Analyse de la consistance mémoire dans les MPSoCs à l'aide du prototypage virtuel

Damien Hedde

▶ **To cite this version:**

Damien Hedde. Analyse de la consistance mémoire dans les MPSoCs à l'aide du prototypage virtuel. Systèmes embarqués. Université de Grenoble, 2013. Français. NNT : 2013GRENM079 . tel-01551796

HAL Id: tel-01551796

https://theses.hal.science/tel-01551796v1

Submitted on 30 Jun 2017

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.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Damien Hedde

Thèse dirigée par **Frédéric Pétrot**

préparée au sein **du TIMA**
et de **l'École Doctorale Mathématiques, Sciences et Techniques de l'Ingénieur, Informatique**

Analyse de la consistance mémoire dans les MPSoCs à l'aide du prototypage virtuel

Thèse soutenue publiquement le **12 juin 2013**,
devant le jury composé de :

M. Philipe Clauss
Professeur des Universités, Université de Strasbourg, Président
M. Daniel Etiemble
Professeur des Universités, Université Paris Sud, Rapporteur
M. Pascal Sainrat
Professeur des Universités, Université Paul Sabatier, Rapporteur
M. Luis-Miguel Santana
Docteur, STMicroelectronics, Examinateur
M. Frédéric Pétrot
Professeur des Universités, Grenoble INP, Directeur de thèse

Remerciements

Je souhaiterais remercier tout ceux qui m'ont permis de réaliser cette expérience qu'est la thèse. Beaucoup de personnes ont contribué, de près ou de loin, à rendre ce travail possible. J'espère n'en oublier aucune.

Je tiens tout d'abord remercier mon directeur de thèse, Frédéric Pétrot, de m'avoir fait confiance en 2008 et confié ce travail de recherche. Il m'a soutenu et conseillé tout au long de mes travaux. Je le remercie aussi pour la patience dont il a fait preuve.

Je souhaite aussi remercier Daniel Etiemble et Pascal Sainrat, mes deux rapporteurs, d'avoir accepté et pris le temps de s'acquitter de cette tâche.

Je remercie Philippe Clauss d'avoir pris part à mon jury et fait le déplacement jusqu'à Grenoble.

Je remercie aussi Luis-Miguel Santana, le responsable de notre partie dans le projet Nano 2012. Merci aussi d'avoir participé à mon jury.

Je remercie l'équipe SLS du laboratoire TIMA de m'avoir accueilli pendant ces quelques années. Merci, en particulier, aux inconditionnels de la pause thé (ou café, selon les années) pour les discussions intéressantes que nous avons eues, leurs conseils et la bonne humeur qui en ressortait. Je remercie tout spécialement Nicolas Fournel, qui a accepté de relire ma thèse, pour ces remarques avisées.

Je remercie aussi toutes les personnes qui m'ont permis de réaliser de l'enseignement et en particulier Olivier Müller, Christophe Durand et Jean-Luc Amalberti.

Merci aussi à mes parents pour les dernières corrections d'orthographes. Finalement je remercie chaleureusement ma famille et mes amis de m'avoir soutenu durant ces années.

Table des matières

Liste des figures xi

Liste des tableaux xiii

Liste des algorithmes xv

Liste des acronymes xvii

Introduction 1
Contexte . . . 2
Problématique générale . . . 2
Organisation de la thèse . . . 3

1 Problématique 5
1.1 Contexte de la thèse . . . 6
1.1.1 Systèmes multi-processeurs à mémoire partagée . . . 6
1.1.1.1 Architecture matérielle . . . 6
1.1.1.2 Pile logicielle . . . 7
1.1.2 Simulation de systèmes multi-processeurs . . . 7
1.2 Consistance mémoire . . . 8
1.2.1 Notations utilisées . . . 9
1.2.2 Consistance séquentielle . . . 9
1.2.3 Optimisations et conséquences . . . 10
1.2.3.1 Exécution désordonnée . . . 10
1.2.3.2 Tampons d'écriture . . . 11
1.2.3.3 Mémoire cache . . . 11
1.2.4 Modèles de consistance mémoire . . . 12
1.2.4.1 Modèles atomiques . . . 12
1.2.4.2 Impact du réordonnancement . . . 13
1.2.4.3 Modèles avec écritures anticipées . . . 14
1.2.4.4 Impact des écritures anticipées . . . 15
1.2.4.5 Modèles non atomiques cohérents . . . 16
1.2.4.6 Impact des écritures non atomiques . . . 17
1.2.4.7 Impact de la cohérence mémoire . . . 17
1.2.4.8 Résumé des modèles . . . 18
1.2.5 Conclusion sur la consistance mémoire . . . 19
1.3 Vérification de la consistance mémoire . . . 19
1.3.1 Principe de la vérification . . . 19
1.3.1.1 Dépendances du programme . . . 19
1.3.1.2 Sémantique de la mémoire . . . 20

1.3.2 Problèmes liés à la vérification . . . 21
1.3.2.1 Collecte des dépendances . . . 21
1.3.2.2 Quantité d'informations . . . 21
1.3.3 Conclusion sur la vérification de la consistance mémoire . . . 22
1.4 Extraction des informations depuis la simulation . . . 22
1.4.1 Besoin d'informations « transversales » . . . 22
1.4.2 Impact sur la simulation . . . 22
1.4.3 Conclusion . . . 23
1.5 Conclusion de la problématique . . . 23

2 État de l'art **25**
2.1 Formalisation de la vérification de la consistance mémoire . . . 26
2.1.1 Un problème d'ordre . . . 26
2.1.2 Formalisation générale . . . 26
2.1.3 Formalismes existants . . . 27
2.2 Représentation d'une exécution . . . 27
2.2.1 Utilisation d'un graphe . . . 27
2.2.2 Gestion de l'atomicité . . . 28
2.2.2.1 Représentation des accès atomiques . . . 28
2.2.2.2 Représentation des écritures anticipées . . . 29
2.2.2.3 Représentation des écritures non atomiques . . . 31
2.2.3 La notion de case mémoire . . . 32
2.2.4 Les accès spéciaux . . . 32
2.3 Vérification de la consistance mémoire . . . 32
2.3.1 Vérification formelle de la consistance mémoire . . . 33
2.3.1.1 Preuve par raisonnement . . . 33
2.3.1.2 *Model checking* . . . 33
2.3.2 Complexité de la vérification de la consistance mémoire . . . 33
2.3.2.1 Complexité de la vérification de la consistance séquentielle . . 33
2.3.2.2 Complexité générale . . . 34
2.3.3 Simplification du problème . . . 34
2.3.3.1 Utilisation de l'association des lectures . . . 34
2.3.3.2 Utilisation d'informations temporelles . . . 35
2.3.3.3 Utilisation de l'ordre des écritures . . . 35
2.3.4 Vérification dynamique . . . 36
2.3.4.1 Vérification dynamique intégrée . . . 36
2.3.4.2 Vérification dynamique en simulation . . . 36
2.4 Conclusion . . . 37

3 Méthode de vérification dynamique de la consistance mémoire **39**
3.1 Stratégie de vérification dynamique de la consistance mémoire . . . 40
3.1.1 Utilisation d'un graphe . . . 40
3.1.1.1 Principe . . . 40
3.1.1.2 Notations . . . 41
3.1.2 Création d'un graphe dynamique . . . 41
3.1.3 Limitation de la taille du graphe . . . 42
3.1.4 Détection des violations du modèle de consistance . . . 43
3.1.4.1 Besoin d'une détection dynamique . . . 43
3.1.4.2 Détection dynamique « moindre effort » . . . 44
3.1.4.3 Détection dynamique « régulière » . . . 45

3.1.5 Bilan sur l'utilisation d'un graphe dynamique 45
3.2 Génération dynamique de graphes partiels 45
3.2.1 Nœuds du graphe 46
3.2.2 Nœuds « G » 46
3.2.2.1 Réduction de la taille du graphe 46
3.2.2.2 Gestion d'ensembles de nœuds 46
3.2.3 Ordre total selon une séquence 48
3.2.4 Dépendances entre deux catégories d'événements 49
3.2.4.1 Dépendances simples 49
3.2.4.2 Variantes 49
3.2.5 Ordres combinant plusieurs types de dépendance 50
3.2.5.1 Dépendances « RaR + WaR » 50
3.2.5.2 Dépendances « RaW + WaW + WaR » 51
3.2.6 Ordres utilisant des événements supplémentaires 51
3.2.7 Bilan sur les graphes partiels 52
3.3 Génération de l'ensemble du graphe 52
3.3.1 Combinaison pour représenter l'ordre des accès à la mémoire 53
3.3.1.1 Accès à plusieurs cases mémoire 53
3.3.1.2 Accès non atomique à une case mémoire 53
3.3.2 Combinaison pour représenter les dépendances du programme 53
3.3.3 Bilan sur les combinaisons 55
3.4 Conclusion de la génération dynamique du graphe 56

4 Algorithme efficace de vérification dynamique de la consistance mémoire 57
4.1 Représentation d'un SMPMP 58
4.1.1 Architecture générale 58
4.1.2 Organisation de la mémoire 58
4.2 Algorithme de vérification dynamique de la consistance mémoire 59
4.2.1 Principe général de l'algorithme 59
4.2.1.1 Notations utilisées 60
4.2.1.2 Entrées de l'algorithme 60
4.2.1.3 Algorithme général 61
4.2.2 Suppression des nœuds 62
4.2.3 Exécution des instructions : ordre du programme 63
4.2.3.1 Émission des accès 63
4.2.3.2 Dépendances de la SI 65
4.2.4 Complétion des accès : ordre de la mémoire 66
4.2.5 Disparition des accès 68
4.2.6 Gestion des cas particuliers 68
4.2.6.1 Gestion des accès spéciaux 68
4.2.6.2 Synchronisation et non-atomicité 70
4.3 Bilan de l'algorithme 72
4.3.1 Bilan sur la taille du graphe 72
4.3.1.1 Cas des accès I/Os, atomiques ou anticipés 73
4.3.1.2 Cas des écritures non atomiques 74
4.3.2 Initialisation de l'algorithme 74
4.3.3 Occupation mémoire en pire cas 74
4.3.4 Complexité algorithmique 75
4.3.5 Bilan préliminaire 76
4.4 Optimisation de l'occupation mémoire 76

4.4.1 Taille d'une case mémoire 76
4.4.2 Nœuds isolés 78
4.4.2.1 Le problème des nœuds isolés 78
4.4.2.2 Suppression temporaire des nœuds isolés 79
4.4.3 Références invalides ou isolées 79
4.4.3.1 Références utilisées pour l'ordre du programme 79
4.4.3.2 Références utilisées pour l'ordre de la mémoire : les accès partiels 80
4.4.4 Bilan des optimisations 81
4.5 Conclusion sur la vérification de la consistance mémoire 81

5 Environnement de trace et d'analyse pour la simulation 83
5.1 Objectifs de l'environnement de trace et d'analyse 84
5.1.1 Extraction d'informations d'un SMPMP 84
5.1.1.1 Extraction depuis un système réel 84
5.1.1.2 Extraction depuis un prototype virtuel 84
5.1.2 Objectifs 85
5.1.3 Environnement unifié de trace et d'analyse 85
5.2 Système de trace 86
5.2.1 Exemple de SMPMP 86
5.2.2 Principe général 86
5.2.2.1 Utilisation de ports de trace 86
5.2.2.2 Architecture générale 86
5.2.3 Relations entre les événements 87
5.2.3.1 Des relations réelles de cause à effet 87
5.2.3.2 Utilisation d'identifiants 88
5.2.3.3 Absence de réciprocité des relations 89
5.2.4 Conclusion sur le mécanisme de trace 90
5.3 Bibliothèque générique d'analyse 90
5.3.1 Principe général 90
5.3.1.1 Les composants 90
5.3.1.2 Informations accessibles 91
5.3.1.3 Traitement centré sur les événements 91
5.3.1.4 Définition des traitants 92
5.3.2 Ordonnancement des traitants 92
5.3.2.1 Contraintes initiales 92
5.3.2.2 Contraintes additionnelles 93
5.3.2.3 Complexité de l'environnement 95
5.4 Conclusion sur l'environnement de trace et d'analyse 95

6 Expérimentations 97
6.1 Implantations 98
6.1.1 Instrumentation d'un prototype virtuel 98
6.1.1.1 Instrumentation des composants 98
6.1.1.2 Interfaces instrumentées 99
6.1.2 Implantation de l'environnement de trace et d'analyse 99
6.1.2.1 Architecture logicielle 99
6.1.2.2 Système de trace 100
6.1.2.3 Environnement d'analyse 101
6.1.2.4 Traitement général d'un événement 102
6.1.3 Implantation de l'algorithme de vérification de la consistance mémoire . 102

6.1.3.1 Utilisation de l'environnement de trace et d'analyse 102
6.1.3.2 Description de l'implantation réalisée 104
6.2 Description des expérimentations 105
6.2.1 Plate-forme matérielle simulée 105
6.2.2 Logiciel embarqué sur la plate-forme 107
6.2.3 Résultats des simulations 107
6.2.3.1 Comportement des applications 107
6.2.3.2 Durées des simulations non-instrumentées 108
6.2.3.3 Événements générés 109
6.2.4 Détection des violations du modèle de consistance mémoire 110
6.2.4.1 Vérification d'un modèle plus contraint 110
6.2.4.2 Erreur de consistance 110
6.2.4.3 Erreur dans les accès atomiques 110
6.3 Évaluation du coût de la vérification de la consistance mémoire 111
6.3.1 Impact sur les simulations 111
6.3.1.1 Impact global de la vérification de la consistance mémoire . . . 111
6.3.1.2 Génération et traitement des événements 111
6.3.2 Coût de la vérification de la consistance mémoire 112
6.3.2.1 Évolution linéaire de la vérification de la consistance mémoire 113
6.3.2.2 Opérations effectuées par l'algorithme 114
6.3.2.3 Impact de la méthode de détection des cycles 115
6.3.2.4 Impact du modèle de consistance 115
6.3.3 Occupation mémoire 115
6.3.3.1 Évolution de l'occupation mémoire 115
6.3.3.2 Impact des paramètres architecturaux 116
6.4 Conclusion 118

Conclusion **119**

Bibliographie **123**

A Coût de la détection des cycles dans le graphe dynamique **129**
A.1 Méthodes de détection 129
A.1.1 Récapitulatif 129
A.1.2 Notations 129
A.2 Étude de la complexité 130
A.2.1 Taille du graphe 130
A.2.2 Coût de la détection 130
A.2.3 Complexité 131

B Fonctions de traitement des accès **133**
B.1 Segment I/O 133
B.2 Segment atomique 134
B.3 Segment non atomique 135

C Description des événements tracés **137**
C.1 Contenu commun 137
C.2 *spread* 137
C.3 exception 137
C.4 instruction 138
C.5 accès 138

C.6 acquittement 138

D Résultats des simulations **139**
D.1 Expérimentations MJPEG_1 139
D.2 Expérimentation MJPEG_2 141
D.3 Expérimentation MJPEG_3 142
D.4 Expérimentation MJPEG_4 143
D.5 Expérimentation OCEAN_4 144
D.6 Expérimentation OCEAN_8 145
D.7 Expérimentation OCEAN_16 146
D.8 Expérimentation OCEAN_32 147

Liste des figures

1 Prévision de l'ITRS de l'évolution des systèmes non portables 1

1.1 Architecture TilePro64 6
1.2 Couches logicielles 7
1.3 Temps d'accès à différents niveaux de caches 11
1.4 Représentation schématique d'architectures SC et atomique 12
1.5 Représentation schématique d'une architecture autorisant les écritures anticipées 14
1.6 Représentation schématique d'une architecture non atomique 16

2.1 Graphe de $\leq_d$. 28
2.2 Graphes de $\leq_m$ et $\leq$ correspondant à l'exemple d'exécution. 29
2.3 Graphes de $\leq_m$ et $\leq$ si $R_{1.3}$ lit $W_{1.1}$. 29
2.4 Graphe de $\leq$ sans les dépendances RAW entre accès de la même SI. 29
2.5 Graphes utilisant des nœuds pour représenter les écritures visibles localement. . . 30
2.6 Graphes globaux sans nœuds pour les écritures locales. 30

3.1 Graphe complet de l'exemple d'exécution SC 41
3.2 Graphe dynamique d'une exécution 42
3.3 Graphes équivalents illustrant le gain de taille obtenu avec un nœud intermédiaire 47
3.4 Ajout d'un nœud dans un groupe 47
3.5 Encadrement d'un ensemble de nœuds avec deux nœuds supplémentaires 47
3.6 Construction de l'ordre total 48
3.7 Construction des ordres RAR et WAW 49
3.8 Construction de l'ordre RAW 49
3.9 Construction d'une variante de l'ordre RAW 50
3.10 Construction de l'ordre « RAR+WAR » 51
3.11 Construction de l'ordre des accès à une case mémoire 51
3.12 Représentation des contraintes d'ordre d'une barrière mémoire 52
3.13 Génération des graphes d'accès simultanés à plusieurs cases mémoire 53
3.14 Génération du graphe des accès non atomiques à une case mémoire 54
3.15 Dépendances du programme via les cases mémoire 54
3.16 Dépendances du programme via les registres 55

