architecture x86.

MICA est un outil utilisant Pin, développé à l'université de Ghent, et dont l'objectif est la caractérisation d'applications indépendamment du matériel. Il a été conçu de façon

4. Disponible à l'adresse :
`http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

modulaire ce qui permet de l'étendre. De base, il est capable de calculer le ratio de chaque type d'instruction, l'empreinte mémoire, ainsi qu'un SDP « compact ». Nous l'avons adapté pour générer un SDP complet.

À l'aide de MICA, nous avons été en mesure d'effectuer de collecter le SDP, l'API ou encore le nombre d'instructions. Nous avons également effectué des expérimentations sur les caches. Cependant, MICA altère la vitesse d'exécution des programmes et il n'est donc pas possible de calculer le CPI ou de voir l'influence de deux programmes s'exécutant en même temps.

A.4.2 Cachegrind

Cachegrind [5] est une extension de Valgrind qui permet d'en faire un « cache profiler ». À noter qu'il est possible d'utiliser Callgrind (une autre extension) de la même manière.

Cachegrind simule le comportement d'une hiérarchie de caches inclusifs à deux niveaux. Sur le premier niveau de cache (L1), les instructions et les données sont séparées (respectivement I1 et D1). Sur le second niveau de cache (LL), instructions et données sont unifiées. La taille des caches, leur associativité et la taille des lignes est configurable facilement.

Babka a modifié le code de Cachegrind afin d'obtenir la séquence des lignes de cache utilisées et ainsi pouvoir générer un SDP [BLMT12].

A.4.3 Stressmark Workload

Stressmark [XCDM10b] fait en sorte qu'une partie du cache soit fortement utilisée ce qui réduit la taille de cache disponible pour l'application que l'on souhaite analyser. En faisant varier la taille de cache disponible et en obtenant le nombre de défauts de cache en utilisant des compteurs de performance matériels, on est en mesure de déterminer le SDP.

A.4.4 StatStack

Les auteurs de la méthode StatStack partent du constat que la mesure de SDP en utilisant Pin ou Valgrind provoque un ralentissement de 100 à 1000 fois afin de collecter la trace complète des adresses accédées [EH10]. Afin d'éviter un tel ralentissement, StatStack se base sur un échantillonnage de *reuse distances*. À partir de ces données, un calcul statistique est mené pour déterminer le SDP.

A.5 Benchmarks

Afin de pouvoir évaluer et comparer les performances d'un ensemble de méthodes ou modèles, il est nécessaire de disposer de benchmarks. L'ensemble de benchmarks le plus utilisé semble être celui du Standard Performance Evaluation Corporation CPU (SPECCPU2006).

5. Documentation disponible à l'adresse : `http://valgrind.org/docs/manual/cg-manual.html`

Ce dernier regroupe un ensemble varié de programmes permettant d'évaluer les performances sur différents types de calcul : traitement multimédia, réseau, compilation, chiffrement, calcul flottant, etc.

Dans le domaine temps réel, il existe plusieurs benchmarks mais aucun ne s'est réellement imposé au sein de l'ensemble de la communauté. Cela est d'autant plus difficile que les critères de performances sont mal définis [6] : overhead, ordonnancement, charge, etc. Comme nous le verrons dans cette partie, plusieurs tentatives ont été faites pour proposer des benchmarks orientés métier. Ces benchmarks proposent des ensembles de programmes répertoriés en fonction de leur domaine d'utilisation (automobile, traitement d'image, etc.). Parallèlement, la communauté scientifique travaillant sur l'estimation du WCET a fait un effort pour uniformiser son travail d'évaluation proposant plusieurs suites de benchmarks.

A.5.1 Benchmarks orientés temps réel

a) Hartstone Uniprocessor Benchmark (HUB)

Au début des années 1990, une suite baptisée Hartstone Uniprocessor Benchmark (HUB) [WK92] a été proposée afin de réaliser des évaluations d'applications temps réel. Elle comprend des ensembles d'applications composées de tâches dont le comportement d'activation varie (périodique, apériodique, serveur, etc.) . Les évaluations visées était plutôt axée sur les politiques d'ordonnancement. Cette suite ne semble pas avoir été beaucoup utilisée et elle est difficile à obtenir.

b) Real-Time Micro Benchmark Suite (rtmb)

L'ensemble des benchmarks nommés Real-Time Micro Benchmark Suite (rtmb) est dédié pour les supports Linux. Il a été réalisé par trois ingénieurs d'IBM. La version la plus récente date d'août 2010. L'objectif principal de cette suite est de fournir un ensemble de benchmarks qui peut être exécuté à partir de plusieurs languages (C, C++, Java). Cet ensemble semble réellement dédié temps réel et propose des tests orienté services : activation, partage de ressource, thread, temps, etc. Il est assez peu référencé et documenté.

c) RT_STAP Benchmark

Le benchmark Real-Time Space-Time Adaptive Processing (RT_STAP) [CTW97] est utilisé pour évaluer l'évolutivité des applications embarqués temps réel. Il se base sur une application pour radar aéroporté. L'évolutivité est étudiée en faisant varié la complexité des algorithmes pouvant être utilisés. Ce benchmark n'est pas libre.

A.5.2 Benchmarks orientés embarqués

a) EDN Embedded Microprocessor Benchmark Consortium (EEMBC)

Dans le domaine des systèmes temps réel, peu de benchmarks existent et sont en général confondus avec ceux pour les systèmes embarqués. L'EDN Embedded Microprocessor

6. N. Weiderman a proposé dès 1989 dans [Wei90] des critères pour évaluer les systèmes temps réel, mais cela ne semble pas avoir eu d'écho.