4.1 Architecture d'un Système Multi-Processeurs à Mémoire Partagée 58
4.2 Graphes partiels en fonction de la case mémoire 59
4.3 Graphes partiels créés lors de l'émission d'un accès 64
4.4 Accès atomiques partiels à deux cases mémoire 64
4.5 Graphes partiels créés lors de la complétion d'un accès 67
4.6 Ordre de la mémoire pour chaque catégorie 73
4.7 Début d'un ordre RAW+WAW+WAR 74
4.8 Impact de la taille d'une case mémoire sur le graphe 77

4.9 Persistance d'un nœud *G* dans l'ordre d'une case mémoire. 78

5.1 Vue globale de l'architecture de l'environnement 85
5.2 Système matériel utilisé pour les exemples 86
5.3 Architecture du mécanisme de trace 87
5.4 Exemples de relations de cause à effet 88
5.5 Événement ayant un nombre dynamique de conséquences 89

6.1 Architecture logicielle de la simulation d'un prototype virtuel instrumenté 100
6.2 Ordres entre les traitants paramétrables par l'analyseur 101
6.3 Traitements effectués sur trois événements durant leurs durées de vie 102
6.4 Localisation des traitement VCM parmi les traitants 103
6.5 Plate-forme utilisée dans les expérimentations 106
6.6 Comportement des applications exécutées 108
6.7 Durées initiales des simulations effectuées 108
6.8 Débit moyen d'événements générés pendant les simulations 109
6.9 Taux de génération d'événements durant la simulation 110
6.10 Coût de traitement par événement 112
6.11 Évolution de la génération d'accès pour *OCEAN_32* 113
6.12 Évolution du temps des simulations pour *OCEAN_32* 113
6.13 Coût moyen de l'algorithme de VCM par accès 114
6.14 Statistiques par accès 114
6.15 Évolution de la mémoire utilisée lors de la simulation 115
6.16 Maxima atteints lors des simulations 116
6.17 Impact de la taille des caches sur l'occupation mémoire 117
6.18 Impact des tampons sur l'occupation mémoire 117

B.1 Graphe partiel d'une case mémoire d'un segment « I/O » 133
B.2 Graphe partiel d'une case mémoire d'un segment « atomique » 134
B.3 Graphe partiel d'une case mémoire d'un segment « non atomique » 135

D.1 Durée des différentes simulations pour MJPEG_1 139
D.2 Évolution de l'algorithme de VCM pour MJPEG_1 140
D.3 Durée des différentes simulations pour MJPEG_2 141
D.4 Évolution de l'algorithme de VCM pour MJPEG_2 141
D.5 Durée des différentes simulations pour MJPEG_3 142
D.6 Évolution de l'algorithme de VCM pour MJPEG_3 142
D.7 Durée des différentes simulations pour MJPEG_4 143
D.8 Évolution de l'algorithme de VCM pour MJPEG_4 143
D.9 Durée des différentes simulations pour OCEAN_4 144
D.10 Évolution de l'algorithme de VCM pour OCEAN_4 144
D.11 Durée des différentes simulations pour OCEAN_8 145
D.12 Évolution de l'algorithme de VCM pour OCEAN_8 145
D.13 Durée des différentes simulations pour OCEAN_16 146
D.14 Évolution de l'algorithme de VCM pour OCEAN_16 146
D.15 Durée des différentes simulations pour OCEAN_32 147
D.16 Évolution de l'algorithme de VCM pour OCEAN_32 147

Liste des tableaux

1.1 Programme sensible au réordonnancement d'une écriture suivie d'une lecture . . 14
1.2 Programme sensible aux écritures anticipées 15
1.3 Programme d'illustration des accès non atomiques 17
1.4 Programme d'illustration des contraintes d'ordre de la cohérence 17
1.5 Programme d'illustration de la propagation des écritures dans un système cohérent 18
1.6 Propriétés de différentes architectures cohérentes 18

2.1 Exécution d'accès à [x]. 28

3.1 Exemple d'exécution d'un programme 40
3.2 Dépendances induites de l'exécution en SC 40

6.1 Événements générés par les simulations 109
6.2 Impact de la VCM sur le temps de simulation 111

Liste des algorithmes

3.1 Algorithme de détection des cycles dans un graphe 44
4.1 Algorithme général équivalent 62
4.2 Algorithme de suppression des nœuds 63
4.3 Traitement général d'une instruction 63
4.4 Émission d'un accès à un segment de catégorie « anticipée » 65
4.5 Mise en place des dépendances via des registres 66
4.6 Complétion d'un accès « anticipé » 67
4.7 Disparition d'un accès « anticipé » 68
4.8 Traitement des accès LdL et StC 70
4.9 Gestion des barrières mémoires 71
4.10 Synchronisation quand les écritures sont non atomiques 71
5.1 Réception d'un ordre d'exécution « cause vers effet » 93
5.2 Réception d'un ordre d'exécution « effet vers cause » 94
5.3 Maintien de l'ordre de la trace lors de l'exécution des traitants initiaux 94
B.1 Émission d'un accès à un segment « I/O » 133
B.2 Complétion d'un accès « I/O » 133
B.3 Émission d'un accès à un segment « atomique » 134
B.4 Complétion d'un accès « atomique » 134
B.5 Disparition d'un accès « atomique » 134
B.6 Émission d'un accès à un segment « non atomique » 135
B.7 Complétion d'un accès « non atomique » 136
B.8 Disparition d'un accès « non atomique » 136

Liste des acronymes

API Interface de programmation (*Application Programming Interface*).
CA Précision au cycle (*Cycle Accurate*).
DBT Traduction binaire dynamique (*Dynamic Binary Translation*).
DMA Accès direct en mémoire (*Direct Memory Access*).
FIFO Premier entré, premier sorti (*First In First Out*).
I/O Entrée/Sortie (*Input/Ouput*).
ISS Simulateur de jeu d'instructions (*Instruction Set Simulator*).
LdL Instruction *Load Linked*.
MCM Modèle de Consistance Mémoire.
MPSoC Système multi-processeurs sur puce (*Multiprocessor System-on-Chip*).
NoC Réseau sur puce (*Network-on-Chip*).
OS Système d'exploitation (*Operating System*).
PSO Consistance *Partial Store Order*.
RaR Lecture après lecture (*Read after Read*).
RaW Lecture après écriture (*Read after Write*).
RMO Consistance *Released Memory Order*.
RTL Niveau de transfert de registre (*Register Transfer Level*).
SC Consistance séquentielle (*Sequential Consistency*).
SDRAM Type de mémoire *Synchronous Dynamic Random Access Memory*.
SI Séquence d'Instructions.
SIMD Instruction unique pour plusieurs données (*Single Instruction on Multiple Data*).
SL Niveau système (*System Level*).
SMP Technique de parallélisme *Symmetric MultiProcessing*.
SMPMP Système Multi-Processeurs à Mémoire Partagée.
SMT Technique de parallélisme *Simultaneous MultiThreading*.
StC Instruction *Store Conditional*.
TLB Unité de traduction des adresses (*Translation Lookaside Buffer*).
TLM Modélisation au niveau des transactions (*Transaction Level Modeling*).
TSO Consistance *Total Store Order*.
VCM Vérification de la Consistance Mémoire.
VDCM Vérification Dynamique de la Consistance Mémoire.
VLIW Mot d'instruction long (*Very Long Instruction Word*).
WaR Écriture après lecture (*Write after Read*).
WaW Écriture après écriture (*Write after Write*).

Introduction

Aujourd'hui, les systèmes informatiques sont majoritairement parallèles. Si les systèmes professionnels tels que les serveurs et calculateurs haute performance le sont depuis longtemps, les systèmes personnels ne le sont que depuis beaucoup plus récemment [1]. Aujourd'hui la plupart des systèmes informatiques grands publics (ordinateurs de bureau, ordinateurs portables, *smartphones*, etc.) sont équipés de processeurs incluant au moins deux cœurs. Beaucoup d'entre eux en possèdent même plus de quatre et le nombre de cœurs continue d'augmenter.

Cette évolution est dictée par le besoin d'augmentation des performances et de l'efficacité énergétique ; il faut dire que l'augmentation de la fréquence de fonctionnement, solution longtemps privilégiée, est limitée par la technologie et entraîne une très grosse augmentation de la consommation. La conséquence est une démocratisation des Systèmes Multi-Processeurs à Mémoire Partagée (SMPMPs) et du modèle de programmation associé. Contrairement au modèle de programmation adapté à la mémoire distribuée, les programmes n'ont pas besoin d'utiliser de bibliothèques spéciales de communication pour pouvoir échanger des données. Le modèle de programmation de la mémoire partagée permet aux différents programmes de communiquer et d'échanger des données à travers l'espace mémoire commun auquel chacun peut accéder directement.

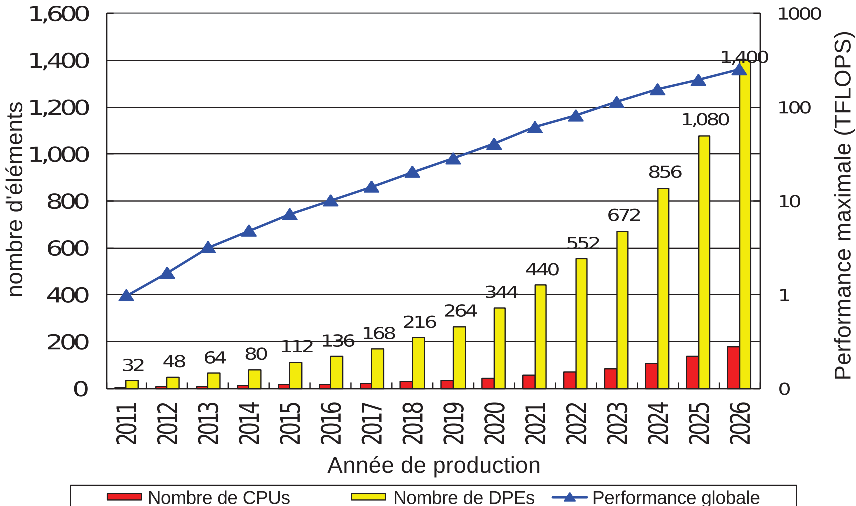


FIGURE 1 - Prévision de l'évolution des systèmes non portables (source ITRS 2011 [ITR11]). Les *CPUs* sont des processeurs génériques tandis que les *DPEs* (*Data Processing Elements*) sont des processeurs spécialisés pour le calcul intensif sur les données.

1. IBM a commercialisé en 2001 le premier processeur multi-cœurs, un POWER4, à destination du marché des stations de travail.

Contexte : la simulation des systèmes multi-processeurs à mémoire partagée

Le parallélisme au sein des SMPMPs va continuer d'augmenter dans les prochaines années. Les prévisions de l'*International Technology Roadmap for Semiconductor* (ITRS) pour les prochaines années sont indiquées dans la FIGURE 1. Le graphique illustre le cas des systèmes non portables pour lesquels la consommation n'est pas cruciale. Le cas des systèmes portables est toutefois similaire selon l'ITRS.

L'augmentation du parallélisme dans les systèmes tend à rendre les programmes parallèles afin d'augmenter leurs performances. Les systèmes exécutent ainsi plusieurs programmes qui sont chacun constitués de plusieurs *threads*.

Dans un système à mémoire partagée, les *threads*, bien qu'ils peuvent accéder directement à la mémoire commune, utilisent aussi des bibliothèques logicielles de synchronisation. Elles sont utiles, en particulier pour éviter les problèmes liés aux *data races* [2]. Ces événements, à cause de l'indéterminisme qu'ils génèrent, peuvent en effet avoir des effets dévastateurs (résultat erroné, interblocage, etc.) lors de l'exécution d'un programme. L'utilisation de bibliothèque permet de regrouper les fonctions nécessaires à une exécution correcte des programmes dans une unique structure et permet de fournir des objets abstraits dont la sémantique est claire. Le programme repose dans ce cas sur un modèle de programmation et de synchronisation assuré par cette bibliothèque qui permet de masquer les spécificités du système qui exécute le programme. Un standard, globalement utilisé pour cela dans les systèmes UNIX, est la norme POSIX.

Dans le cadre du développement des SMPMPs, le prototypage virtuel est actuellement utilisé de manière généralisée. Celui-ci est nécessaire pour vérifier que les caractéristiques du système correspondent bien aux prévisions [Lud+02]. Plusieurs caractéristiques peuvent ainsi être vérifiées, par exemple :

- la fonctionnalité d'une partie ou de l'ensemble du système,
- la performance et l'efficacité.

Ces tests sont réalisés durant toutes les phases de développement, des phases préliminaires de définition de l'architecture aux phases de génération des circuits contenus dans les puces électroniques. Différents niveaux de représentation du système, adaptés aux différentes phases, sont utilisés à cette fin.

L'utilisation du prototypage virtuel et en particulier de la simulation a plusieurs avantages principaux. Un prototype virtuel permet en effet de réaliser des estimations bien avant de disposer d'un prototype réel et donc, éventuellement, de détecter et corriger des problèmes en amont des phases de développement. Le développement d'un prototype virtuel pour la simulation (c'est-à-dire une modélisation informatique) est par ailleurs rapide à effectuer par rapport à un prototype réel, du moins si le niveau d'abstraction est assez élevé. Le prototypage virtuel permet aussi de limiter l'utilisation et le nombre de prototypes réels réalisés qui coûtent très cher à réaliser et à déployer.

Problématique générale

Les spécificités et propriétés des SMPMPs concernant les accès à la mémoire relèvent de ce qu'on appelle la consistance mémoire. Comme ces propriétés sont différentes d'un SMPMP à l'autre, elles sont décrites dans ce qui est appelé un Modèle de Consistance Mémoire (MCM). Les mécanismes de traitement des accès à la mémoire sont un point essentiel des SMPMPs car les

2. Une *data race* correspond à un scénario dans lequel le résultat d'accès à la mémoire partagée est indéterminé car l'ordonnancement de ces accès n'est pas contraint. Quand le comportement du programme dépend de cet ordonnancement non prévisible, des problèmes graves, mais bien souvent indétectables par le programme, peuvent apparaître.

accès sont une source de dégradation des performances à cause de la lenteur des composants de mémorisation relativement aux calculs. Pour en augmenter les performances, les systèmes matériels font appel à des optimisations qui diminuent les contraintes pesant sur les accès [DSB88]. La gestion de ces contraintes est laissée au logiciel, on dit que la consistance mémoire est alors « relâchée ».

Un MCM caractérise donc le comportement des accès à la mémoire ; il définit en particulier si les accès peuvent être réordonnés par rapport à l'ordre dans lequel ils apparaissent dans le programme ; le cas échéant, il donne les conditions pour pouvoir le faire. Grâce aux bibliothèques de synchronisation, les applications n'ont dans la majorité des cas pas à se préoccuper du MCM si elles respectent les règles d'utilisation de ces bibliothèques. De manière générale, plus les performances de communication entre les différentes parties d'un programme sont importantes, plus le MCM devra être pris en considération. Les bibliothèques de synchronisation et, *a fortiori*, le système d'exploitation doivent par contre y faire très attention.

Pour obtenir un fonctionnement correct de l'ensemble d'un SMPMP et de son logiciel, il est crucial que le logiciel soit développé par rapport au MCM fourni par le système matériel. La vérification des propriétés de consistance mémoire d'un SMPMP est donc très importante mais est de manière générale un problème très complexe. Nous étudions dans cette thèse la vérification de cette consistance lors de la simulation d'un système. Cela consiste à simuler le SMPMP et le logiciel qui s'y exécute et à vérifier que la consistance mémoire est correcte durant la simulation. Nous présentons un algorithme, passant à l'échelle, pour effectuer cette vérification de manière dynamique, c'est à dire en même temps que le SMPMP exécutant le programme est simulé.

Cette vérification nécessite la connaissance de nombreuses informations. L'extraction de ces informations est complexe et doit faire l'objet d'une attention particulière afin de ne pas trop impacter la vitesse de la simulation. Dans cette thèse, nous définissons un environnement générique permettant d'extraire des informations et de faciliter l'analyse de ces informations.

Organisation de la thèse

Ce manuscrit est structuré de la manière suivante :

Le chapitre 1 présente la problématique de cette thèse. Nous y abordons le problème de la consistance mémoire et, en particulier de sa vérification. Le problème étant très complexe dans sa généralité, nous nous situons dans le contexte de la simulation.

L'état de l'art est décrit dans le chapitre 2. Nous y discutons les techniques utilisées pour réaliser la vérification de la consistance mémoire, ainsi que les approches visant à simplifier le problème.

Le chapitre 3 est la première contribution de cette thèse. Il est centré sur la vérification dynamique de la consistance mémoire, c'est-à-dire en parallèle à l'exécution du programme. Une représentation ainsi qu'une stratégie adaptée sont présentées dans ce chapitre.

Le chapitre 4 est la deuxième contribution de cette thèse. Il détaille un algorithme pour la vérification dynamique de la consistance mémoire et s'appuie sur des exemples d'implantation. Nous nous attardons en particulier sur le problème de l'occupation mémoire, condition importante pour le passage à l'échelle.

Le chapitre 5 est une contribution plus anecdotique, mais cependant nécessaire pour mener à bien la vérification dynamique de la consistance mémoire. Il définit un environnement d'analyse ciblant la simulation. Les enjeux et objectifs de cet environnement sont décrits avant d'entrer dans les détails de celui-ci.

Le chapitre 6 est dédié à la description des implantations et expérimentations effectuées. L'implantation de l'environnement d'analyse et son utilisation dans un simulateur de SMPMPs sont détaillées. Des résultats concernant l'environnement et l'algorithme de vérification dynamique de la consistance mémoire sont donnés.

Dans le dernier chapitre, nous résumons nos contributions et donnons des perspectives et des pistes d'amélioration des travaux présentés.

Quatre annexes sont ensuite incluses. L'annexe A donne une démonstration du passage à l'échelle de la méthode de détection de cycle dans un graphe utilisé pour la vérification de la consistance mémoire. L'annexe B donne des précisions sur certaines parties de l'algorithme qui ne sont pas incluses dans le chapitre 4. L'annexe C décrit les événements gérés par l'environnement de trace que nous avons implanté. L'annexe D donne les résultats pour les expérimentations qui ne sont pas détaillées dans le chapitre 6.

1 | Problématique

Dans ce chapitre, nous abordons les problèmes liés à la Vérification de la Consistance Mémoire (VCM) dans le contexte du prototypage virtuel. Ainsi que nous l'avons vu dans l'introduction, la consistance mémoire concerne les propriétés des accès à la mémoire dans les SMPMPs. Nous introduisons ici, de manière plus précise, le concept de la consistance mémoire avant de détailler les différents points étudiés dans cette thèse.

Chapitre 1

1.1 Contexte de la thèse 6
- 1.1.1 Systèmes multi-processeurs à mémoire partagée 6
- 1.1.2 Simulation de systèmes multi-processeurs 7

1.2 Consistance mémoire 8
- 1.2.1 Notations utilisées 9
- 1.2.2 Consistance séquentielle 9
- 1.2.3 Optimisations et conséquences 10
- 1.2.4 Modèles de consistance mémoire 12
- 1.2.5 Conclusion sur la consistance mémoire 19

1.3 Vérification de la consistance mémoire 19
- 1.3.1 Principe de la vérification 19
- 1.3.2 Problèmes liés à la vérification 21
- 1.3.3 Conclusion sur la vérification de la consistance mémoire 22

1.4 Extraction des informations depuis la simulation 22
- 1.4.1 Besoin d'informations « transversales » 22
- 1.4.2 Impact sur la simulation 22
- 1.4.3 Conclusion 23

1.5 Conclusion de la problématique 23

1.1 Contexte de la thèse

Les travaux de cette thèse se situent dans le contexte de la simulation des SMPMPs, et en particulier celle des Systèmes Multi-Processeurs sur Puces (MPSoCs). Avant d'aborder les problèmes étudiés il convient donc de présenter ces systèmes ainsi que les différentes techniques utilisées pour leur simulation.

1.1.1 Systèmes multi-processeurs à mémoire partagée

1.1.1.1 Architecture matérielle

Les SMPMPs sont des systèmes qui intègrent de nombreux composants, en particulier des processeurs et des périphériques de mémorisation. Ces composants sont connectés entre eux par un système d'interconnexion qui permet principalement à chaque processeur de communiquer avec les périphériques ou les autres processeurs.

La FIGURE 1.1 montre l'architecture TilePro64 [Tile64] d'une puce récente multi-processeurs de Tilera. Elle utilise un système d'interconnexion de type réseau sur puce (NoC) et se présente sous la forme d'un réseau en grille de 8×8 tuiles principales. Chacune de ces tuiles contient en particulier un processeur. D'autres éléments sont connectés à ce réseau et permettent d'interfacer celui-ci à divers périphériques.

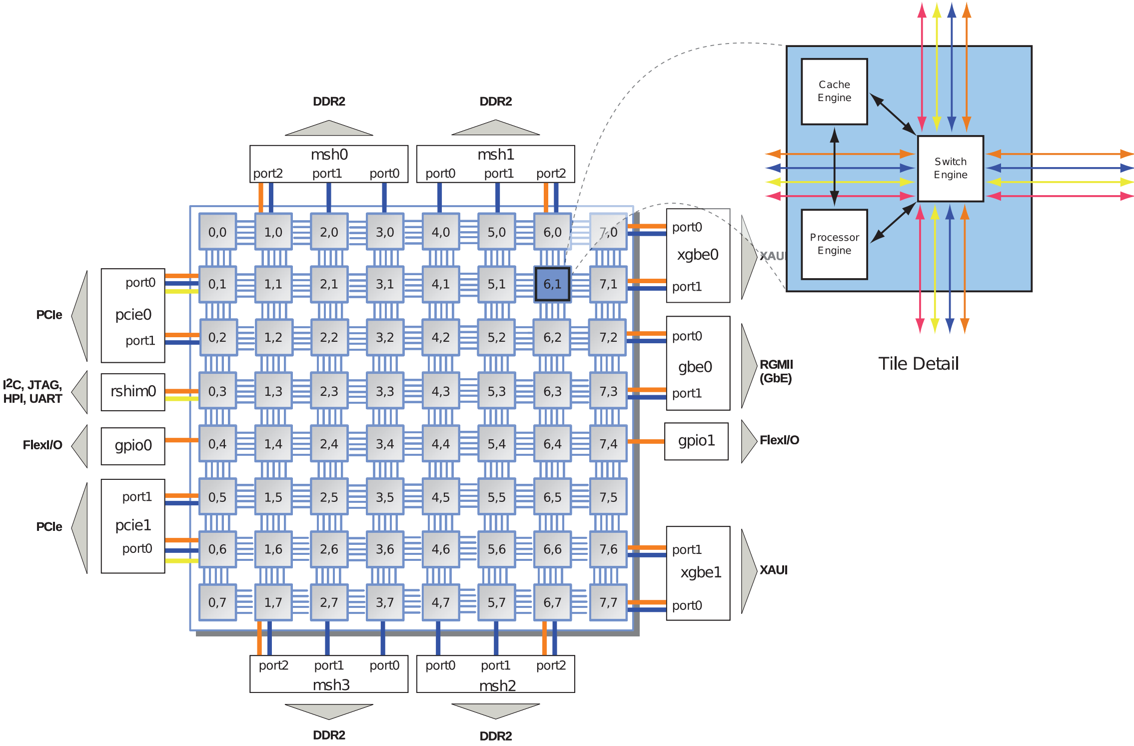


FIGURE 1.1 - Architecture TilePro64 (source : *Tile Processor User Architecture Manual* [Tile64])

Dans un SMPMP, les processeurs ont accès à une zone mémoire commune. Cette mémoire sert de moyen principal de communication entre les processeurs. En particulier ils peuvent y partager des données ou se synchroniser à travers celle-ci. Dans la FIGURE 1.1 tous les processeurs ont accès à de la mémoire SDRAM externe à la puce. La totalité de cette mémoire est partagée par l'ensemble des processeurs.

1.1.1.2 Pile logicielle

Les SMPMPs sont utilisés pour l'exécution de logiciels qui peuvent être très complexes. Dans nos travaux, nous considérons la plupart du temps le logiciel comme un tout. Néanmoins ce logiciel est classiquement divisé en deux couches : la couche du Système d'Exploitation (OS) et la couche applicative.

Cette représentation est illustrée dans la Figure 1.2 où est aussi représenté le support d'exécution matériel contenant les processeurs (*cpu* dans la figure). La couche applicative est constituée d'un programme parallèle divisé en plusieurs programmes séquentiels (*thread* dans la figure). La couche de l'OS contient le logiciel permettant l'exécution de la couche applicative sur le SMPMP. Dans ce document, le mot *thread* désigne un *thread* « applicatif ».

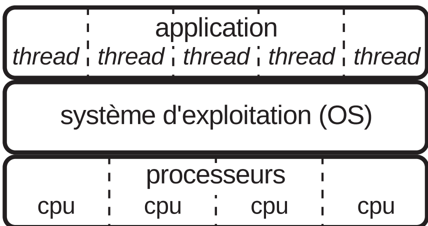


Figure 1.2 - Couches logicielles

1.1.2 Simulation de systèmes multi-processeurs

La simulation est beaucoup utilisée lors de la mise au point des systèmes multi-processeurs, en particulier pour les MPSoCs [Lud+02]. Elle permet d'effectuer différents tests sans nécessiter de système réel et donc d'éviter la conception détaillée et la fabrication d'un prototype qui est longue et coûteuse.

Pour éviter toute confusion, nous précisons d'abord certains termes utilisés dans le domaine de la simulation.

Cible (*target*) Le système cible est le système matériel à simuler.

Hôte (*host*) Le système hôte est le système utilisé pour réaliser la simulation.

Lors de la simulation d'un SMPMP, différentes techniques sont utilisées pour simuler l'exécution du logiciel sur le processeur cible. Les principales sont les suivantes.

Exécution native L'exécution native consiste à exécuter directement le logiciel sur le système hôte. Le logiciel, éventuellement adapté, est donc compilé pour le système hôte.

DBT La Traduction Binaire Dynamique (DBT) consiste à transformer à la volée le logiciel, compilé pour le système cible, en instructions à exécuter directement sur le système hôte.

ISS Un Simulateur de Jeu d'Instructions (ISS) est un programme qui permet de simuler un processeur en interprétant les instructions cibles. Le logiciel est donc compilé pour le système cible.

Différents niveaux d'abstraction existent pour modéliser les composants et le système cible en vue de la simulation. Ils ont chacun leurs avantages et leurs inconvénients. Un haut niveau d'abstraction offre en particulier une précision moindre mais permet une simulation plus rapide alors qu'un modèle très précis sera plus lent. Le niveau d'abstraction est un paramètre important. Il est choisi en fonction des objectifs recherchés. Au cours du développement d'un SMPMP plusieurs niveaux de simulation sont généralement utilisés. Les principaux niveaux de simulation communément acceptés sont décrits ci-dessous par ordre d'abstraction décroissant.

***System Level* (SL)** Le niveau SL est très abstrait. Le système cible est représenté sous la forme d'un ensemble de tâches qui communiquent entre elles grâce à des APIs spécifiques. Une tâche peut correspondre à un programme logiciel ou un composant matériel du système cible. L'architecture matérielle n'est donc pas réellement simulée et le système d'exploitation n'est pas représenté. Les tâches sont compilées pour le système hôte et sont une représentation purement fonctionnelle du système cible. La simulation de cet ensemble de tâches permet donc surtout de vérifier les fonctionnalités du système complet.

***Transaction Level Modeling* (TLM)** Dans une simulation au niveau TLM, les principaux éléments (processeurs, caches, mémoires, périphériques, ...) de l'architecture matérielle du système sont représentés. Les communications entre ceux-ci sont néanmoins abstraites sous forme de transactions. Le comportement des éléments simulés est représentatif du système cible. Néanmoins la précision de la simulation est fortement dépendante des choix d'implantations. En particulier plusieurs solutions peuvent être utilisées pour la simulation d'un processeur et du logiciel qu'il exécute : exécution native, DBT ou ISS.

***Cycle Accurate* (CA)** Dans la simulation CA les interfaces des composants matériels du système cible ne sont plus abstraites et sont formées des signaux réels. La valeur de ces signaux au cours de la simulation correspond à chaque cycle d'horloge aux valeurs réelles. La modélisation de ces composants doit donc refléter le comportement réel au regard de ses interfaces avec les autres composants. Les modèles des composants peuvent néanmoins abstraire le comportement interne de ceux-ci. Un processeur peut par exemple utiliser un ISS.

***Register Transfer Level* (RTL)** Le niveau RTL permet de simuler une description RTL [1] du système matériel. Celle-ci est donc extrêmement précise mais très lente. Aucune technique particulière n'est utilisée pour le logiciel cible : celui-ci est exécuté par la représentation RTL du processeur.

Dans cette thèse nous nous intéressons principalement aux problématiques des accès mémoire et à l'impact qu'ils ont sur le logiciel. Ainsi les problématiques étudiées s'appliquent aux simulations qui implantent les mécanismes fins de gestion des accès mémoire. Ainsi cela ne s'applique pas à des simulations très abstraites : en particulier les accès à la mémoire doivent être réellement joués et le logiciel ne doit pas être modifié. Le niveau SL et le niveau TLM abstrait ne sont donc pas suffisamment précis car le logiciel n'est pas exécuté par le système simulé. Le niveau RTL n'est pas très intéressant car sa vitesse de simulation est très lente. Nous nous plaçons donc dans le contexte de simulations TLM et CA qui permettent d'implanter ces mécanismes.

1.2 Consistance mémoire

Chaque SMPMP ayant ses particularités, il est très compliqué de développer des programmes dont le comportement est prévisible sans hypothèses extrêmement précises et explicites, en particulier vis-à-vis des accès à la mémoire. La notion de consistance mémoire permet d'éviter aux programmeurs de devoir se préoccuper de l'implantation des mécanismes matériels de gestion de la mémoire utilisés dans un SMPMP. La consistance mémoire permet en effet de modéliser entièrement le comportement du système de gestion de la mémoire. De cette manière il suffit de développer un programme pour le modèle de consistance implanté dans le SMPMP ciblé. Dans le contexte de la consistance mémoire, le système matériel est une boîte noire et seul le comportement exposé au programme exécuté compte. Les propriétés de consistance mémoire d'un SMPMP sont regroupés dans ce qu'on appelle un MCM.

1. Une description RTL décrit comment calculer le contenu de chaque registre du système matériel réel à chaque cycle d'horloge.

1.2.1 Notations utilisées

Afin d'éviter toute confusion il est nécessaire de préciser le sens de certains mots que nous utilisons dans la suite de ce document.

Définition 1 (Séquence d'instructions)**.** Une Séquence d'Instructions (SI) est une suite ordonnée d'instructions. Cette SI correspond à ce qui est généralement appelé un *thread* « matériel ». Nous utilisons le terme « SI » pour éviter d'introduire une ambiguïté avec les *threads* applicatifs.

Définition 2 (Accès)**.** Un accès à la mémoire, que nous appelons simplement accès, est une opération qui permet d'observer ou de modifier une partie de la mémoire identifiée par son adresse. Un accès est généralement une simple lecture ou écriture. Certaines architectures offrent néanmoins d'autres accès spéciaux, tels que des *swaps* [2] ou des accès conditionnés. Un accès, en particulier une écriture, est caractérisé par une certaine atomicité qui sera explicitée dans la suite de cette partie.

Définition 3 (Instructions)**.** Une instruction est un élément d'une SI. Il est important de noter que cette définition ne restreint pas une instruction à correspondre à une unique opération (accès à la mémoire, opération arithmétique, etc.). Une instruction peut correspondre à plusieurs opérations comme par exemple les instructions de type VLIW ou SIMD. Les différentes opérations ne sont par ailleurs pas forcément séquentielles au sein d'une instruction. Une instruction peut typiquement effectuer plusieurs accès.

Définition 4 (Programme)**.** Un programme est un ensemble de Séquences d'Instructions (SIs) qui sont exécutées en parallèle. L'exécution des instructions des SIs en suivant l'ordre des séquences permet de réaliser le comportement attendu du programme.

Définition 5 (Processeur)**.** Un processeur est un composant matériel qui exécute une Séquence d'Instructions (SI).

On peut noter que la manière dont est implanté un processeur n'entre pas en compte. Ainsi un système qui partage des éléments matériels pour exécuter deux SIs, en utilisant par exemple la technique du *Simultaneous MultiThreading* (SMT) [3], est considéré ici comme constitué de deux processeurs.

En résumé, un système multi-processeurs exécute un programme réparti en plusieurs parties séquentielles, les SIs, exécutées chacune par un processeur. Dans cette thèse, nous nous concentrons sur la consistance mémoire d'un système matériel, le programme englobe ainsi toutes les couches logicielles (OS et application). La consistance mémoire peut néanmoins être appliquée à différentes interfaces. Par exemple, on peut considérer qu'un OS fournit une certaine consistance mémoire aux applications exécutées virtuellement par autant de processeurs qu'il y a de *threads*.

1.2.2 Consistance séquentielle

Parmi les différents MCMs, la Consistance Séquentielle (SC) est considérée comme le modèle de consistance de référence. C'est le modèle qui est le plus intuitif pour les développeurs de programmes. Il découle en effet d'une application aux systèmes multi-processeurs du comportement attendu d'un programme sur un unique processeur. En ce qui concerne les accès à la mémoire, un programme s'exécute correctement sur un système mono-processeur si :

2. Un *swap* correspond à une lecture suivie immédiatement d'une écriture.

3. Pour mémoire, la technique du SMT consiste à faire exécuter simultanément plusieurs SIs par un même « processeur » grâce à une architecture *ad hoc*. Cela permet d'optimiser l'utilisation des ressources matérielles du processeur qui peuvent être sous-utilisées lors de l'exécution d'une unique SI.

1. les accès à la mémoire se font dans l'ordre spécifié par le programme,
2. une lecture lit la valeur écrite par la dernière (dans l'ordre du programme) écriture à la même case mémoire.

Ce comportement correspond uniquement à ce qui doit être observé du point de vue du programme. Les composants peuvent faire des écarts par rapport à une implantation stricte de ces points tant que ceux-ci ne sont pas observables par le programme. Il n'y a par exemple aucune conséquence à changer l'ordre de réalisation d'écritures à des zones mémoires différentes dans un système mono-processeur.

Dans un système mono-processeur, on considère que la mémoire est accédée de manière séquentielle par le processeur : la mémoire traite les accès les uns après les autres. La consistance séquentielle se base sur ce principe et l'étend à un système multi-processeurs. Elle a été définie par Leslie Lamport en 1979 de cette manière [Lam79] :
A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Cela signifie que le résultat d'une exécution dans un système multi-processeurs fournissant la consistance séquentielle doit correspondre à l'exécution d'une séquence formée de tous les accès du programme. Dans cette séquence, les accès issus d'une même SI doivent de plus apparaître dans l'ordre de cette SI. Il est très important de noter que ce qui compte est uniquement ce qui est vu par le programme. La manière dont sont réalisés les accès par le système matériel n'a pas d'importance.