Benchmark Consortium (EEMBC) propose des séries de benchmarks pour évaluer le logiciel et le matériel dans le domaine de l'embarqué. Chaque série est dédiée à un domaine particulier : automobile, traitement d'images, multimédia, consommation d'énergie, applications Java pour mobiles, réseau, bureautique, multi-processeur et télécommunication. Une série supplémentaire existe pour uniquement tester les performances des coeurs des architectures. Tous les programmes benchmarks sont écrits en C ou en Java. Cependant, l'accès à ces benchmarks sont payants même pour des universitaires (de l'ordre de 200$ par série).

b) MiBench

MiBench [GRE+01] suit la même logique que l'EEMBC en proposant des suites en fonction de domaines dédiés à l'embarqué. Bien que ne semblant pas très actif, les codes proposés sont disponibles gratuitement et la plupart des benchs sont encore capable de compiler et s'exécuter sur un système Linux récent. Plus récemment, ParMiBench [ILG10] propose une extension pour les architectures multi-processeur. Les benchmarks sont des versions parallélisées de certaines applications de MiBench.

A.5.3 Benchmarks orientés WCET

a) Mälardalen WCET

Un effort pour proposer des benchmarks a aussi été fait dans la communauté scientifique traitent de l'évaluation des pire temps d'exécution (WCET). L'ensemble de programmes mis à disposition par l'équipe de recherche WCET de l'université de Mälardalen [GBEL10] a été collecté depuis 2005 auprès de différents chercheurs travaillant dans le domaine des temps d'exécution. Il se compose d'une quarantaine de programmes dont plusieurs analyses ont déjà été réalisées.

b) WCET Tool Challenge 2008

Lors du WCET Tool Challenge de 2008 [HGB+08] plusieurs benchmarks ont été proposés pour faire de l'évaluation de WCET. Quatre ont été proposés par l'université de Saarland et portent le nom de rathijit. L'objectif de ces benchmarks est d'avoir une empreinte mémoire importante en cache d'instruction et de données. Le programme utilise une macro en C avec des boucles pour que le préprocesseur C génère un code de taille conséquente.

Le benchmark *debie1* est un logiciel écrit en C pour une architecture 8051 basé sur une application embarquée dans un satellite. Originellement conçu par Space Systems Finland, il a été adapté par N. Holsti de Tidorum Ltd dans le cadre d'ARTIST2. Il est composé de six tâches dont les activations sont conduites par les événements. Son utilisation n'est pas complètement ouverte, mais il est possible d'en demander une version pour des expérimentations.

c) PapaBench

PapaBench est un benchmark temps réel embarqué qui est issu d'un logiciel de contrôle de drone appelé Paparazzi. Il est proposé par l'équipe TRACES de l'Institut de Recherche

en Informatique de Toulouse sous licence GNU. Initialement prévu pour une architecture biprocesseur AVR, le code source a été adapté pour d'autres plateformes.

Plus de détails sur l'architecture sont fournis dans l'article [NCS+06]. PapaBench a notamment été utilisé lors du WCET Tool Challenge de 2011 [vHHL+11].

d) SNU Real-Time Benchmarks

La suite SNU Real-Time Benchmarks est constituée d'un ensemble de petits programmes écrits en C servant à faire de l'analyse de pire temps d'exécution. Les algorithmes sont en majorité dédiés au calcul numérique et pour les DSP tels que la transformée de Fourier rapide, le calcul de la suite de Fibonacci, le tri par insertion, etc. Les structures de ces programmes sont relativement simples car limitées par le compilateur du concepteur (gotocc) de ces benchmarks.

A.5.4 Conclusion sur les benchmarks

La majorité des benchmarks proposés sont sous la forme d'un ensemble de programmes relativement petits n'implémentant que des fonctionnalités simples. Ils peuvent être pertinents pour étudier des points précis (WCET, cache, etc.) pour les applications temps réel, mais ne représentent pas la complexité de ces systèmes. Le benchmark PapaBench est certainement le plus complet et réaliste pour mener une étude complète. MiBench offre de son côté un nombre intéressant de programmes et est facilement utilisable.

Pour choisir et utiliser une suite de benchmarks, il est avant tout nécessaire de bien préciser le cadre et les objectifs de l'évaluation à mener. Seuls ces critères permettent d'effectuer un choix pertinent. Pour les systèmes temps réel les critères d'évaluation peuvent être extrêmement larges : performances du logiciel sur le matériel, overheads de l'OS, adaptation de l'ordonnanceur, estimation du pire temps d'exécution, etc.

Dans le cadre de l'étude sur l'influence du cache sur l'ordonnancement, il semble nécessaire de différencier les deux : d'une part utiliser des benchmarks pour estimer les modèles de cache et d'autre part y ajouter l'ordonnancement. MiBench Automobile, la suite Mälardalen WCET, voire SNU Real-Time Benchmarks pourraient servir de base pour sélectionner des programmes de référence pour évaluer individuellement les modèles de cache. Dans un second temps, ces programmes pourront être utilisés pour alimenter des évaluations sur le partage de cache avant de de passer sur une expérimentation avec PapaBench.

Bibliographie

[AB08] B. Andersson et K. Bletsas : Sporadic multiprocessor scheduling with few preemptions. *In Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 243 –252, 7 2008.

[ABB08] B. Andersson, K. Bletsas et S. Baruah : Scheduling arbitrary-deadline sporadic task systems on multiprocessors. *In Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS)*, pages 385 –394, 2008.

[ABD05] J.H. Anderson, V. Bud et U. Devi : An EDF-based scheduling algorithm for multiprocessor soft real-time systems. *In Proceedings in the 17th Euromicro Conference on Real-Time Systems, 2005. (ECRTS 2005).*, pages 199 – 208, 07 2005.

[ABJ01] B. Andersson, S. Baruah et J. Jonsson : Static-priority scheduling on multiprocessors. *In Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 193 – 202, 12 2001.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell et A. J. Wellings : Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[ABRW94] N. C. Audsley, A. Burns, M. F. Richardson et A. J. Wellings : STRESS : A simulator for hard real-time systems. *Software : Practice and Experience*, 24(6):543–564, 1994.

[ACD06] J.H. Anderson, J. Calandrino et U. Devi : Real-time scheduling on multicore platforms. *In Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, 2006.*, pages 179 – 190, 4 2006.

[ACP02] G. Almási, C. Caşcaval et D. A. Padua : Calculating stack distances efficiently. *In Proceedings of the 2002 workshop on Memory system performance (MSP)*, pages 37–43. ACM, 2002.

[ADLD14] S. Altmeyer, R. Douma, W. Lunniss et R. Davis : Evaluation of cache partitioning for hard real-time systems. *In Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[AEDC14] J. H Anderson, J. P Erickson, U. C Devi et B. N Casses : Optimal semi-partitioned scheduling in soft real-time systems. *In Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014.

[AHH89] A. Agarwal, J Hennessy et M Horowitz : An analytical cache model. *ACM Transactions on Computer Systems (TOCS)*, 7(2):184–215, 1989.

[AJ00] B. Andersson et J. Jonsson : Fixed-priority preemptive multiprocessor scheduling : to partition or not to partition. *In Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 337–346, 2000.

[And03] B. Andersson : Static-priority scheduling on multiprocessors, 2003.

[And08] B. Andersson : Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. *In Principles of Distributed Systems*, volume 5401 de *Lecture Notes in Computer Science*, pages 73–88. Springer, 2008.

[ANK11] M. Asberg, T. Nolte et S. Kato : A loadable task execution recorder for hierarchical scheduling in linux. *In Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

[AR13] E. K. Ardestani et J. Renau : ESESC : A fast multicore simulator using time-based sampling. *In Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 448–459, Washington, DC, USA, 2013. IEEE Computer Society.

[ARM10] CoreLink Level 2 Cache Controller L2C-310. Rapport technique, ARM, 2010.

[AS99] J.H. Anderson et A. Srinivasan : A new look at pfair priorities. Rapport technique, TR00-023, University of North Carolina at Chapel Hill, 9 1999.

[AS00a] J.H. Anderson et A. Srinivasan : Early-release fair scheduling. *Euromicro Conference on Real-Time Systems*, page 35, 2000.

[AS00b] J.H. Anderson et A. Srinivasan : Pfair scheduling : beyond periodic task systems. *In Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 297–306, 2000.

[AT06] B. Andersson et E. Tovar : Multiprocessor scheduling with few preemptions. *In Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322 –334, 2006.

[Aud91] N. C Audsley : *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* 1991.

[BA09] K. Bletsas et B. Andersson : Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *In Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 447 –456, 12 2009.

[Bar03] S. Baruah : Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24:93–128, 2003.

[Bar04] S. Baruah : Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781 – 784, 6 2004.

[BB01] E. Bini et G. Buttazzo : A hyperbolic bound for the rate monotonic algorithm. *In Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 59 –66, 2001.

[BB02] E. Bini et G. Buttazzo : The space of rate monotonic schedulability. *In Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 169 – 178, 2002.

[BB05] E. Bini et G. C. Buttazzo : Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2), 2005.

[BBA11] A. Bastoni, B.B. Brandenburg et J.H. Anderson : Is semi-partitioned scheduling practical ? *In Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill et D. A. Wood : The gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[BCGM99] S. Baruah, D. Chen, S. Gorinsky et A. Mok : Generalized multiframe tasks. *Real-Time Systems*, 17:5–22, 1999.

[BCL05a] M. Bertogna, M. Cirinei et G. Lipari : Improved schedulability analysis of EDF on multiprocessor platforms. *In Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 209–218, July 2005.

[BCL05b] M. Bertogna, M. Cirinei et G. Lipari : New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. *In Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, pages 306–321, 2005.

[BCPV96] S. Baruah, N. Cohen, C. Plaxton et D. Varvel : Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[BDH+06] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi et S.K. Reinhardt : The m5 simulator : Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[BFO14] A. Bertout, J. Forget et R. Olejnik : Minimizing a real-time task set through task clustering. *In Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS)*, pages 23 :23–23 :31. ACM, 2014.

[BG03a] S. Baruah et J. Goossens : Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966 – 970, 7 2003.

[BG03b] S. Baruah et J. Goossens : Scheduling real-time tasks : Algorithms and complexity, 2003.

[BGOS12] A. Butko, R. Garibotti, L. Ost et G. Sassatelli : Accuracy evaluation of gem5 simulator system. *In Proceedings of the 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–7, 2012.

[BGP95] S. K Baruah, J. E Gehrke et C G. Plaxton : Fast scheduling of periodic tasks on multiple resources. *In International Symposium on Parallel Processing*, pages 280–280. IEEE Computer Society, 1995.

[BH04] E. Berg et E. Hagersten : Statcache : a probabilistic approach to efficient and accurate data locality analysis. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[BK75] B. T. Bennett et V. J. Kruskal : Lru stack processing. *IBM J. Res. Dev.*, 19(4):353–357, juillet 1975.

[BLMT12] V. Babka, P. Libič, T. Martinec et P. Tůma : On the accuracy of cache sharing models. *In Proceedings of the third joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*, 2012.

[BNVT14] A. Baldovin, G. Nelissen, T. Vardanega et E. Tovar : SPRINT : Extending RUN to Schedule Sporadic Tasks. *In Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[BP12] B. Berna et I. Puaut : PDPA : period driven task and cache partitioning algorithm for multi-core systems. *In Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS)*, 2012.

[But05] G. Buttazzo : Rate Monotonic vs. EDF : Judgment Day. *Real-Time Systems*, 29:5–26, 2005.