1.2.3 Optimisations et conséquences

Le modèle SC est très intuitif et correspond à un système multi-processeurs simple. Un système basé sur un unique bus de communication, auquel la mémoire et les processeurs sont directement reliés, respecte par exemple le modèle SC. Des systèmes plus optimisés peuvent être néanmoins utilisés pour la SC, par exemple en utilisant des mécanismes d'exécution spéculative dans les processeurs.

La complexité actuelle des SMPMPs laisse envisager qu'un support de la SC soit envisageable et ne pénalise pas trop les performances [Hil98 ; Uba+12]. A l'heure actuelle, les systèmes développés ne sont néanmoins pas compatibles avec le modèle SC. De nombreux mécanismes ou optimisations sont en effet utilisés par les systèmes actuels pour en améliorer les performances et ces mécanismes violent les contraintes de la SC. Les principaux mécanismes sont décrits dans cette section ainsi que leurs conséquences sur la mémoire.

1.2.3.1 Exécution désordonnée

Les processeurs utilisent des pipelines d'exécution qui exécutent les instructions en plusieurs étapes. Cette technique permet d'augmenter leur fréquence de fonctionnement. Mais lorsque deux instructions de la SI sont dépendantes (par exemple si une instruction utilise une donnée calculée par une instruction précédente), il peut être nécessaire de retarder la deuxième instruction. Il en résulte une utilisation non optimale du pipeline car des « bulles » (ne réalisant aucune opération) sont dites insérées dans le pipeline en attendant de pouvoir exécuter la prochaine instruction.

Afin d'optimiser l'utilisation du pipeline certains processeurs permettent l'exécution des instructions dans le désordre. Ainsi lorsque une instruction est bloquée à cause d'une dépendance, les instructions qui la suivent sont susceptibles d'être exécutées avant elle. Ces changements d'ordre sont bien sûr effectués uniquement si les instructions suivantes ne sont pas dépendantes des précé-

dentes afin de garantir que l'exécution de la SI se comporte comme si elle avait été exécutée dans l'ordre. Les dépendances classiques sont *Read after Write* (RAW), *Write after Read* (WAR), *Write after Write* (WAW) et éventuellement *Read after Read* (RAR).

La notion de dépendance n'est néanmoins évaluée et évaluable que dans le contexte local de chaque SI. Cela n'a pas d'impact dans le cadre d'opérations arithmétiques, mais cela peut en avoir dans le cadre d'accès à la mémoire car ceux-ci sont sensibles à des événements extérieurs au processeur et inversement. Afin de ne pas avoir d'impact, certaines architectures interdisent simplement tout réordonnancement d'accès à la mémoire. La majorité des architectures permettent aujourd'hui certains réordonnancements (par exemple entre une écriture suivie d'une lecture).

Par exemple les processeurs de l'architecture TilePro64 (cf. la FIGURE 1.1) permettent de réordonner les lectures et les écritures. Les dépendances de données RAW, WAR et WAW sont néanmoins conservées : deux accès au même emplacement mémoire, dont au moins un est une écriture, ne sont pas réordonnés.

1.2.3.2 Tampons d'écriture

Un tampon d'écriture (*write buffer*) permet de stocker temporairement des écritures en attendant qu'elles soient réellement effectuées. Lors d'une écriture, au contraire d'une lecture, un processeur n'attend pas de données. Le tampon d'écriture permet donc au processeur de ne pas se bloquer en attendant que les écritures soient terminées, c'est à dire qu'elles atteignent effectivement la mémoire.

Dans certains cas, le processeur peut même effectuer d'autres accès, y compris des lectures, à la mémoire sans attendre que le tampon d'écriture soit vidé. Certains tampons d'écriture permettent aussi de combiner plusieurs accès lorsque ceux-ci accèdent à des zones mémoires adjacentes. Cela permet de réduire le nombre de requêtes d'accès en écriture et de réduire l'utilisation du système de communication.

1.2.3.3 Mémoire cache

Une mémoire cache, ou cache, est une zone de mémoire qui est insérée entre un processeur et la mémoire de manière qu'il soit plus rapide d'accéder au cache qu'à la mémoire. Cette mémoire est petite et utilise généralement une technologie différente de la mémoire principale pour pouvoir être plus rapide. Le principe consiste à y stocker des copies de certaines parties de la mémoire pour pouvoir réduire le temps d'accès aux données qu'elles contiennent.

En pratique plusieurs caches peuvent être utilisés de manière hiérarchique et les caches peuvent être différents pour les accès aux données ou aux instructions d'un même processeur. Ils sont alors différenciés par niveaux hiérarchiques : les niveaux inférieurs étant plus rapides d'accès mais de plus petite taille que les niveaux supérieurs.

Niveau 1	Niveau 2	Mémoire
2 cycles	8 cycles	80 cycles

FIGURE 1.3 - Temps d'accès en fonction du niveau de cache pour l'architecture TilePro64 [Tile64]

Par exemple dans l'architecture TilePro64 (cf. la FIGURE 1.1), chaque tuile processeur contient 2 caches de niveau 1. Le cache des instructions a une capacité de 16 Kio tandis que celui des données a une capacité de 8 Kio. Chaque tuile contient aussi un cache de niveau 2 de 64 Kio. La FIGURE 1.3 montre les temps d'accès à ces caches et à la mémoire pour les données. Le temps d'accès à la mémoire est d'un ordre de grandeur plus élevé que les temps d'accès aux caches.

Dans un système où il y a plusieurs caches, en particulier dans les systèmes multi-processeurs, il est primordial, pour assurer une exécution correcte du logiciel, que les caches soient cohérents. Cela signifie en particulier que lorsqu'une case mémoire est modifiée par un processeur, cette modification est propagée à tous les caches qui en détiennent une copie. En l'absence de mécanisme automatique, appelé protocole de cohérence de cache, cette tâche doit être prise en charge par le logiciel lui-même. Dans ce cas des règles très strictes doivent être suivies pour la programmation parallèle. Il y a par ailleurs un impact non négligeable puisque cela nécessite souvent de devoir invalider régulièrement l'ensemble des caches. C'est pourquoi dans la plupart des systèmes multi-processeurs destinés à exécuter un logiciel utilisant le principe de la mémoire partagée, il y a un mécanisme matériel qui assure la cohérence. Ce mécanisme implique des communications spéciales entre les caches.

A cause de ces mécanismes d'optimisation, la consistance mémoire implantée dans les systèmes multi-processeurs n'est pas la SC.

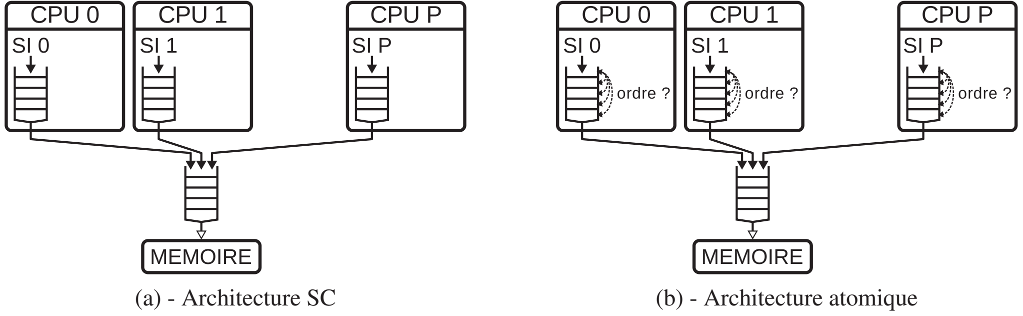
1.2.4 Modèles de consistance mémoire

De nombreux MCM existent donc. Ceux-ci peuvent néanmoins être comparés par leurs propriétés. Comme Arvind et Maessen [AM06] l'ont fait, un modèle de consistance peut être caractérisé par deux grandes métriques : le réordonnancement et l'atomicité des écritures.

Le réordonnancement définit simplement quels sont les instructions et les accès dont l'ordre peut être interverti dans une SI lors de son exécution. Il prend en compte les différentes dépendances entre les instructions et caractérise le pipeline d'un processeur.

L'atomicité est par contre une notion qui n'est pas locale aux processeurs. Dans cette partie nous classons les modèles en trois groupes en fonction de l'atomicité : les modèles atomiques, quasiment atomiques et non atomiques. Afin d'alléger le texte, pour parler d'une architecture implantant un certain MCM (SC par exemple), nous utilisons simplement l'expression : architecture SC.

1.2.4.1 Modèles atomiques



(a) - Architecture SC (b) - Architecture atomique

Figure 1.4 - Représentation schématique d'architectures SC et atomique

Une écriture est dite atomique si elle est vue au même moment par tous les processeurs du système. Comme la notion de temps n'est pas importante, « au même moment » ne s'évalue ici que par rapport aux autres écritures. Ainsi l'atomicité des écritures revient à imposer que tous les processeurs voient toutes les écritures dans un ordre commun. Dans les modèles atomiques il existe un ordre entre tous les accès à la mémoire, en particulier les écritures. Et cet ordre vérifie le principe de la mémoire : une lecture lit la dernière valeur écrite à la même zone mémoire.

Un processeur, et la SI qu'il exécute, ne peuvent bien sûr « voir » une écriture que s'ils lisent la partie de la mémoire qui a été écrite. En pratique, chaque processeur ne voit pas toutes les écritures car il ne fait pas les lectures adéquates. La vision partielle qu'il obtient doit être cohérente avec celles des autres processeurs.

Le modèle SC est atomique mais il n'est pas le seul. L'unique degré de liberté des modèles atomiques est le réordonnancement au sein des processeurs. L'ordre commun des accès n'est établi qu'après les réordonnancements autorisés et ne respecte donc pas obligatoirement l'ordre strict des SIs.

Par exemple dans l'architecture zSeries [Z], qui est atomique, il est possible que les lectures soient effectuées avant que les écritures précédentes soient terminées. Ce réordonnancement peut être, par exemple, implanté via l'autorisation aux lectures de doubler les écritures dans un tampon d'écriture. Néanmoins cela n'est possible que si les cases mémoire accédées sont différentes : une lecture ne peut logiquement pas doubler une écriture à la même case mémoire.

La Figure 1.4 représente schématiquement deux architectures : l'une implante la SC, l'autre est seulement atomique. La schématisation est similaire à celle utilisée par Gharachorloo [Gha95]. La différence entre les deux schémas se situe dans l'autorisation de réordonnancement symbolisé au niveau du processeur par un tampon de réordonnancement dans la Figure 1.4b. Dans les deux cas, chaque accès à la mémoire est réalisé par la mémoire et passe à travers un système de communication équivalent à une file de type *First In First Out* (FIFO). Seuls les canaux de communications correspondant aux requêtes vers la mémoire sont indiqués.

1.2.4.2 Impact du réordonnancement

Afin d'illustrer les différents MCMs et leurs impacts sur les programmes, nous utilisons quelques programmes à titre d'exemples. Ils mettent en œuvre des scénarios simples d'accès à la mémoire. Chacun est formé de plusieurs SIs, séparées dans des colonnes, qui s'exécutent en parallèle sur différents processeurs.

La syntaxe utilisée dans les programmes est la suivante. Les lettres x et y représentent les adresses de deux variables différentes stockées en mémoire. Au début d'un programme, toutes les variables sont considérées comme initialisées à la valeur 0. Les registres d'un processeur sont représentés par $r_1, r_2, \ldots$ Pour plus de clarté des numéros différents de registres sont utilisés dans les différentes SI d'un même programme. Les registres ne sont néanmoins pas partagés par les processeurs : chaque processeur utilise un jeu de registre qui lui est propre.

Chaque ligne d'une SI représente une instruction. $r_1 := [x]$ représente par exemple la lecture de la variable x, la valeur lue est stockée dans le registre r_1. $[y] := 1$ représente l'écriture de la valeur 1 dans la variable y. Les instructions d'accès en lecture et en écriture sont identifiées respectivement par $R_{n.i}$ et $W_{n.i}$ au sein d'un programme. n y représente le numéro de la SI et i l'indice de l'opération dans la SI.

En plus de ces instructions, nous utilisons aussi une instruction spéciale. L'instruction *no_reorder* est utilisée pour signifier que deux accès de part et d'autre de *no_reorder* ne peuvent pas être réordonnés. Cette instruction n'empêche que les réordonnancements locaux aux processeurs.

Le programme A du Tableau 1.1 consiste en deux SIs qui écrivent la valeur 1 dans une variable et lisent ensuite la variable écrite par l'autre SI. Considérons une exécution du programme par une architecture atomique, une des deux écritures sera effectuée avant l'autre. On s'intéresse aux résultats possibles contenus dans les registres r_1 et r_2. Comme les deux SIs sont symétriques, on peut supposer, sans perte de généralité, que c'est $W_{1.1}$ qui s'est produite en premier. On utilise dans la suite la notation « $A < B$ » pour signifier qu'un accès A se produit avant un autre B. Dans le modèle SC, qui ne permet aucun réordonnancement, on obtient les contraintes suivantes :

Programme A. Initialement $([x],[y]) = (0,0)$.

SI A_1	SI A_2
W1.1 : [x] := 1	W2.1 : [y] := 1
R1.2 : r1 := [y]	R2.2 : r2 := [x]

Résultat : $(r1, r2) = (0,0)$?

TABLEAU 1.1 - Programme sensible au réordonnancement d'une écriture suivie d'une lecture

1. l'ordre de la SI C_1 : $W_{1.1} < R_{1.2}$;
2. l'ordre de la SI C_2 : $W_{2.1} < R_{2.2}$; et
3. l'hypothèse que $W_{1.1}$ est avant $W_{1.2}$: $W_{1.1} < W_{2.1}$.

Si on essaye de séquencer ces quatre accès en respectant ces contraintes, il n'y a que trois séquences possibles :

– « $W_{1.1}, R_{1.2}, W_{2.1}, R_{2.2}$ » qui donne le résultat $(r_1, r_2) = (0, 1)$;
– « $W_{1.1}, W_{2.1}, R_{1.2}, R_{2.2}$ » qui donne le résultat $(r_1, r_2) = (1, 1)$; et
– « $W_{1.1}, W_{2.1}, R_{2.2}, R_{1.2}$ » qui donne aussi le résultat $(r_1, r_2) = (1, 1)$.

Ainsi, si on enlève l'hypothèse de symétrie, les résultats possibles pour (r_1, r_2) sont $(0, 1)$, $(1, 1)$ et par symétrie $(1, 0)$. $(0, 0)$ n'est donc pas un résultat possible. C'est en utilisant ce principe que les algorithmes d'exclusion mutuelle (tel l'algorithme de Decker) sans accès spéciaux fonctionnent : la SI qui lit le 0 peut entrer dans la section critique.

Par contre, si l'architecture autorise les lectures à doubler les écritures précédentes, ce qui est très souvent le cas, alors il n'y a plus de contrainte entre la lecture et l'écriture d'une même SI. Ainsi la séquence « $R_{1.2}, R_{2.2}, W_{1.1}, W_{2.1}$ » est par exemple possible dans l'architecture zSeries. Comme les lectures sont exécutées avant les écritures, le résultat est $(0, 0)$. Dans le cadre d'un algorithme d'exclusion mutuelle, les deux SIs sont donc susceptibles de rentrer dans la section critique.

1.2.4.3 Modèles avec écritures anticipées

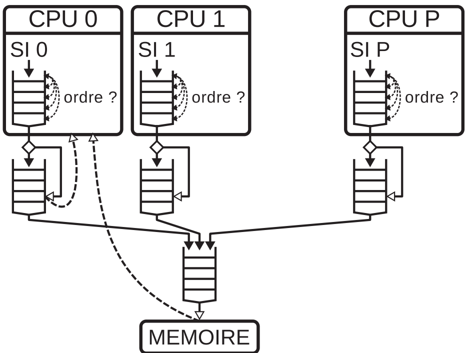


FIGURE 1.5 - Représentation schématique d'une architecture autorisant les écritures anticipées

Ce que nous appelons dans cette thèse les modèles avec écritures anticipées sont les modèles où un processeur peut lire le résultat de ses propres écritures avant qu'elles ne soient réellement effectuées et donc propagées à l'ensemble du système. Une écriture lue de cette manière n'est alors plus atomique au sens donné précédemment. Elle peut être vue, par le processeur qui l'a émise, avant les autres processeurs.

La FIGURE 1.5 montre une représentation d'une architecture permettant la lecture anticipée des écritures. Par rapport à l'architecture atomique, un tampon a été ajouté après le processeur et celui-ci peut servir à fournir le résultat d'une lecture si une écriture à la même case mémoire y est présente. Les flèches pointillées indiquent les deux possibilités de chemin de réponse dans le cas du *CPU 0*. Dans le cas où le tampon sert une lecture, la lecture n'atteint jamais la mémoire.