[BV13] G. Bilardi et F. Versaci : Optimal eviction policies for stochastic address traces. *Theor. Comput. Sci.*, 514, 2013.

[CA08] J.M. Calandrino et J.H. Anderson : Cache-aware real-time scheduling on multicore platforms : Heuristics and a case study. *In Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 299–308, 2008.

[CAB07] J. M. Calandrino, J. H. Anderson et D. P. Baumberger : A hybrid real-time scheduling approach for large-scale multicore platforms. *In Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 247–258, 2007.

[CAD09] J. Chen, M. Annavaram et M. Dubois : SlackSim : A platform for parallel simulations of CMPs on CMPs. *SIGMETRICS Perform. Eval. Rev.*, 37(2): 77–78, octobre 2009.

[Cal09] J. M. Calandrino : *On the Design and Implementation of a Cache-aware Soft Real-time Scheduler for Multicore Platforms*. Thèse de doctorat, Chapel Hill, NC, USA, 2009. AAI3366308.

[CBLL98] A. Casile, G. C. Buttazzo, G. Lamastra et G. Lipari : A scheduling simulator for real-time distributed systems. *In Proceedings of the 15th workshop Distributed Computer Control Systems, Como, Italy*, 1998.

[CDH13] M. Chéramy, A.-M. Déplanche et P.-E. Hladik : Simulation of real-time multiprocessor scheduling with overheads. *In Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 2013.

[CDRPR00] C. Cascaval, L. De Rose, D. A. Padua et D. A. Reed : Compile-time based performance prediction. *In Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 365–379. Springer-Verlag, 2000.

[CFH+04] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson et S. Baruah : A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on Scheduling Algorithms, Methods, and Models, pages*, pages 30–1, 2004.

[CFM+12] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet et M. Qamhieh : Yartiss : A tool to visualize, test, compare and evaluate real-time scheduling algorithms. *In Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2012.

[CG11] P. Courbin et L. George : FORTAS : Framework for real-time analysis and simulation. *In 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2011.

[CGH+05] L. N. Chakrapani, J. Gyllenhaal, W. Hwu, S.A. Mahlke, K. V. Palem et R. M. Rabbah : Trimaran : An infrastructure for research in instruction-level parallelism. *In Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC)*, pages 32–41. Springer-Verlag, 2005.

[CGJ97] E. Coffman, Jr., M. Garey et D. Johnson : *Approximation algorithms for bin packing : a survey*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[CGKS05] D. Chandra, F. Guo, S. Kim et Y. Solihin : Predicting inter-thread cache contention on a chip multi-processor architecture. *In Proceedings of the*

11th International Symposium on High-Performance Computer Architecture (HPCA), 2005.

[CHD13] M. Chéramy, P.-E. Hladik et A.-M. Déplanche : Simulation d'ordonnancement temps réel avec prise en compte de l'impact des caches. *In Proceedings of Ecole d'Eté Temps Réel (ETR)*, 2013.

[CHD14] M. Chéramy, P.-E. Hladik et A.-M. Déplanche : SimSo : A simulation tool to evaluate real-time multiprocessor scheduling algorithms. *In Proceedings of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014.

[CHDD14] M. Chéramy, P.-E. Hladik, A.-M. Déplanche et S. Dubé : Simulation of real-time scheduling with various execution time models. *In Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES), Work-in-Progress session*, 2014.

[CHE11] T. E. Carlson, W. Heirman et L. Eeckhout : Sniper : Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. *In International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, novembre 2011.

[CLB+06] J. Calandrino, H. Leontyev, A. Block, U. Devi et J. Anderson : LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. *In Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, 2006.

[CMV14] D. Compagnin, E. Mezzetti et T. Vardanega : Putting RUN into practice : implementation and evaluation. *In Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[CPRBM03] AM. Campoy, A Perles, F. Rodriguez et J. V. Busquets-Mataix : Static use of locking caches vs. dynamic use of locking caches for real-time systems. *In Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, volume 2, pages 1283–1286 vol.2, May 2003.

[CR06] H. Cho et B. Ravindran : An optimal real-time scheduling algorithm for multiprocessors. *In Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 101–110, 2006.

[CTW97] K. C. Cain, J. A. Torres et R. T. Williams : Rt_stap : Real-time space-time adaptive processing benchmark. Rapport technique, DTIC Document, 1997.

[DA05] U. Devi et J. Anderson : Tardiness bounds under global EDF scheduling on a multiprocessor. *In Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, page 12, 12 2005.

[DB09] R. Davis et A. Burns : Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *In Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[DB11] R. I. Davis et A. Burns : A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35 :1–35 :44, octobre 2011.

[Den68] P. J. Denning : The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[DGR11] M. Dellinger, P. Garyali et B. Ravindran : ChronOS Linux : A best-effort real-time multiprocessor linux kernel. *In Proceedings of the 48th Design Automation Conference (DAC)*, pages 474–479. ACM, 2011.

[DK12] R. Davis et S. Kato : FPSL, FPCL and FPZL schedulability analysis. *Real-Time Systems*, 48(6):750–788, 2012.

[DL78] S. Dhall et C. Liu : On a real-time scheduling problem. *Operations Research*, 26(1):pp. 127–140, 1978.

[DNS94] M. Di Natale et J. A Stankovic : Dynamic end-to-end guarantees in distributed real time systems. *In Real-Time Systems Symposium, 1994., Proceedings.*, pages 216–227. IEEE, 1994.

[DROMA12] A. Diaz-Ramirez, D.K. Orduno et P. Mejia-Alvarez : A multiprocessor real-time scheduling simulation tool. *In Proceedings of the 22nd International Conference on Electrical Communications and Computers (CONIELECOMP)*, pages 157–161, Feb 2012.