La différence principale avec les architectures atomiques est généralement dans la gestion des tampons et des possibilités de « court-circuiter » ceux-ci. Il est par exemple très courant de pouvoir lire des données en cache même si des écritures sont encore en attente dans un tampon d'écriture. Cela peut être vu comme un réordonnancement dans la plupart des cas, mais pas lorsqu'une précédente lecture a déjà utilisé le résultat d'une écriture du tampon d'écriture. Si l'écriture lue n'est pas encore effectuée, alors lire des données directement dans le cache peut mener à des cas de non atomicité. Au moment où l'écriture est réellement effectuée, la ligne lue du cache a en effet pu être invalidée et la donnée lue aurait été peut-être différente si le processeur avait attendu.

Ainsi dans une architecture atomique type zSeries, un processeur peut lire une donnée qui est en cours d'écriture dans le tampon d'écriture. Mais les écritures suivantes ne pourront plus doubler l'écriture dont la donnée a été lue puisqu'elles doubleraient alors la lecture, ce qui est interdit. En autorisant les écritures anticipées, cela ne pose pas de problème puisque les écritures sont visibles localement au processeur avant de l'être globalement.

Dans ces architectures, les écritures sont quasiment atomiques. Un ordre global des accès, et en particulier des écritures, existe encore. Celui-ci correspond à l'ordre dans lequel sont faites les écritures au niveau de la mémoire. Par contre chaque processeur peut voir un ordre légèrement différent de celui-ci car il peut observer ses écritures plus tôt.

Les architectures de ce type sont très courantes, la plus connue est l'architecture *IA-32* [IA-32] (ou *AMD64* [AMD64]) qui équipe la plupart des ordinateurs personnels. Celle-ci permet le réordonnancement des lectures qui suivent les écritures et dans certains cas les écritures avec les écritures. L'architecture *SPARC* [SPARC] permet aussi les écritures anticipées. Cette dernière implémente plusieurs MCMs qui autorisent différents réordonnancements au sein d'un processeur. Le modèle utilisé peut être changé dynamiquement parmi TSO, PSO et RMO. Le modèle *Total Store Order* (TSO) permet uniquement de réordonner les écritures suivies de lectures. Le modèle *Partial Store Order* (PSO) permet, en plus, de changer l'ordre entre des écritures. Le modèle *Released Memory Order* (RMO) permet de changer l'ordre entre n'importe quels accès. L'architecture *TilePro64* utilise un modèle similaire au modèle RMO.

1.2.4.4 Impact des écritures anticipées

Programme B. Initialement $([x], [y]) = (0, 0)$.

SI B_1	SI B_2	SI B_3
$W_{1.1}(1)$: [x] := 1	$W_{2.1}(1)$: [y] := 1 $R_{2.2}(1)$: r1 := [y] $R_{2.3}(0)$: r2 := [x]	$R_{3.1}(1)$: r3 := [x] $R_{3.2}(?)$: r4 := [y]

Hypothèse : $(r_1, r_2, r_3) = (1, 0, 1)$.
Résultat : $r4 = ?$

TABLEAU 1.2 - Programme sensible aux écritures anticipées

Le programme B du TABLEAU 1.2 est un programme utilisant trois SIs. La première écrit simplement dans la variable x. La deuxième écrit dans la variable y puis lit y dans r_1 puis x dans r_2. La troisième lit x dans r_3 puis y dans r_4. On suppose pour cette illustration que les valeurs lues des registres (r_1, r_2, r_3) sont $(1, 0, 1)$. Les valeurs écrites et lues sont mises entre parenthèses dans les SIs. Il peut paraître artificiel et inutile que la SI B_2 écrive y ($W_{2.1}$) puis le lise juste après ($R_{2.2}$). C'est le

cas car le programme est minimal et ne représente qu'un scénario, il est en pratique parfaitement possible d'inclure d'autres accès entre $W_{2.1}$ et $R_{2.2}$. De même seuls les accès sont représentés, en manipulant des adresses mémoire, il est parfaitement possible d'ignorer que c'est la même case mémoire qui est accédée.

Comme $R_{2.3}$ et $R_{3.1}$ lisent respectivement les valeurs 0 et 1 pour x, ils sont nécessairement ordonnés par rapport à $W_{1.1}$ qui écrit 1 dans x :

$$R_{2.3} < W_{1.1} < R_{3.1}.$$

Pour ce programme nous supposons que seul un réordonnancement des lectures suivant les écritures est possible. On obtient donc les contraintes suivantes pour l'ordre des SIs B_2 et B_3 :

$$R_{2.2} < R_{2.3} \text{ et } R_{3.1} < R_{3.2}.$$

Il y a aussi une dépendance de donnée entre $W_{2.1}$ et $R_{2.2}$ qui accèdent la même variable y. En supposant que l'architecture soit atomique (par exemple de type *zSeries*), cela ajoute la contrainte :

$$W_{2.1} < R_{2.2}.$$

Dans ce cas la seule séquence possible des 6 événements est « $W_{2.1}, R_{2.2}, R_{2.3}, W_{1.1}, R_{3.1}, R_{3.2}$ ». Celle-ci donne le résultat $r_4 = 1$ car $R_{3.2}$ est après $W_{2.1}$.

Par contre, si on considère une architecture autorisant les écritures anticipées (par exemple *x86*), alors la dépendance de donnée entre $W_{2.1}$ et $R_{2.2}$ n'implique pas $W_{2.1} < R_{2.2}$ car $R_{2.2}$ peut être effectuée avant $W_{2.1}$ par un tampon intermédiaire. Le résultat de $R_{2.2}$ sera néanmoins toujours 1. Dans ce cas $W_{2.1}$ n'est plus contraint, la séquence précédente est toujours possible mais elle n'est pas la seule. La suivante, dans laquelle $W_{2.1}$ passe de la première à la dernière position, est aussi possible : « $R_{2.2}, R_{2.3}, W_{1.1}, R_{3.1}, R_{3.2}, W_{2.1}$ ». Dans celle-ci, le résultat est $r_4 = 0$. B_2 et B_3 n'y voient donc pas les deux écritures dans le même sens. B_2 a en effet l'impression que l'écriture de y a lieu avant celle de x tandis que B_3 a l'impression inverse.

1.2.4.5 Modèles non atomiques cohérents

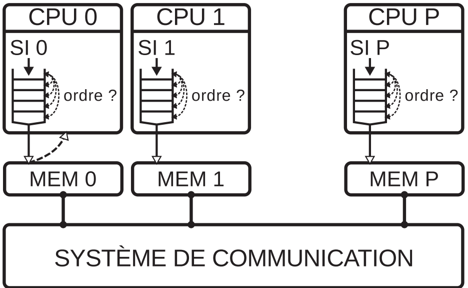


FIGURE 1.6 - Représentation schématique d'une architecture non atomique

Les modèles non atomiques représentent le cas général pour les systèmes à mémoires partagées. Dans un tel modèle, une écriture peut être vue à des moments différents : c.-à-d. les écritures peuvent être entrelacées les unes avec les autres. Dans cette thèse nous nous limitons néanmoins aux architectures dites cohérentes. La cohérence, appelée aussi *cache consistency*, correspond à un MCM minimal qui impose uniquement que les écritures à une même case mémoire soient vues dans le même ordre par tous les processeurs.

Dans les architectures non atomiques, le fait que chaque processeur accède à sa propre copie, au moins partielle, de la mémoire est exposé au logiciel. Ceci est représenté dans la FIGURE 1.6 qui représente une architecture non atomique. Dans celle-ci, un processeur possède sa propre version de la mémoire qui permet de répondre aux lectures. Lorsqu'un processeur fait une écriture, sa mémoire se charge de propager les modifications aux autres versions. Les différentes copies (ou versions) de la mémoire communiquent grâce à un système de communication qui permet à une version de communiquer avec une autre.

Les architectures *ARMv7* [ARMv7] et *Power* [POWER] sont non atomiques mais cohérentes. Elles autorisent en particulier le réordonnancement entre n'importe quels accès. L'architecture *Itanium* [Itanium] est, dans une certaine mesure, aussi non atomique.

1.2.4.6 Impact des écritures non atomiques

Programme C. Initialement $[x] = 0$.

SI C_1	SI C_2	SI C_3	SI C_4
$W_{1.1}$: [x] := 1	$W_{2.1}$: [y] := 1	$R_{3.1}$: r_1 := [x] no_reorder $R_{3.2}$: r_2 := [y]	$R_{4.1}$: r_3 := [y] no_reorder $R_{4.2}$: r_4 := [x]

Tout résultat est possible.

TABLEAU 1.3 - Programme d'illustration des accès non atomiques

Le cas des écritures non atomiques complique énormément les programmes. Avec seulement des écritures anticipées, les effets de bords d'une écriture sont limités à la SI qui l'émet. Si les écritures ne sont pas atomiques, aucune conclusion sur l'ordre des accès ne peut être tiré de l'observation d'une SI. Dans le programme C du TABLEAU 1.3, le résultat $(r1, r2, r3, r4) = (1, 0, 1, 0)$ est possible malgré le fait que ni C_3 ni C_4 ne font d'écritures : ils peuvent voir les écritures de manière inversée.

Aucun ordre global des accès ne peut être déduit, il est possible de raisonner uniquement sur chaque SI séparément. Avec une quatrième SI, les résultats des lectures des deux SIs seraient complètement indépendants. Une solution pour gérer une écriture atomique est par exemple de la représenter par un ensemble de sous-écritures atomiques impactant chacune la vision d'un processeur. L'écriture $W_{1.1}$ du programme C serait ainsi représentée dans une séquence par $W^1_{1.1}$, $W^2_{1.1}$, $W^3_{1.1}$ et $W^4_{1.1}$ pour différencier les SIs C_1, C_2, C_3 et C_4.

1.2.4.7 Impact de la cohérence mémoire

La cohérence mémoire imposée par les protocoles de cohérence de cache a des conséquences sur les programmes que nous illustrons par deux programmes présentés dans les TABLEAUX 1.4 et 1.5.

Programme D. Initialement $[x] = 0$.

SI D_1	SI D_2	SI D_3	SI D_4
$W_{1.1}$: [x] := 1	$W_{2.1}$: [x] := 2	$R_{3.1}$: r_1 := [x] no_reorder $R_{3.2}$: r_2 := [x]	$R_{4.1}$: r_3 := [x] no_reorder $R_{4.2}$: r_4 := [x]

Le résultat $(r_1, r_2, r_3, r_4) = (1, 2, 2, 1)$ est impossible.

TABLEAU 1.4 - Programme d'illustration des contraintes d'ordre de la cohérence

Le programme D du Tableau 1.4 est composé de quatre SIs. Il est très similaire au programme précédent. Les deux premières SIs écrivent une valeur différente dans la variable x tandis que les deux dernières lisent chacune deux fois x. Grâce à la cohérence mémoire, tous les processeurs voient nécessairement les écritures à x dans le même ordre. Cela empêche par exemple que $(r_1, r_2) = (1, 2)$ et $(r_3, r_4) = (2, 1)$ à la fin du programme. Si $(r_1, r_2) = (1, 2)$ cela signifie que l'écriture $W_{1.1}$ précède l'écriture $W_{2.1}$ et le résultat de la SI D_4 doit y être conforme : le résultat ne peut être $(r_3, r_4) = (2, 1)$

Programme B. Initialement $[x] = 0$.

SI B_1	SI B_2
$W_{1.1}$: `[x] = 1`	$R_{2.1}$: `do {` `r1 = [x]` `} while (r1 ≠ 1)`

La SI B_2 ne peut pas boucler indéfiniment.

Tableau 1.5 - Programme d'illustration de la propagation des écritures dans un système cohérent

Le programme du Tableau 1.5 illustre le fait que dans un système à mémoire cohérente les écritures sont garanties d'être propagées à toute les copies du système. Ainsi un programme qui boucle sur la valeur d'une variable finira par voir les modifications faites par d'autres processeurs. Cela a, en particulier, pour conséquence que les tampons intermédiaires ne peuvent pas retenir indéfiniment les valeurs ou accès.

1.2.4.8 Résumé des modèles

Le Tableau 1.6 présente les caractéristiques des modèles de consistance de quelques architectures multi-processeurs. Il contient aussi les caractéristiques du modèle de SC. Les deux métriques (le maintien de l'ordre de la SI et l'atomicité des écritures) sont représentées. Pour les réordonnancements autorisés, ils sont divisés en trois catégories : $W \rightarrow R$, $W \rightarrow W$ et $R \rightarrow \{R, W\}$. R représente les lectures et W les écritures. $W \rightarrow R$ représente l'ordre entre une écriture suivie d'une lecture. Un « oui » indique que l'ordre est maintenu.

Dans la plupart des cas, et en particulier dans les architectures du Tableau 1.6, le fait que les écritures soient globalement non atomiques inclut le cas particulier des écritures anticipées.

<table>
<tr><th colspan="2"></th><th>SC</th><th>zSeries</th><th>x86/IA-32 AMD64</th><th>SPARC TSO</th><th>SPARC PSO</th><th>SPARC RMO</th><th>Tile64Pro</th><th>ARMv7</th><th>Power</th><th>IA-64/Itanium</th></tr>
<tr><td rowspan="3">Ordre</td><td>$W \rightarrow R$</td><td>Oui</td><td>Non</td><td colspan="5">Non</td><td colspan="3">Non</td></tr>
<tr><td>$W \rightarrow W$</td><td colspan="2">Oui</td><td>O/N</td><td>Oui</td><td colspan="3">Non</td><td colspan="3">Non</td></tr>
<tr><td>$R \rightarrow \{R, W\}$</td><td colspan="2">Oui</td><td colspan="3">Oui</td><td colspan="2">Non</td><td colspan="3">Non</td></tr>
<tr><td colspan="2">Écritures</td><td colspan="2">Atomiques</td><td colspan="5">Anticipée</td><td colspan="3">Non atomiques</td></tr>
</table>

Tableau 1.6 - Propriétés de différentes architectures cohérentes

Certaines architectures se ressemblent mais sont en réalité toutes différentes. Les propriétés présentées dans cette partie sont générales. En particulier les jeux d'instructions des architectures ne se limitent pas aux lectures et aux écritures simples. Le réordonnancement éventuel de deux

accès n'est de toute façon possible que si les deux accès en cause sont indépendants. Cette notion de dépendance est typiquement extrêmement spécifique aux architectures. Elle peut s'évaluer en fonction de paramètres complexes : les opérations conditionnelles, les registres, les opérandes de type donnée, les opérandes de type adresse, etc. .

1.2.5 Conclusion sur la consistance mémoire : besoin de vérification

Dans cette partie nous avons présenté les différentes caractéristiques des modèles de consistance mémoire ainsi que les conséquences de ceux-ci sur l'exécution des programmes. Les applications classiques sont toutefois généralement assez insensibles aux variations des modèles dans la mesure où celles-ci utilisent des données partagées uniquement par l'intermédiaire de bibliothèques spécialisées.

Une erreur dans l'implantation du Modèle de Consistance Mémoire (MCM) peut néanmoins poser de gros problèmes dans un programme, en particulier dans les parties optimisées pour une architecture donnée car critiques. C'est pourquoi celui-ci doit faire l'objet d'une attention particulière lors de la mise au point d'un système multi-processeurs. Des techniques sont en particulier nécessaires afin de vérifier qu'un certain MCM est correctement implanté.

1.3 Vérification de la consistance mémoire

La VCM consiste à vérifier que l'exécution d'un programme par un système multi-processeurs s'est produite conformément à un certain MCM. La VCM se contente de vérifier qu'une exécution donnée est correcte par rapport à un MCM : elle ne prouve en aucun cas que le système est correct. Prouver qu'un système est correct est complexe et constitue un domaine de recherche à part entière, mais cela ne fait pas partie de notre étude.

1.3.1 Principe de la vérification

Par rapport à l'étude des programmes d'illustration utilisés dans la partie précédente, la vérification est assez similaire. Elle diffère néanmoins puisque le résultat du programme est connu : il faut seulement vérifier qu'il y a bien une séquence qui mène à ce résultat. Comme nous l'avons esquissé dans la partie précédente, l'établissement des séquences possibles est assez fastidieux. De nombreux paramètres doivent être pris en compte pour établir si une séquence est acceptable par un MCM.

1.3.1.1 Dépendances du programme

Les dépendances de programme obligent le maintien de l'ordre initial d'une SI entre certains accès de celle-ci. Les dépendances d'un MCM sont comprises entre deux positions extrêmes.

- La première, la plus simple, est la plus contraignante. C'est celle du modèle Consistance Séquentielle (SC). Un accès est d'un certaine manière considéré dépendant de tous les précédents. Aucune flexibilité n'est permise pendant l'exécution d'une SI et les accès sont effectués dans l'ordre de celle-ci.
- La seconde est la plus flexible et n'impose que les contraintes minimales. L'unique condition est que le résultat d'une SI avec ces contraintes par un processeur, dans un contexte mono-processeur, doit donner le même résultat que si elle avait exécutée dans l'ordre. En contexte mono-processeur, seules les dépendances de type RAW, WAW ou WAR entre accès à la même adresse importent. On notera que, en particulier, l'ordre entre deux lectures à la même adresse n'a aucune importance (elles liront la même valeur dans tous les cas) et

peut donc être relâché, comme dans l'architecture *ARMv7*. Si elles sont inversées en multiprocesseurs, cela peut mener à des scénarios étranges car, une écriture extérieure peut très bien s'intercaler entre les deux.

On peut trouver plusieurs types de dépendances. Nous donnons ci dessous quelques catégories de dépendances.