[DZ13] Y. Ding et W. Zhang : Multicore real-time scheduling to reduce inter-thread cache interferences. *Journal of Computing Science and Engineering*, 7(1):67–80, 2013.

[EA12] J.P. Erickson et J.H. Anderson : Fair Lateness Scheduling : Reducing Maximum Lateness in G-EDF-Like Scheduling. *In Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[EBSH11] D. Eklov, D. Black-Schaffer et E. Hagersten : Fast modeling of shared caches in multicore systems. *In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, 2011.

[EH98] J. Edler et M. D Hill : Dinero iv trace-driven uniprocessor cache simulator, 1998.

[EH10] D. Eklov et E. Hagersten : StatStack : efficient modeling of LRU caches. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.

[ESD10] P. Emberson, R. Stafford et R.I. Davis : Techniques for the synthesis of multiprocessor tasksets. *In Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.

[ESL09] A. Easwaran, I. Shin et I. Lee : Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[FKY08] K. Funaoka, S. Kato et N. Yamasaki : Work-conserving optimal real-time scheduling on multiprocessors. *In Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[FN09] S. Funk et V. Nanadur : LRE-TL : An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets. *In* L. George, M. Chetto et M. Sjodin, éditeurs : *Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 159–168, Paris, France, 2009.

[FQ12] M. Fan et G. Quan : Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. *In Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 503–508, San Jose, CA, USA, 2012. EDA Consortium.

[FSS06] A. Fedorova, M. Seltzer et M.D. Smith : Cache-fair thread scheduling for multicore processors. Rapport technique TR-17-06, Division of Engineering and Applied Sciences, Harvard University, 2006.

[FT96] R. Fromm et N. Treuhaft : Revisiting the cache interference costs of context switching. *Computer Science Division, University of California-Berkeley*, 1996.

[Fun04] S. Funk : *EDF scheduling on heterogeneous multiprocessors*. Thèse de doctorat, University of North Carolina at Chapel Hill, 2004.

[GBEL10] J. Gustafsson, A. Betts, A. Ermedahl et B. Lisper : The mälardalen wcet benchmarks - past, present and future. *In Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[GBF02] J. Goossens, S. Baruah et S. Funk : Real-time scheduling on multiprocessors. *In Proceedings of the 10th International Conference on Real-Time System (RTS)*, 2002.

[GFB02] J. Goossens, S. Funk et S. Baruah : EDF scheduling on multiprocessor platforms : some (perhaps) counterintuitive observations. *In Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2002.

[GFB03] J. Goossens, S. Funk et S. Baruah : Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[GRE+01] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge et R.B. Brown : Mibench : A free, commercially representative embedded benchmark suite. *In Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[GSYY09] N. Guan, M. Stigge, W. Yi et G. Yu : Cache-aware scheduling and analysis for multicores. *In Proceedings of the 7th ACM international conference on Embedded Software (EMSOFT)*, 2009.

[GSYY10] N. Guan, M. Stigge, W. Yi et G. Yu : Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. *In Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 165 –174, 4 2010.

[HA05] P. Holman et J. Anderson : Adapting Pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1:543–564, December 2005.

[HE07] K. Hoste et L. Eeckhout : Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3), 2007.

[HGB+08] N. Holsti, J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. de Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat et M. Schordan : WCET 2008 – report from the tool challenge 2008 – 8th intl. workshop on worst-case execution time (WCET) analysis. *In* Raimund Kirner, éditeur : *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 8 de *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[HGD+13] M. González Harbour, J. J. Gutiérrez, J. M. Drake, P. López Martínez et J. C. Palencia : Modeling distributed real-time systems with MAST 2. *Journal of Systems Architecture*, 59(6):331 – 340, 2013.

[HGGM01] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez et J. M. Drake Moyano : Mast : Modeling and analysis suite for real time applications. *In Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, 2001.

[HGT99] J. Hildebrandt, F. Golatowski et D. Timmermann : Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems. *In Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 208–215, 1999.

[HL94] R. Ha et J. W S Liu : Validating timing constraints in multiprocessor and distributed real-time systems. *In Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 162–171, Jun 1994.

[HP12] J. L. Hennessy et D. A. Patterson : *Computer architecture : a quantitative approach*. Elsevier, 2012.

[HS89] M.D. Hill et AJ. Smith : Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec 1989.

[ILG10] S.M.Z. Iqbal, Yuchen L. et H. Grahn : ParMiBench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, 9(2):45–48, Feb 2010.

[Int08] Intel 64 and IA-32 Architectures Optimization Reference Manual. Rapport technique, Intel, 2008.

[JHA02] R. Jain, C.J. Hughes et S.V. Adve : Soft real-time scheduling on simultaneous multithreaded processors. *In Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 134–145, 2002.

[KAK00] T. Kramp, M. Adrian et R. Koster : *An open framework for real-time scheduling simulation*. Springer, 2000.

[KBS+09] F. Konig, D. Boers, F. Slomka, U. Margull, M. Niemetz et G. Wirrer : Application specific performance indicators for quantitative evaluation of the timing behavior for embedded real-time systems. *In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2009.

[KC11] H. Kim et Y. Cho : A new fair scheduling algorithm for periodic tasks on multiprocessors. *Information Processing Letters*, 111(7):301 – 309, 2011.

[KCS04] S. Kim, D. Chandra et Y. Solihin : Fair cache sharing and partitioning in a chip multiprocessor architecture. *In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122. IEEE Computer Society, 2004.

[Kir89] D.B. Kirk : SMART (strategic memory allocation for real-time) cache design. *In Real Time Systems Symposium, 1989., Proceedings.*, pages 229–237, Dec 1989.

[Kor03] R. Korf : An improved algorithm for optimal bin packing. *In Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1252–1258. Morgan Kaufmann Publishers Inc., 2003.