Dépendances globales via la mémoire Ces dépendances sont celles qui sont généralement mises en avant. Le Tableau 1.6 ne montre ainsi que celles-ci. Elles ne dépendent pas des adresses utilisées par les accès mais uniquement du type des accès. Par exemple : l'ordre de la SI entre deux écritures est maintenu. Les autres dépendances servent uniquement à limiter les relâchements autorisés de manière générale.

Dépendances de données via une case mémoire Si les dépendances générales sont relâchées, il faut néanmoins assurer celles qui concernent deux accès à la même case mémoire. Pour conserver une exécution correcte, même en mono-processeur, seules les dépendances de type RaR à la même adresse peuvent être relâchées. Les dépendances de type RaW, WaW et WaR doivent être maintenues.

Dépendances de données internes d'une SI Les dépendances de données à l'intérieur d'une SI peuvent être maintenues par un MCM. Celles-ci représentent le fait que l'adresse d'un accès ou la donnée d'une écriture peuvent provenir de données lues par des accès précédents. Ces dépendances peuvent remonter très loin dans une SI, car des calculs intermédiaires sont généralement nécessaires pour calculer une adresse ou une donnée.

Dépendances de contrôle Les architectures traitent de manières différentes les branchements conditionnels effectués dans une SI. Des dépendances peuvent être par exemple mises en place sur les accès suivant un test par rapport à la donnée testée.

1.3.1.2 Sémantique de la mémoire

L'existence d'une séquence respectant les dépendances du programme ne suffit pas. Comme nous l'avons fait avec les exemples de programmes, il faut prendre en compte la sémantique de la mémoire. Cette sémantique signifie principalement qu'une lecture doit lire la valeur de la dernière écriture à la même adresse. Néanmoins, il peut y avoir des exceptions dans le cadre des écritures anticipées. Il existe ainsi un ordre des écritures à la même adresse, qui est géré par les mécanismes de maintien de la cohérence dans un SMPMP. Dans les exemples précédents nous nous appuyons par exemple sur le fait que la valeur initiale des variables est 0 pour en déduire qu'une lecture de cette valeur a forcément lieu avant toute écriture ou lecture d'une autre valeur. Par contre deux lectures de la même valeur ne sont pas ordonnées.

Mais la mémoire ne se limite pas forcément à de la mémoire classique : des espaces adressables correspondent à des périphériques. Les accès à ces périphériques, même les lectures, peuvent générer des effets de bords ou être sensibles à des événements extérieurs. Cela peut même concerner la mémoire classique quand elle contient des données spéciales. Par exemple la modification des tables de pages utilisées par le mécanisme de mémoire virtuelle entraîne des effets de bords. Ainsi dans l'architecture *ARMv7*, une écriture n'est considérée comme effectuée que quand les divers TLBs [4] ont été mis à jour lorsque cela est nécessaire.

Comme la mémoire contient aussi le programme, une écriture peut aussi impacter le programme en cours d'exécution. Du fait des pipelines et caches d'instructions, la prise en compte d'une écriture par le mécanisme de chargement des instructions n'est pas forcément immédiat ce qui a de l'importance dans le cadre d'un programme auto-modifiant ou simplement le chargement

4. Un TLB contient les informations de traduction d'une adresse virtuelle en une adresse physique.

dynamique de programme. Ce point est aussi spécifié par les MCMs des architectures. L'*ARMv7* dispose ainsi d'instructions appelées « *Instructions synchronisation barrier* » permettant de forcer la prise en compte des écritures précédentes par le mécanisme de chargement des instructions.

1.3.2 Problèmes liés à la vérification

La réalisation de la vérification de la consistance mémoire soulève plusieurs problèmes qui sont abordés dans cette partie.

1.3.2.1 Collecte des dépendances : besoin d'informations

Afin de pouvoir vérifier si les dépendances décrites dans la section précédente sont vérifiées, il faut dans un premier temps obtenir les informations sur celles-ci. Cela ne pose que peu de problèmes pour les dépendances de programme. Ces dépendances peuvent en effet être déduites uniquement de l'ordre des instructions de chaque SI exécutée un processeur. Les SIs étant dynamiques, la source du programme ne suffit pas, il faut connaître la séquence réelle des instructions exécutées par chaque processeur. Il faut aussi disposer d'informations dynamiques comme les adresses accédées par les instructions : celles-ci peuvent résulter de calculs.

Dans les exemples précédents, il était facile de trouver de quelle écriture provenait la donnée lue par chaque lecture. Les valeurs des écritures étaient en effet uniques pour chaque case mémoire, et permettaient ainsi d'identifier les écritures simplement grâce aux valeurs des données. Dans un programme réel, il n'y a, dans le cas général, aucune hypothèse de ce type qui peut être faite. Sans autre information permettant de déduire l'ordre des accès à une case mémoire, il n'y donc pas d'autres solutions que de tester divers scénarios jusqu'à en trouver un satisfaisant pour le MCM. S'il n'en existe aucun, alors le MCM n'est pas respecté. On notera que le problème de la vérification est dans ce cas NP-difficile par rapport au nombre d'accès qu'il y a eu durant l'exécution [CLS05].

Dans un système réel, l'observabilité est généralement extrêmement limitée. Ce qui est observable correspond à ce qui a été conçu pour l'être : un SMPMP contient des fonctionnalités permettant d'observer certaines parties de celui-ci. Dans notre contexte, la simulation, l'observabilité n'a théoriquement pas de limites. On peut ainsi imaginer, par exemple, de regarder aussi ce qu'il se passe au niveau de la mémoire ou des caches, ce qui n'est généralement pas possible dans un système réel, pour obtenir l'ordre des accès à une case mémoire.

1.3.2.2 Quantité d'informations : besoin d'un traitement efficace « à la volée »

Collecter les informations sur l'exécution est nécessaire pour réaliser une VCM. Néanmoins ces informations sont très nombreuses (au moins l'adresse de chaque accès à la mémoire de chaque SI) et les stocker prend beaucoup de place.

La stratégie, consistant à exécuter un programme par un SMPMP tout en collectant des informations, puis à vérifier si l'exécution a été correcte est limitée. L'espace de stockage nécessaire, qui est au moins de l'ordre du nombre d'accès à la mémoire, réduit les possibilités de vérification.

A l'opposé une vérification effectuée « à la volée » n'est pas dépendante des capacités de stockage. De plus, dans le cadre de la simulation des SMPMPs, cela permettrait de vérifier la consistance mémoire en parallèle de la simulation et éventuellement de manière systématique.

À la vue de la quantité d'informations qu'il est nécessaire de traiter, il est primordial de disposer d'une méthode efficace de vérification. Il n'y a en effet qu'une solution ayant un coût linéaire par rapport au nombre d'accès qui puisse passer à l'échelle dans le cadre de simulations moyennes ou longues.

1.3.3 Conclusion sur la vérification de la consistance mémoire

Dans cette partie, nous avons explicité les problèmes principaux de la Vérification de la Consistance Mémoire (VCM). Dans le cadre de la simulation, où l'observabilité est élevée, il semble néanmoins raisonnable de pouvoir extraire les informations nécessaires à la VCM.

Afin de ne pas être limité dans la VCM, il est toutefois nécessaire de disposer d'une méthode permettant de vérifier « à la volée » ne nécessitant pas de ressources de stockage démesurées. De plus le passage à l'échelle n'est possible que si la méthode de vérification a une complexité linéaire, en particulier en fonction du nombre d'accès.

1.4 Extraction des informations depuis la simulation

Comme nous l'avons vu précédemment, la VCM nécessite d'avoir à disposition de nombreuses informations. Néanmoins la VCM n'est pas la seule analyse qui nécessite beaucoup d'informations. D'autres situations bénéficient d'une observabilité accrue. On peut penser par exemple au problème des *data races* qui ne sont en rien empêchées par la consistance mémoire, même la SC.

La simulation offre de grandes possibilités d'extraction d'informations grâce à son observabilité. Dans cette partie, nous précisons donc quels sont les problèmes liés à cette extraction et à leur analyse, en particulier pour la VCM.

1.4.1 Besoin d'informations « transversales »

Il paraît relativement simple de pouvoir extraire des informations en différents points d'un SMPMP lors de sa simulation. Par exemple, on peut espérer regarder au niveau d'un processeur, les requêtes d'accès à la mémoire qu'il effectue. On peut aussi espérer pouvoir regarder l'ordre des accès à une certaine adresse mémoire, au niveau de la mémoire ou du périphérique. Par contre, pouvoir relier quelle requête d'un processeur correspond à quelle requête reçue par la mémoire est sans aucun doute beaucoup plus compliqué.

Il y a probablement de nombreux autres composants intermédiaires (par exemple un cache) qui peuvent avoir modifié la requête, voire même y avoir répondu. Si les systèmes de communication intègrent un mécanisme permettant d'identifier de manière unique les transactions entre les différents composants, cela permettrait de pouvoir faire le lien. Il n'y a néanmoins, dans le cas général, aucun mécanisme identifiant une requête depuis le processeur émetteur jusqu'à sa destination finale. Cela ne résoudrait d'ailleurs le problème que pour ce cas particulier des transactions sur un système de communication.

Les informations qu'il est possible d'obtenir en un point d'un SMPMP dépend par ailleurs grandement de la modélisation de celui-ci. Dans un modèle TLM, il est vraisemblable que le processeur soit représenté de manière monolithique et qu'on puisse obtenir diverses informations sur les opérations arithmétiques, les accès, etc. . Par contre si la modélisation est plus fine (et plus décomposée), il ne sera peut-être pas possible d'obtenir cela en un seul point. Et il sera nécessaire d'avoir recours à des informations extraites en plusieurs points, qu'il faudra recouper avec des informations transversales correspondant aux interactions entre les différentes parties du processeur.

De manière générale, il est ainsi souhaitable de pouvoir relier des informations extraites en différents points adaptés au niveau de modélisation du SMPMP si ces informations sont liées : par exemple, les informations concernant un même accès.

1.4.2 Impact sur la simulation

L'impact du fait de l'action d'extraire des informations sur la simulation d'un SMPMP peut se traduire sous deux formes :

– la première conséquence est un ralentissement de la vitesse de simulation,
– la deuxième conséquence correspond à une modification du comportement de la simulation.

Extraire des informations dans une simulation est forcément intrusif. Le ralentissement de la simulation est inévitable mais il n'est pas très grave s'il reste limité. Par contre les modifications du comportement ne sont pas souhaitables. Dans la mesure où l'on cherche à analyser des problèmes liés à des phénomènes de concurrence d'événements, il est très important de limiter au maximum l'impact comportemental car ce qui est analysé n'est pas le comportement initial mais le comportement modifié.

1.4.3 Conclusion : besoin d'une infrastructure d'analyse efficace

Récolter des informations de la manière la plus transparente possible est important mais il faut pouvoir les traiter. Il faut de plus pouvoir les traiter de manière efficace et dynamique. Comme nous l'avons en effet précisé auparavant pour la VCM, une analyse efficace en parallèle de la simulation est nécessaire pour pouvoir passer à l'échelle.

En plus de disposer d'une infrastructure permettant d'extraire des informations correspondant à ce qu'il se passe dans la simulation, cette infrastructure doit servir de support aux analyses de ces informations.

1.5 Conclusion de la problématique

En conclusion, les travaux de cette thèse concernent la problématique de l'utilisation de la simulation à des fins d'analyse des systèmes muti-processeurs à mémoire partagée. Le problème de la vérification de la consistance mémoire par ce biais est en particulier étudié. Plus précisément, nous tenterons de répondre aux questions suivantes :

– Comment vérifier si un modèle de consistance mémoire est bien respecté lors de l'exécution d'un programme d'une manière qui passe à l'échelle ? La vérification de la consistance mémoire se fait classiquement *a posteriori* et nécessite de ce fait une grande quantité de ressources de mémorisation. Malheureusement cela rend impossible la vérification de la consistance mémoire sur de longues exécutions à cause de la quantité de données mises en jeu. Afin de pallier à ce problème il est donc nécessaire de vérifier la consistance en même temps que la simulation tout en limitant au maximum le coût algorithmique et l'empreinte mémoire.
– Comment prendre en compte certaines caractéristiques spéciales des modèles de consistance mémoire ? Les modèles de consistance mémoire ne se limitent pas à définir comment sont gérées les lectures et les écritures à la mémoire centrale. Ils prennent en compte des problèmes bien plus fins tels que les accès spéciaux et les propriétés spécifiques des périphériques adressables en mémoire. Afin de réaliser la plus complète des vérifications, il est nécessaire de vérifier ces propriétés.
– Quelles sont les informations minimales nécessaires qu'il faut extraire de la simulation pour pouvoir effectuer cette vérification ? Il est théoriquement possible d'extraire toute information de la simulation. Afin de limiter l'impact sur la simulation, il est néanmoins nécessaire de n'extraire que peu d'informations. Il faut toutefois que ces informations soient le plus facilement accessibles pour éviter tout coût de traitement excessif.
– De manière plus générale, quelle infrastructure générique peut-on mettre en place pour permettre des analyses poussées sur ce qui se passe durant la simulation d'un système muti-processeurs à mémoire partagée ? Certaines analyses, comme la vérification de la consistance mémoire, sont compliquées à mettre en place car elles nécessitent de nombreuses informations sur le comportement du système muti-processeurs à mémoire partagée simulé.

Ces analyses se caractérisent par le besoin d'information disponible à différents emplacement du système simulé mais aussi par les interactions qu'il y a à travers le système. Afin de pouvoir les faciliter, une infrastructure d'extraction et d'analyse est nécessaire.

2 | État de l'art

Dans ce chapitre, nous faisons l'état de l'art de la vérification de la consistance mémoire. Nous formalisons tout d'abord le problème de la vérification de la consistance mémoire de manière très générique. Cette formalisation est nécessaire pour comprendre pourquoi la vérification de la consistance mémoire utilise une modélisation à base de graphes. Cette modélisation est présentée dans un second temps. Les techniques pour réaliser la vérification de la consistance mémoire sont ensuite décrites dans une troisième partie.

Chapitre 2

2.1 Formalisation de la vérification de la consistance mémoire 26
2.1.1 Un problème d'ordre 26
2.1.2 Formalisation générale 26
2.1.3 Formalismes existants 27
2.2 Représentation d'une exécution 27
2.2.1 Utilisation d'un graphe 27
2.2.2 Gestion de l'atomicité 28
2.2.3 La notion de case mémoire 32
2.2.4 Les accès spéciaux 32
2.3 Vérification de la consistance mémoire 32
2.3.1 Vérification formelle de la consistance mémoire 33
2.3.2 Complexité de la vérification de la consistance mémoire 33
2.3.3 Simplification du problème 34
2.3.4 Vérification dynamique 36
2.4 Conclusion 37

2.1 Formalisation de la vérification de la consistance mémoire

Avant d'aborder la formalisation, un rappel des notions utilisées est donné dans un premier temps. Les formalismes existants sont ensuite discutés.

2.1.1 Un problème d'ordre

Une *relation d'ordre* $\leq$ sur un ensemble $\mathcal{E}$ est une relation binaire :
- réflexive ($x \leq x, \forall x \in \mathcal{E}$),
- antisymétrique ($(x \leq y$ et $y \leq x) \Rightarrow x = y, \forall (x, y) \in \mathcal{E}^2$),
- transitive ($(x \leq y$ et $y \leq z) \Rightarrow x \leq z, \forall (x, y, z) \in \mathcal{E}^3$).

Une relation d'ordre est totale si $\forall (x, y) \in \mathcal{E}^2, x \leq y$ ou $y \leq x$. Dans le cas contraire elle est partielle. La *relation d'ordre strict*, qui n'est pas une *relation d'ordre*, associée à une *relation d'ordre* est alors définie comme ceci : $\forall (x, y) \in \mathcal{E}^2, x < y$ si, et seulement si, ($x \leq y$ et $x \neq y$). Cette relation est transitive et irréflexive ($\forall x \in \mathcal{E}$, non $x < x$).

Une relation binaire n'est en fait qu'un ensemble des couples qui sont en relation : $\leq = \{(x, y) | x \leq y\}$. Il est donc tout à fait possible de faire des unions de plusieurs relations. Si on a par exemple deux relations d'ordre $\leq_1$ et $\leq_2$ sur $\mathcal{E}$, l'union est alors l'ensemble $\leq_1 \cup \leq_2 = \{(x, y) | x \leq_1 y$ ou $x \leq_2 y\}$. On notera que même si les deux relations initiales sont des relations d'ordre, l'union n'en est généralement pas une car la transitivité en particulier n'est pas conservée. On utilise alors l'opération de *fermeture transitive* d'une relation R qui correspond à construire la relation transitive minimale incluant la relation R.

L'opération d'union suivie d'une fermeture transitive est notée $\cup_{tr}$. En pratique l'union simple n'est jamais utilisée dans notre contexte. $\cup_{tr}$ sur des relations d'ordre permet d'obtenir une relation réflexive et transitive. La relation obtenue n'est néanmoins pas forcément une relation d'ordre car elle n'est pas forcément antisymétrique. Par exemple, si $\leq = \leq_1 \cup_{tr} \leq_2$, et qu'il existe $x \neq y$ tel que $x \leq_1 y$ et $y \leq_2 x$. Alors, dans la relation obtenue, $x \leq y$ et $y \leq x$ bien que $x \neq y$ et $\leq$ n'est donc pas antisymétrique.