[KS97] A. Khemka et R.K. Shyamasundar : An optimal multiprocessor real-time scheduling algorithm. *Journal of Parallel and Distributed Computing*, 43(1):37 – 45, 1997.

[KS04] T.S. Karkhanis et J.E. Smith : A first-order superscalar processor model. *In Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 338 – 349, june 2004.

[KY07] S. Kato et N. Yamasaki : Real-time scheduling with task splitting on multiprocessors. *In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 441 –450, 08 2007.

[KY08a] S. Kato et N. Yamasaki : Global EDF-based scheduling with efficient priority promotion. *In Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 197–206, Aug 2008.

[KY08b] S. Kato et N. Yamasaki : Portioned EDF-based scheduling on multiprocessors. *In Proceedings of the 8th ACM international conference on Embedded software (EMSOFT)*, pages 139–148. ACM, 2008.

[KY08c] S. Kato et N. Yamasaki : Portioned static-priority scheduling on multiprocessors. *In Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1 –12, 4 2008.

[KY08d] S. Kato et N. Yamasaki : Semi-partitioning technique for multiprocessor real-time scheduling, 2008.

[KY09] S. Kato et N. Yamasaki : Semi-partitioned fixed-priority scheduling on multiprocessors. *In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 23–32, April 2009.

[KYI09] S. Kato, N. Yamasaki et Y. Ishikawa : Semi-partitioned scheduling of sporadic task systems on multiprocessors. *In 21st Euromicro Conference on Real-Time Systems*, pages 249–258. IEEE Computer Society, 2009.

[LCM+05] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi et K. Hazelwood : Pin : building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, juin 2005.

[LDS07] C. Li, C. Ding et K. Shen : Quantifying the cost of context switch. *In Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS)*, 2007.

[Lee94] S. K. Lee : On-line multiprocessor scheduling algorithms for real-time tasks. *In Proceedings of the IEEE Region 10's Ninth Annual International Conference*, 1994.

[LES10] J. Lee, A. Easwaran et I. Shin : LLF schedulability analysis on multiprocessor platforms. *In Proceedings of the IEEE 31st Real-Time Systems Symposium (RTSS)*, pages 25–36, Nov 2010.

[LFCL12] J. Lelli, D. Faggioli, T. Cucinotta et G. Lipari : An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software*, 85(10), 2012.

[LFS+10] G. Levin, S. Funk, C. Sadowski, I. Pye et S. Brandt : DP-FAIR : A Simple Model for Understanding Optimal Multiprocessor Scheduling. *In 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3 –13, 7 2010.

[LGS+08] F. Liu, F. Guo, Y. Solihin, S. Kim et A. Eker : Characterizing and modeling the behavior of context switch misses. *In Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, 2008.

[LL73] C. L. Liu et J. Layland : Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, 1 1973.

[LL85] C. C. Lee et D. T. Lee : A simple on-line bin-packing algorithm. *J. ACM*, 32:562–572, July 1985.

[LLD+96] J. WS Liu, C. L. Liu, Z. Deng, T.-S. Tia, J. Sun, M. Storch, D. Hull, JL Redondo, R. Bettati et A Silberman : PERTS : A prototyping environment for real-time systems. *International Journal of Software Engineering and Knowledge Engineering*, 6(02):161–177, 1996.

[LLD+08] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang et P. Sadayappan : Gaining insights into multicore cache partitioning : Bridging the gap between

simulation and real systems. *In Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 367–378, Feb 2008.

[LR14] A. Lindsay et B. Ravindran : On cache-aware task partitioning for multicore embedded real-time systems. *In Proceedings of the 11th IEEE International Conference on Embedded Software and Systems*, 2014.

[LRL09] K. Lakshmanan, R. Rajkumar et J. Lehoczky : Partitioned fixed-priority preemptive scheduling for multi-core processors. *In 21st Euromicro Conference on Real-Time Systems*, pages 239–248, Washington, DC, USA, 2009. IEEE Computer Society.

[LS10] F. Liu et Y. Solihin : Understanding the behavior and implications of context switch misses. *ACM Transactions on Architecture and Code Optimization*, 7(4):21 :1–21 :28, décembre 2010.

[LSD89] J. Lehoczky, L. Sha et Y. Ding : The rate monotonic scheduling algorithm : exact characterization and average case behavior. *In Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 166 –171, 12 1989.

[LW82] J. Leung et J. Whitehead : On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[MB91] J. C. Mogul et A. Borg : The effect of context switches on cache performance. *SIGOPS Oper. Systems Review*, 25, 1991.

[MC96] A. Mok et D. Chen : A multiframe model for real-time tasks. *In Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 22 –29, 12 1996.

[MCE+02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt et B. Werner : Simics : A full system simulation platform. *Computer*, 35(2), 2002.

[MGST70] R.L. Mattson, J. Gecsei, D.R. Slutz et I.L. Traiger : Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[ML10] E. Massa et G. Lima : A bandwidth reservation strategy for multiprocessor real-time scheduling. *In Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 175–183, April 2010.

[MLD05] G. Muller, J.L. Lawall et H. Duchesne : A framework for simplifying the development of kernel schedulers : design and performance evaluation. *In Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, pages 56–65, Oct 2005.

[MLR+14] E. Massa, G. Lima, P. Regnier, G. Levin et S. Brandt : Optimal and adaptive multiprocessor real-time scheduling : The quasi-partitioning approach. *In Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[Mok83] A. Mok : Fundamental design problems of distributed systems for the hard-real-time environment. Rapport technique, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[MSB+05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill et D. A. Wood : Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, novembre 2005.

[NBN+12] G. Nelissen, V. Berten, V. Nelis, J. Goossens et D. Milojevic : U-EDF : An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. *In Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–23, 2012.