2.1.2 Formalisation générale

Comme nous l'avons illustré dans la problématique, la consistance mémoire est principalement une question d'ordre. La consistance mémoire fixe en effet des contraintes sur l'ordre dans lequel sont réalisés les accès à la mémoire qui sont les écritures et les lectures. La formalisation présentée dans cette partie est très abstraite afin de rester générique.

On nomme $\mathcal{A}$ l'ensemble des accès à la mémoire réalisée pendant l'exécution d'un programme. L'ordre du programme est noté $\leq_p$. Cet ordre est partiel et représente l'ordre des accès de chaque SI indépendamment les unes des autres. Deux accès x et y générés par deux processeurs différents ne sont pas ordonnés par $\leq_p$.

Le modèle de consistance définit alors un ordre $\leq_d$ dérivé de $\leq_p$ ($\leq_d \subseteq \leq_p$) qui représente les dépendances du programme. Ces dépendances contraignent l'exécution du programme. On rappelle que ces contraintes peuvent exprimer différentes dépendances comme les dépendances de données à travers les registres d'un processeur ou celles à travers la mémoire. En fonction du modèle de consistance, $\leq_d$ est un sous ensemble plus ou moins réduit de $\leq_p$. Dans le modèle SC par exemple, $\leq_d = \leq_p$.

L'ordre partiel des accès à la mémoire est noté $\leq_m$. Cet ordre représente les contraintes liées à la sémantique de la mémoire. Cet ordre existe uniquement car la mémoire est cohérente.

Dans un modèle atomique, chaque lecture est par exemple ordonnée entre l'écriture dont elle lit la donnée et l'écriture suivante à la même adresse. Cela représente le fait qu'une lecture lise la donnée écrite par la dernière écriture à la même adresse. Ainsi si une case de la mémoire est

écrite deux fois : d'abord par W_0 puis par W_1 et que deux lectures, R_3 et R_4, lisent la donnée écrite par W_0. Alors on a $W_0 \leq_m R_3 \leq_m W_1$ et $W_0 \leq_m R_4 \leq_m W_1$. Par contre il n'y a ni $R_3 \leq_m R_4$ ni $R_4 \leq_m R_3$.

Pour qu'une exécution vérifie un modèle de consistance mémoire, il faut que les ordres $\leq_d$ et $\leq_m$ soient compatibles. C'est à dire que $\leq_m$, l'ordre dans lequel les accès se produisent, ne soit pas contradictoire avec $\leq_d$, l'ordre dans lequel ils doivent se produire. C'est le cas si il existe au moins une séquence ordonnée des accès de $\mathcal{A}$ dans laquelle $\leq_d$ et $\leq_m$ sont respectés. En termes mathématiques, cela correspond à vérifier que $\leq = \leq_d \cup_{\text{tr}} \leq_m$ est une relation d'ordre.

2.1.3 Formalismes existants

$\leq_m$ et $\leq_d$ sont en pratique complètement dépendants du modèle de consistance mémoire. Les différentes documentations techniques des architectures multi-processeurs que nous avons vues dans le chapitre précédent les définissent de manière plus ou moins formelle. Certaines sont très formelles comme celle de l'architecture SPARC [SPARC]. D'autres sont principalement basées sur de petits exemples de programme comme celle de l'architecture TilePro64 [Tile64].

Plusieurs formalismes ont été proposés [Alg+09 ; CSB93 ; SFC91 ; SN04] pour décrire un MCM. Ces formalismes sont plus ou moins génériques : certains formalismes ne permettent pas de représenter tous les MCMs. Le travail le plus complet est celui de STEINKE et NUTT [SN04] qui définissent un formalisme permettant d'exprimer de nombreux MCMs afin de réaliser une classification de ceux-ci. Le lecteur intéressé y trouvera une description mathématique de nombreux modèles de consistance basée en partie sur des relations d'ordre. Dans leur formalisme, un accès est toujours représenté par un unique nœud ou élément. Si plusieurs visions de ces accès sont possibles par différents processeurs par exemple, plusieurs ordres indépendants sont définis. Pour faire un parallèle avec la formalisation générale donnée ci-dessus, cela correspond à définir plusieurs ordres $\leq_d^i$ et $\leq_m^i$ pour différentes valeur de i. Le MCM est respecté si, et seulement si, chacun des $\leq^i = \leq_d^i \cup_{\text{tr}} \leq_m^i$ est une relation d'ordre. Pour certains MCMs, i correspond à un processeur. Cela est le plus intuitif car cela permet de différencier clairement les différentes visions de chacun des processeurs.

Ce formalisme permet de classer et comparer les différents modèles, ce qui est l'objectif de STEINKE et NUTT, mais n'est pas très adapté pour réaliser la vérification. Il est néanmoins possible de construire un unique ordre $\leq$ en dupliquant les éléments présents dans plusieurs ordres $\leq^i$. De cette manière chaque $\leq^i$ concerne des éléments qui lui sont spécifiques et l'union ($\cup_{\text{tr}}$) des différents $\leq^i$ ne pose aucun problème.

2.2 Représentation d'une exécution pour la Vérification de la Consistance Mémoire

Différentes solutions existent pour représenter le problème de la VCM. Il existe en particulier différentes solutions pour pouvoir gérer l'atomicité et l'anticipation des écritures. Dans cette partie nous présentons les méthodes permettant de représenter les MCMs en fonction de ces propriétés.

2.2.1 Utilisation d'un graphe

Les relations binaires et en particulier les relations d'ordre sont représentables par des graphes. Dans un graphe représentant une relation binaire, les nœuds sont les éléments de $\mathcal{E}$ et une relation $x \leq y$ est représenté par un arc orienté noté $x \rightarrow y$. Si $\leq$ est une relation d'ordre, on représente en pratique la relation d'ordre strict associée ($<$) pour éviter de mettre un arc entre chaque nœud et lui-même. En utilisant la transitivité de la relation il est aussi possible de ne représenter qu'une infime partie des relations entre les nœuds. Par exemple, seuls $x \rightarrow y$ et $y \rightarrow z$ sont nécessaires

pour représenter la relation d'ordre strict $\{x < y, y < z, x < z\}$. Comme généralement seuls les prédécesseurs et successeurs directs sont connus dans les relations partielles, cela simplifie aussi grandement la génération d'un tel graphe.

Dans notre contexte, une telle représentation a un intérêt majeur : il est extrêmement simple de réaliser l'union ($\cup_{\text{tr}}$) de plusieurs ordres. Ainsi pour représenter $\leq = \cup_{\text{tr}}^{i} \leq_i$ constitué de différentes relations d'ordre $\leq_i$, il suffit de représenter ces différentes relations d'ordre dans un même graphe. Une propriété intéressante de ce graphe est qu'il est alors acyclique si, et seulement si, $\leq$ est une relation d'ordre. Vérifier la consistance mémoire consiste donc à représenter $\leq = \leq_d \cup_{\text{tr}} \leq_m$ dans un graphe orienté et vérifier si celui-ci est acyclique.

Il y a deux stratégies pour vérifier si un graphe contenant n nœuds et m arcs est acyclique.

- La première consiste à vérifier la présence de cycles sans connaissance préalable sur le graphe. Un algorithme souvent utilisé dans ce cas consiste à faire un parcours du graphe de type *Depth First Search* [Tar71]. Cet algorithme a une complexité algorithmique linéaire en fonction de la taille du graphe ($O(n+m)$). Il est donc efficace s'il est utilisé ponctuellement. L'utiliser à chaque ajout d'un arc coûterait en effet très cher au total.
- La seconde consiste à maintenir en permanence un ordre topologique entre les nœuds du graphe. Ainsi cela permet de détecter, pour un coût en $O(1)$, dès l'ajout d'un arc si celui-ci ferme un cycle dans le graphe. Le maintien de l'ordre topologique requiert néanmoins un traitement à chaque ajout d'un arc. La mise au point d'algorithmes efficaces pour maintenir un tel ordre est toujours en cours. Actuellement des algorithmes permettent l'ajout d'un arc dans un graphe en $O(m^{3/2})$ ou en $O(n^{5/2})$ suivant la méthode utilisée [Hae+12]. Ces algorithmes sont utiles s'il est nécessaire de détecter un cycle dès sa formation.

Les techniques que nous allons présenter dans ce chapitre se classent grossièrement en deux catégories : soit elles utilisent un graphe et vérifient les cycles une seule fois à la fin ; soit elles utilisent un ordre topologique mais celui-ci est extrait de l'exécution et il n'y a pas besoin de graphe.

2.2.2 Gestion de l'atomicité

2.2.2.1 Représentation des accès atomiques

La représentation classique considère un accès comme étant un nœud des graphes servant à représenter les ordres $\leq_d$ et $\leq_m$. C'est cette représentation qui a été initialement utilisée dans les premiers travaux sur la VCM [GK92]. C'est en effet la représentation la plus intuitive car en utilisant un unique nœud, l'accès est considéré comme un événement indivisible. Cette représentation correspond en fait aux modèles atomiques n'autorisant pas les écritures anticipées. On compte parmi ces modèles la consistance séquentielle.

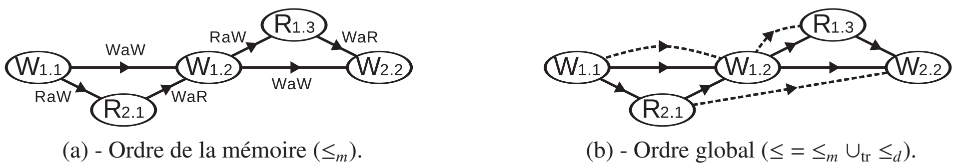
SI_1	$W_{1.1}(x,1)$, $W_{1.2}(x,2)$, $R_{1.3}(x,2)$
SI_2	$R_{2.1}(x,1)$, $W_{2.2}(x,3)$

TABLEAU 2.1 - Exécution d'accès à $[x]$.

FIGURE 2.1 - Graphe de $\leq_d$.

Le TABLEAU 2.1 présente un exemple d'exécution d'accès à une même case mémoire ($[x]$) d'un programme contenant deux SIs (SI_1 et SI_2). Les écritures sont notées $W_{p.i}(x,w)$ où p indique la SI, i indique l'indice de l'accès dans la SI. w indique l'indice de l'écriture dans l'ordre auxquelles elles se sont produites dans la case mémoire et permet donc d'identifier de manière unique les écritures. Les lectures sont notées de manière similaire $R_{p.i}(x,w)$ mais w indique dans ce cas l'indice de l'écriture lue. La FIGURE 2.1 représente le graphe correspondant à l'ordre du programme ($\leq_d$) de

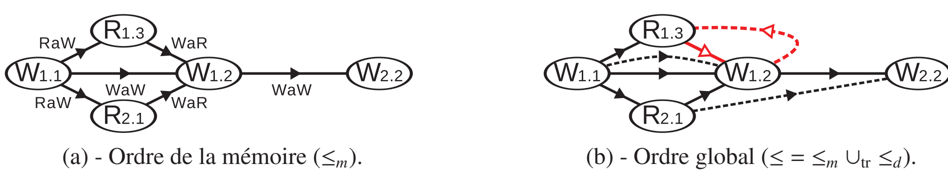
cette exécution. Les arcs sont étiquetés avec le type de contraintes qu'ils représentent (*Read after Write* (RaW), *Write after Read* (WaR) ou *Write after Write* (WaW)). L'ordre du programme ($\leq_d$) est en particulier complètement indépendant du résultat des lectures.



(a) - Ordre de la mémoire ($\leq_m$). (b) - Ordre global ($\leq = \leq_m \cup_{tr} \leq_d$).

Figure 2.2 - Graphes de $\leq_m$ et $\leq$ correspondant à l'exemple d'exécution.

La Figure 2.2a montre le graphe de $\leq_m$ correspondant à l'exécution du Tableau 2.1. Dans ce graphe, il n'y a aucune différence de traitement entre $R_{1.3}$ qui lit une écriture de sa propre SI et $R_{2.1}$ dont ce n'est pas le cas. $\leq$ est représenté dans la Figure 2.2b. Le graphe de $\leq$ est représenté en gardant la disposition des nœuds du graphe de $\leq_m$. Les arcs de $\leq_d$ (représenté dans la Figure 2.1) sont ajoutés en plus de ceux de $\leq_m$. Dans $\leq$ les arcs de $\leq_m$ sont des flèches pleines tandis que les arcs de $\leq_d$ sont en pointillés. Entre deux nœuds, certains arcs sont présents à la fois dans $\leq_d$ et $\leq_m$. Les arcs étant tous orientés de gauche à droite, il n'y a par contre aucun cycle dans $\leq$: les deux ordres concordent.



(a) - Ordre de la mémoire ($\leq_m$). (b) - Ordre global ($\leq = \leq_m \cup_{tr} \leq_d$).

Figure 2.3 - Graphes de $\leq_m$ et $\leq$ si $R_{1.3}$ lit $W_{1.1}$.

La Figure 2.3 représente l'exécution du programme en considérant que $R_{1.3}$ ne lit pas la valeur retournée par l'écriture $W_{1.2}$ mais celle de $W_{1.1}$. Cela est clairement interdit puisque $W_{1.2}$ est située entre $W_{1.1}$ et $R_{1.3}$ dans SI_1. Il n'y a aucun changement de l'ordre du programme : $\leq_d$ est toujours celui représenté dans la Figure 2.1. $\leq_m$ est par contre impacté et est représenté dans la Figure 2.3a. Le graphe de $\leq$ l'est aussi et est représenté dans la Figure 2.3b en conservant la disposition des nœuds du graphe de $\leq_m$. Cette fois, on peut voir que l'ajout des arcs de $\leq_d$ forme un cycle (arcs avec flèches blanches). L'exécution est donc bien considérée comme invalide.

2.2.2.2 Représentation des écritures anticipées

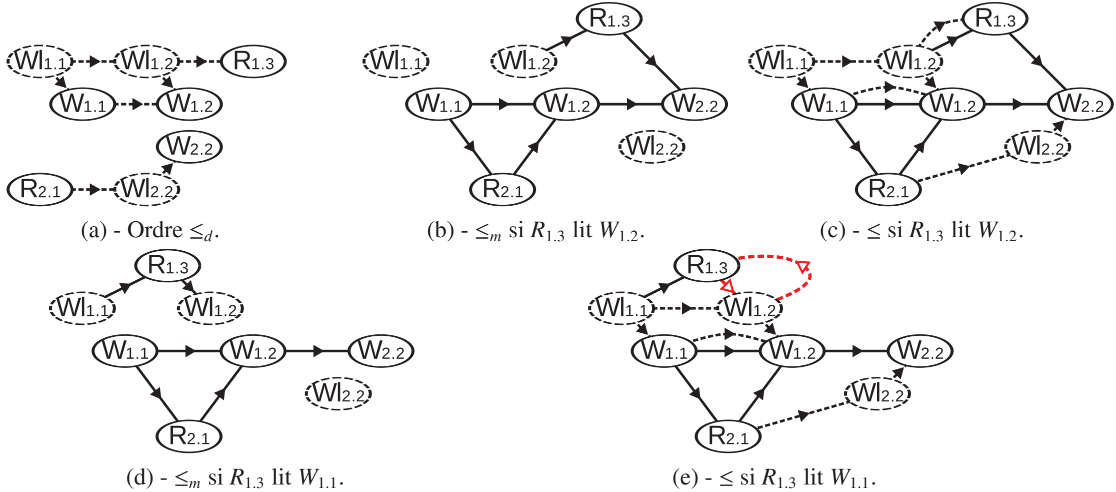
(a) - $\leq$ si $R_{1.3}$ lit $W_{1.2}$. (b) - $\leq$ si $R_{1.3}$ lit $W_{1.1}$.

Figure 2.4 - Graphe de $\leq$ sans les dépendances RaW entre accès de la même SI.

La gestion des écritures anticipées pose un réel problème de représentation. Pour autoriser une lecture à se produire avant son écriture associée, il faut nécessairement supprimer les contraintes de type RaW entre accès de la même SI. La Figure 2.4 représente le graphe obtenu lorsque l'on supprime ces arcs. Le graphe représente le cas interdit où $R_{1.3}$ lit $W_{1.1}$. Il correspond au graphe

de la FIGURE 2.3b sans les arcs correspondants à $W_{1.2} \leq_d R_{1.3}$ et $W_{1.1} \leq_m R_{1.3}$. La suppression du premier arc supprime le cycle qui existait. Cette suppression permet de représenter les écritures anticipées mais ne représente plus les dépendances locales qui persistent. Différentes solutions existent pour les représenter.

La manière la plus intuitive pour représenter ces dépendances est de représenter la non-atomicité locale des écritures. Du fait des ces écritures anticipées, l'écriture lue n'est en effet plus atomique et ne peut donc plus être représentée par un seul nœud. Une solution consiste donc à représenter une écriture en deux parties distinctes : l'écriture locale et l'écriture globale. L'écriture locale correspond à celle vue par la SI qui a émis l'écriture tandis que l'écriture globale à celle vue par les autres SIs. Cette représentation est utilisée par CONDON *et al.* dans leur travaux sur la vérification de la consistance mémoire [Con+99].



(a) - Ordre $\leq_d$. (b) - $\leq_m$ si $R_{1.3}$ lit $W_{1.2}$. (c) - $\leq$ si $R_{1.3}$ lit $W_{1.2}$.

(d) - $\leq_m$ si $R_{1.3}$ lit $W_{1.1}$. (e) - $\leq$ si $R_{1.3}$ lit $W_{1.1}$.

FIGURE 2.5 - Graphes utilisant des nœuds pour représenter les écritures visibles localement.