[NCS+06] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun et M. De Michiel : Papabench : a free real-time benchmark. *In In WCET '06*, 2006.

[NFG12] G. Nelissen, S. Funk et J. Goossens : Reducing preemptions and migrations in EKG. *In Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[NS13] F. Ndoye et Y. Sorel : Sustainable multiprocessor real-time scheduling with exact preemption cost. *International Journal On Advances in Systems and Measurements*, 6(3 and 4):353–363, 2013.

[NSG+14] G. Nelissen, H. Su, Y. Guo, D. Zhu, V. Nélis et J. Goossens : An optimal boundary fair scheduling. *Real-Time Systems*, pages 1–53, 2014.

[OB98] D.-I. Oh et T. Bakker : Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15:183–192, 1998.

[OY98] S.-H. Oh et S.-M. Yang : A modified least-laxity-first scheduling algorithm for real-time tasks. *In Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 31 –36, oct 1998.

[PACG11] A. Patel, F. Afram, S. Chen et K. Ghose : Marss : A full system simulator for multicore x86 cpus. *In Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1050–1055, 2011.

[PHK+05] M. Park, S. Han, H. Kim, S. Cho et Y. Cho : Comparison of deadline-based scheduling algorithms for periodic real-time tasks on multiprocessor. *IEICE transactions on information and systems*, 88(3):658–661, 03 2005.

[PLM09] S. Plazar, P. Lokuciejewski et P. Marwedel : WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. *In* Niklas Holsti, éditeur : *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 de *OpenAccess Series in Informatics (OASIcs)*, pages 1–11. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009.

[PS01] P. Pillai et K. G. Shin : Real-time dynamic voltage scaling for low-power embedded operating systems. *In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102. ACM, 2001.

[RCM96] I. Ripoll, A. Crespo et A. Mok : Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1), 1996.

[RLM+11] P. Regnier, G. Lima, E. Massa, G. Levin et S. Brandt : RUN : Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. *In Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2011.

[SB97] S.M. Smith et J.M. Brady : SUSAN - a new approach to low level image processing. *Int. Journal of Computer Vision*, 23(1):45–78, May 1997.

[SB02] A. Srinivasan et S. Baruah : Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.*, 84:93–98, 10 2002.

[SBA97] F. Sensini, G. C. Buttazzo et P. Ancilotti : GHOST : A tool for simulation and analysis of real-time scheduling algorithms. *In In Proceedings of the IEEE Real-Time Educational Workshop*, pages 42–49, 1997.

[SBSH11] A. Sandberg, D. Black-Schaffer et E. Hagersten : A simple statistical cache sharing model for multicores. *In Proceedings of the 4th Swedish Workshop on Multi-Core Computing*, pages 31–36. Linköping University, 2011.

[SBSH12] A. Sandberg, D. Black-Schaffer et E. Hagersten : Efficient techniques for predicting cache sharing and throughput. *In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 305–314. ACM, 2012.

[SD72] J. R. Spirn et P. J. Denning : Experiments with program locality. *In Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*. ACM, 1972.

[SDR02] G.E. Suh, S. Devadas et L. Rudolph : A new memory monitoring scheme for memory-aware scheduling and partitioning. *In Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 117 – 128, feb. 2002.

[SEGY11] M. Stigge, P. Ekberg, N. Guan et W. Yi : The digraph real-time task model. *In Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 71 –80, 4 2011.

[SEL08] I. Shin, A. Easwaran et I. Lee : Hierarchical scheduling framework for virtual clustering of multiprocessors. *In Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 181–190. IEEE, 2008.

[Sha04] P. Shaw : A constraint for bin packing. *In* Mark Wallace, éditeur : *Principles and Practice of Constraint Programming*, volume 3258 de *Lecture Notes in Computer Science*, pages 648–662. Springer, 2004.

[Sim12] SimPy Developer Team : http ://simpy.sourceforge.net/, 2012.

[SLNM04] F. Singhoff, J. Legrand, L. Nana et L. Marcé : Cheddar : a flexible real time scheduling framework. *Ada Lett.*, XXIV(4), 2004.

[SM08] V. Suhendra et T. Mitra : Exploring locking & partitioning for predictable shared caches on multi-cores. *In Proceedings of the 45th Annual Design Automation Conference (DAC)*, pages 300–303. ACM, 2008.

[SPHC02] T. Sherwood, E. Perelman, G. Hamerly et B. Calder : Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, octobre 2002.

[SSTB13] P. Baltarejo Sousa, P. Souto, E. Tovar et K. Bletsas : The Carousel-EDF scheduling algorithm for multiprocessor systems. *In Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013.

[Sta88] J. A. Stankovic : Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10):10–19, octobre 1988.

[Tin94] K. Tindell : Adding time-offsets to schedulability analysis. Rapport technique, University of York, England, 1994.

[TSRB14] H. N. Tran, F. Singhoff, S. Rubini et J. Boukhobza : Instruction cache in hard real-time systems : modeling and integration in scheduling analysis tools with aadl. *In Proceedings of the 12th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, 2014.

[TSW08] T.M. Taha et D. Scott Wills : An instruction throughput model of superscalar processors. *IEEE Transactions on Computers*, 57(3):389 –403, march 2008.

[UDT10] R. Urunuela, A.-M. Déplanche et Y. Trinquet : STORM a simulation tool for real-time multiprocessor scheduling evaluation. *In Proceedings of the Emerging Technologies and Factory Automation (ETFA)*, 2010.

[VBEC06] M. Van Biesbrouck, L. Eeckhout et B. Calder : Considering all starting points for simultaneous multithreading simulation. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 143–153, 2006.

[vHHL+11] R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, A. Bonenfant, H. Cassé, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N. M. Islam, D. Kästner, R. Kirner, L. Kovacs, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda et J. Zwirchmayr : WCET tool challenge 2011 : Report. *In* Chris Healy, éditeur : *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*. OCG, July 2011.

[WA12] J. Whitham et N.C. Audsley : Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. *In Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–12, April 2012.

[Wei90] N. Weiderman : Hartstone : Synthetic benchmark requirements for hard real-time applications. *In Proceedings of the Working Group on Ada Performance Issues*, pages 126–136. ACM, 1990.

[WGG10] K. Wehrle, Mesut Güneş et J. Gross : *Modeling and tools for network simulation*. Springer, 2010.

[WK92] N. H. Weiderman et N. I. Kamenoff : Hartstone uniprocessor benchmark : Definitions and experiments for real-time systems. *Real-Time Syst.*, 4(4): 353–382, décembre 1992.

[XCDM10a] C. Xu, X. Chen, R.P. Dick et Z.M. Mao : Cache contention and application performance prediction for multi-core systems. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 76–86, 2010.

[XCDM10b] C. Xu, X. Chen, R.P. Dick et Z.M. Mao : Cache contention and application performance prediction for multi-core systems. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 76–86, 2010.

[YA14] K. Yang et J. H Anderson : Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. *In Proceedings of the 12th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, 2014.

[You07] M.T. Yourst : PTLsim : A cycle accurate full system x86-64 microarchitectural simulator. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 23–34, 2007.

[ZMM03] D. Zhu, D. Mosse et R. Melhem : Multiple-resource periodic scheduling problem : how much fairness is necessary ? *In 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 142 – 151, dec. 2003.

[ZQMM11] D. Zhu, X. Qi, D. Mossé et R. Melhem : An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, 71(10):1411 – 1425, 2011.

Title : Study and evaluation of real-time multiprocessor scheduling policies.

The study of real-time scheduling has regained interest during the last twenty years with the widespread use of multiprocessor architectures. Initially, uniprocessor scheduling algorithms were adapted by partitioned or global approaches, but they do not allow to use these architectures optimally. Thus, new algorithms have emerged using new strategies such as fairness, semi-partitioning, clustering, etc. We have identified more than fifty scheduling algorithms, and over half of them have been published during the last decade.

With such an abundance of algorithms, it has become difficult to evaluate and compare them. Furthermore, most research work focus on algorithmic and theoretical aspects, ignoring the hardware architecture. But, the computation time of the jobs is impacted by the hardware and more particularly in the case of a multiprocessor architecture. Among the hardware components, caches have a significant impact on the execution of the jobs and their effectiveness may depend strongly on the choices of the scheduler. In addition, new questions related to the implementation arise, such as the choice of the processor running the scheduler.

In this context, this work proposes to evaluate the main scheduling policies for real-time multiprocessor architectures while taking into account the execution platform. The first contribution is the design of a simulation tool, SimSo, dedicated to the evaluation of scheduling policies. We show that its design was guided by the desire to make the experimentations easy while ensuring strong adaptability of the execution models and the ability to describe mechanisms close to realistic systems. More than twenty-five scheduling algorithms have been implemented and evaluated with SimSo. The experiments conducted helped to consolidate or revise experimental results already known by the community but in known, controlled and identical conditions.

The second contribution of this work is the integration in a simulator of statistical cache models in order to simulate the computation times of the jobs on multiprocessor architectures. To that end, various existing models have been studied and evaluated using architecture simulators. From these studies, we selected a set of models that we have implemented in SimSo. A last work shows how these models can be used in the evaluation of the performance of scheduling policies.

Auteur : Maxime Chéramy

Titre : Étude et évaluation de politiques d'ordonnancement temps réel multiprocesseur

Directeurs de thèse : Pierre-Emmanuel Hladik et Anne-Marie Déplanche

Date et lieu de soutenance : 11/12/2014 à Toulouse

Discipline : Informatique 4200018

Laboratoire : LAAS-CNRS

Résumé

De multiples algorithmes ont été proposés pour traiter de l'ordonnancement de tâches temps réel dans un contexte multiprocesseur. Encore très récemment de nouvelles politiques ont été définies. Ainsi, sans garantie d'exhaustivité, nous en avons recensé plus d'une cinquantaine. Cette grande diversité rend difficile une analyse comparée de leurs comportements et performances. L'objectif de ce travail de thèse est de permettre l'étude et l'évaluation des principales politiques d'ordonnancement existantes. La première contribution est SimSo, un nouvel outil de simulation dédié à l'évaluation des politiques. Grâce à cet outil, nous avons pu comparer les performances d'une vingtaine d'algorithmes. La seconde contribution est la prise en compte, dans la simulation, des surcoûts temporels liés à l'exécution du code de l'ordonnanceur et à l'influence des mémoires caches sur la durée d'exécution des travaux par l'introduction de modèles statistiques évaluant les échecs d'accès à ces mémoires.

Mots clés : ordonnancement, temps réel, multiprocesseur, multicœur, évaluation, cache

Résumé en anglais

Numerous algorithms have been proposed to address the scheduling of real-time tasks for multiprocessor architectures. Yet, new scheduling algorithms have been defined very recently. Therefore, and without any guarantee of completeness, we have identified more than fifty of them. This large diversity makes the comparison of their behavior and performance difficult. This research aims at allowing the study and the evaluation of key scheduling algorithms. The first contribution is SimSo, a new simulation tool dedicated to the evaluation of scheduling algorithms. Using this tool, we were able to compare the performance of twenty algorithms. The second contribution is the consideration, in the simulation, of temporal overheads related to the execution of the scheduler and the impact of memory caches on the computation time of the jobs. This is done by the introduction of statistical models evaluating the cache miss ratios.

Keywords : scheduling, real-time, multiprocessor, multicore, evaluation, cache