La FIGURE 2.5 illustre ainsi cette représentation. Les nœuds $W_{p.i}$ représentent les écritures globales et les nœuds représentant les écritures locales sont notées $Wl_{p.i}$ et sont en pointillés. La FIGURE 2.5a représente $\leq_d$. Dans celui-ci les écritures locales sont ordonnées avant les écritures globales et les lectures sont ordonnées avec les écritures locales. Les FIGURES 2.5b et 2.5c représentent $\leq_m$ et $\leq$ dans le cas où $R_{1.3}$ lit bien $W_{1.2}$. Les FIGURES 2.5d et 2.5e représentent le cas où $R_{1.3}$ lit par erreur l'écriture $W_{1.1}$. Dans $\leq_m$ le principe général est toujours de placer une lecture entre l'écriture lue et la suivante. Néanmoins le nœud d'une écriture locale est utilisé si la lecture est issue de la même SI. Les nœuds correspondant aux lectures sont ordonnés avec les nœuds locaux si elles sont issues de la même SI. On peut remarquer qu'en supprimant les nœuds locaux des FIGURES 2.5c et 2.5e, on retrouve les graphes des FIGURES 2.4a et 2.4b.

(a) - $\leq$ si $R_{1.3}$ lit $W_{1.2}$. (b) - $\leq$ si $R_{1.3}$ lit $W_{1.1}$.

FIGURE 2.6 - Graphes globaux sans nœuds pour les écritures locales.

Une autre solution consiste à s'assurer qu'une lecture lit bien au moins la dernière écriture de sa SI. Pour cela, une possibilité [Han+04] est d'ajouter une contrainte entre l'écriture précédant la lecture dans sa SI et l'écriture lue ($W_{\text{précédente}} \leq W_{\text{lue}}$). En pratique comme c'est une relation ≤, il ne faut ajouter l'arc que si les deux écritures sont différentes. La FIGURE 2.6 représente ce cas de figure. L'arc ajouté est étiqueté avec un « ? ». Dans le cas où $R_{1.3}$ lit $W_{1.2}$ (figure 2.6a), l'arc n'est pas réellement ajouté car l'écriture précédant $R_{1.3}$ dans SI_1 est l'écriture lue. Dans le cas de la FIGURE 2.6b, l'écriture précédant $R_{1.3}$ est $W_{1.2}$ et l'écriture lue est $W_{1.1}$: l'arc est ajouté et ferme un cycle. Seul les ordres globaux pour les deux résultats de la lecture $R_{1.3}$ sont représentés. Dans la FIGURE 2.6a, l'arc n'est pas réellement mis car la source et la destination sont identiques. Cette solution, bien que plus simple, n'est pas forcément plus facile à mettre en place. Il faut en effet pouvoir retrouver à la fois la dernière écriture dans la SI et l'écriture lue.

2.2.2.3 Représentation des écritures non atomiques

Lorsque les écritures ne sont pas globalement atomiques, il devient impossible d'utiliser un seul nœud pour chaque écriture en utilisant un unique graphe. Parmi ces modèles, on trouve les modèles *weak consistency* [DSB86 ; AH90] et *release consistency* [Gha+90] qui ne diffèrent que par leurs opérations de synchronisation ou barrières mémoire. Sous le modèle *release consistency*, la barrière mémoire est ainsi divisée en deux opérations distinctes complémentaires *acquire* et *release*.

LINDER et HARDEN [LH94] proposent une représentation où chaque accès est divisé en quatre nœuds correspondant aux différentes étapes du traitement de cet accès. Ces étapes sont l'initiation de l'accès par le processeur, l'initiation de l'accès par le système mémoire, la complétion de l'accès par le système mémoire et la complétion de l'accès par le processeur. Pour chaque accès ces quatre nœuds se suivent et permettent de représenter le fait que réaliser un accès à la mémoire n'est en pratique pas instantané et qu'il est donc possible que plusieurs accès soient simultanément en cours de réalisation. Ceci permet d'autoriser que temporairement la dernière écriture à une case mémoire ne soit pas la même dans l'ensemble du système. LINDER et HARDEN utilisent cette représentation pour vérifier si l'exécution d'un programme a respecté un certain modèle de synchronisation. Ce modèle de synchronisation définit des règles pour obtenir une exécution vérifiant la consistance séquentielle sur une plateforme matérielle qui ne l'est pas. Ces modèles de synchronisation sont utilisés par les développeurs de programmes parallèles pour réaliser des programmes n'ayant, par exemple, pas de *data-races*. Ils sont basés sur l'utilisation de primitives de synchronisation tels que les *mutex* de *POSIX*.

Cette représentation n'est néanmoins pas adaptée pour vérifier des modèles de consistance non atomiques. La présence d'un nœud correspondant à la complétion globale d'une écriture permet de gérer des synchronisations globales. Ainsi il est possible de contraindre que tous les processeurs voient une écriture avant qu'un des processeurs puisse voir la suivante. Par contre il est impossible de contraindre chaque processeur à voir une écriture avant la suivante.

A l'inverse la solution utilisée par CHONG et ISHTIAQ [CI08] permet de représenter une telle contrainte mais échoue à représenter des synchronisations globales. Ils utilisent une représentation basée sur un graphe qui différencie plusieurs types d'arcs. La vérification consiste dans leur cas à rechercher des cycles dans le graphe pour chaque processeur. A chaque fois les arcs qui sont exclusifs aux autres processeurs sont ignorés.

La représentation des systèmes matériels autorisant les écritures non atomiques donnée par GHARACHORLOO [Gha95] subdivise explicitement chaque écriture en autant de sous-écritures qu'il y a de processeurs. En représentant chaque sous-écriture par un nœud différent, les deux types de contraintes peuvent être modélisés.

Il n'est pas impossible de devoir représenter à la fois des écritures non-atomiques et des écritures anticipées. Le modèle *processor consistency* [Aha+93] le nécessite par exemple. Dans ce cas il faut cumuler la subdivision d'une écriture du fait de la non-atomicité avec une des techniques permettant de représenter le fait d'anticiper localement une écriture.

Dans la formalisation du modèle de l'Itanium [Itanium] chaque écriture est ainsi divisée en $P+1$ opérations distinctes, une pour l'écriture locale anticipée et P pour les écritures correspondant à chaque processeur.

2.2.3 La notion de case mémoire

Une case mémoire représente une zone de la mémoire d'une certaine taille. L'espace mémoire est ainsi divisé en plusieurs cases. La cohérence assure l'atomicité des accès au sein d'une même case mémoire mais pas entre deux cases mémoire différentes. Dans le cas le plus général, une case mémoire a la taille de la plus petite zone de la mémoire qui peut être accédée. Cela correspond généralement à un octet. Néanmoins représenter une division si fine de l'espace mémoire peut coûter cher en particulier s'il est nécessaire de garder des données pour chaque case mémoire.

Les processeurs sont de plus capables de faire des accès atomiques sur plusieurs octets contigus. C'est pour cette raison que les méthodes de vérification considèrent les cases mémoire de la taille des mots accédés par les processeurs [CLS05]. Les processeurs ne pouvant pas faire, la plupart du temps, d'accès atomiques non alignés sur des mots, cela n'a pas d'incidence. Il est même possible de calquer les cases mémoire sur les lignes de caches, qui correspondent généralement aux zones mémoires cohérentes [MS05].

Toutes les cases mémoire n'ont, par ailleurs, pas forcément la même sémantique d'utilisation. Certaines, comme celles qui correspondent à des registres de périphériques, ne sont pas de simples cellules de mémorisation. Des effets de bords sont possiblement associés aux accès. Le contenu de certaines cases mémoire peut aussi changer en fonction d'événements extérieurs indépendants des processeurs. Dans leur travail sur la vérification des modèles implantés dans les architectures *IA-32* et *Itanium*, Roy *et al.* [Roy+06] gèrent cinq catégories différentes de cases mémoire. Une catégorie, qui correspond en pratique à la zone mémoire du *frame buffer*, permet par exemple de réordonner les écritures, alors que toutes les autres ne le permettent pas.

2.2.4 Les accès spéciaux

La plupart des accès à la mémoire sont les lectures et les écritures. Certaines variantes ou compositions de ces accès existent néanmoins. En particulier, les opérations d'échanges (ou *swaps*) consistent à lire puis écrire une donnée de manière atomique. Ces opérations peuvent être éventuellement conditionnées.

Un échange peut être modélisé par un couple d'une lecture et d'une écriture. La plupart des travaux font l'hypothèse que le *swap* est une unique opération [Han+04 ; Roy+06] et ne vérifient donc pas l'atomicité de celui-ci.

Aujourd'hui de nombreux *swap* sont en fait le résultat de deux accès distincts effectués par le programme. Par exemple, le couple formé d'un *Load Linked* (LdL) et d'un *Store Conditional* (StC) permet de réaliser un *swap*. Il faut alors vérifier que le couple est bien effectué de manière atomique : aucune écriture ne doit s'immiscer entre la lecture et l'écriture.

2.3 Vérification de la consistance mémoire

Dans cette partie nous étudions les différentes techniques de VCM. L'évaluation de la complexité de ces techniques s'exprime à partir de paramètres de l'exécution. N représente ainsi le nombre total d'accès et P le nombre de processeurs (ou de SIs) ayant exécuté le programme.

2.3.1 Vérification formelle de la consistance mémoire

De nombreux travaux ont été effectués pour essayer de trouver des solutions pour prouver qu'un système vérifie bien un MCM. À la différence de la vérification d'une exécution particulière, ces travaux s'attaquent au problème de prouver que le système est correct. Cela implique alors que toutes les exécutions issues de ce système vérifieront le MCM. Ce problème est indécidable dans sa généralité [AMP96] et il est nécessaire de faire des hypothèses simplificatrices. Dans ce cadre, la consistance mémoire est donc analysée par rapport au protocole de communication ou protocole de cohérence de cache. Cela est donc fait à très haut niveau d'abstraction puisque seules les machines à états du protocole sont généralement modélisées.

2.3.1.1 Preuve par raisonnement

Dans certains cas il est possible d'exprimer la description des modèles de protocole de communication d'une manière permettant de raisonner formellement sur les exécutions permises par ce protocole. Landin, Hagersten et Haridi ont ainsi étudié une catégorie de systèmes appelés *Race free networks* qui impose des contraintes sur les communications. Ces travaux permettent de définir quelles sont les conditions pour implémenter différents MCMs [LHH91].

D'autres travaux utilisent les horloges de Lamport pour prouver sans exploration qu'un protocole de cohérence permet d'implanter le modèle SC [Con+99 ; Pla+98 ; Sor+98]. Cette technique consiste à décrire le protocole de cohérence avec un étiquetage approprié des messages échangés lors des communications. En étendant les horloges de Lamport, des modèles quasiment atomiques tels que le modèle TSO ont été vérifiés [CH01].

2.3.1.2 *Model checking*

La technique du *model checking* permet d'explorer l'ensemble des états possibles atteignables du système afin de vérifier qu'ils sont corrects. Afin de limiter le nombre d'états à explorer, les travaux utilisant cette technique étudient des modèles de haut niveau de protocoles de cohérence de cache. Comme les processeurs ne sont pas modélisés, les réordonnancements associés ne le sont pas non plus. Cela permet de vérifier le modèle SC lorsque l'on considère les accès atomiques [HQR99 ; Qad03].

A l'opposé certains travaux vérifient uniquement si la cohérence est vérifiée [Ati+10 ; CD06 ; PD00]. Ces cas possèdent des caractéristiques qui permettent de réduire énormément la taille de l'espace à explorer [PD95]. Pour l'étude de la cohérence par exemple, les cases mémoire sont indépendantes donc il est possible de se limiter à l'étude d'une seule case mémoire.

Ces techniques sont utiles lors de la définition des protocoles de communication utilisés dans un système mais sont néanmoins limitées. Dès que le niveau de modélisation devient précis, et en particulier d'un niveau correspondant à une implantation, elles sont inutilisables à cause de l'explosion combinatoire des cas possibles.

2.3.2 Complexité de la vérification de la consistance mémoire

2.3.2.1 Complexité de la vérification de la consistance séquentielle

Le problème majeur avec la vérification de la consistance mémoire est l'obtention de l'ordre $\leq_m$. Dans un système réel, les seules informations que l'on peut obtenir sont souvent à propos du programme. Le programme ($\leq_p$) est en effet connu et permet donc d'établir $\leq_d$. Il est aussi possible d'obtenir facilement les valeurs écrites et lues en mémoire pour chaque accès en observant ou traçant le programme. Mais cela ne permet pas de reconstituer $\leq_m$. Ainsi pour vérifier la consistance mémoire, il faut éventuellement tester de nombreux ordonnancements possibles des accès à

la mémoire, ce qui est très coûteux. Le problème n'est alors pas seulement un problème de vérification mais un problème d'existence d'une séquence vérifiant la sémantique correspondant à $\leq_m$. Gibbons et Korach ont par exemple démontré que vérifier la SC d'une exécution d'un programme utilisant trois processeurs est NP-complet dans ces conditions [GK94 ; GK97].

2.3.2.2 Complexité générale

D'autres travaux ont étudié d'autres modèles de consistance [CLS03 ; GPS04 ; MH05]. Par exemple Manovit et Hangal [MH05] ont étudié le cas du modèle TSO qui est utilisé dans les architectures *SPARC*. A l'opposé de la consistance séquentielle, qui est très contraignante, la vérification de la cohérence mémoire, qui est très relâchée, est aussi un problème NP-complet [CLS03].

De manière générale, selon Cantin, Lipasti et Smith [CLS05], vérifier n'importe quel modèle qui implique la cohérence mémoire est un problème NP-dur. Même en présence d'une exécution cohérente, vérifier la consistance mémoire reste compliqué. Ainsi vérifier si une exécution vérifie la SC sachant qu'elle est cohérente est toujours NP-complet [CLS05].

2.3.3 Simplification du problème

De nombreux algorithmes pour vérifier la consistance mémoire ont été proposés. À cause de la complexité, de nombreuses solutions effectuent une vérification de la consistance dite incomplète pour diminuer la complexité. Cela signifie que l'algorithme peut accepter des exécutions qui ne vérifient en fait pas le modèle de consistance. A l'inverse une solution complète garantit que l'exécution respecte vraiment un modèle de consistance. Les différentes solutions se basent sur l'obtention d'informations supplémentaires concernant $\leq_m$.

2.3.3.1 Utilisation de l'association des lectures

L'association des lectures est une catégorie d'informations très importante. Elle permet d'associer chaque lecture avec l'écriture dont la donnée est lue. Cette information pourrait, par exemple, provenir du système mémoire matériel. En pratique, il est difficile sur un système réel d'obtenir cette information parce qu'aucune information n'est généralement accessible sur les opérations effectuées par le système mémoire.

Néanmoins l'association des lectures peut être obtenu directement depuis le programme exécuté dans certaines conditions. Si chaque écriture écrit une valeur unique, cette valeur permet d'identifier l'écriture. Ainsi tracer le résultat des valeurs lues et des valeurs écrites permet d'obtenir cette association. Cette technique est néanmoins restreinte à des programmes très spécifiques développés uniquement dans ce but. L'environnement *TSOTools* [Han+04] permet, par exemple, de générer des programmes de test pseudo-aléatoires qui effectuent des *data-races* à des cases de la mémoire partagée. Ces programmes sont construits de manière à n'écrire que des valeurs uniques pour pouvoir ensuite retrouver l'association des lectures. *TSOTools* permet ensuite d'analyser la trace du programme pour définir s'il a respecté le modèle TSO.

Manovit et Hangal [MH06] proposent plusieurs algorithmes pour vérifier si une exécution se comporte selon le modèle TSO de l'architecture *SPARC*. Ces algorithmes sont basés sur la connaissance du *read mapping*. Les complexités obtenues pour les deux algorithmes sont respectivement $O(PN^3)$ et $O((N/P)^P PN^3)$ pour la version incomplète et complète. Leur technique consiste à utiliser une heuristique utilisant des vecteurs d'horloges [CL02] ce qui leur permet d'effectuer un tri topologique parmi les nœuds du graphe. La version complète nécessite néanmoins un mécanisme de *backtracking* très coûteux.

Roy *et al.* [Roy+06] proposent d'étendre la solution de Manovit et Hangal à d'autres modèles de consistance. Ils peuvent ainsi vérifier une exécution par rapport à n'importe quel modèle de consistance ayant des écritures atomiques mais autorisant les écritures anticipées. Cela inclut en

particulier les modèles des architectures *SPARC*, *IA-32* et des implantations actuelles de l'*Itanium*. Cet algorithme effectue une vérification incomplète avec une complexité de $O(N^4)$. Pour atteindre cette complexité, l'occupation mémoire nécessaire est en $O(N^2)$. L'algorithme peut être étendu aux écritures non atomiques pour un facteur algorithmique égal au nombre de processeurs P.

2.3.3.2 Utilisation d'informations temporelles

Afin de réduire encore la complexité des algorithmes de vérifications, des informations temporelles peuvent être utilisées. Dans leur travaux sur la vérification de la consistance mémoire du *Godson*, Chen *et al.* [Che+09] ont réduit la complexité des algorithmes à $O(P^3N)$ pour une vérification incomplète et $O(C^P P^2 N^2)$ pour une vérification complète (C est une constante).

Leur solution consiste à se servir d'informations temporelles récoltées sur les accès à la mémoire pour limiter le nombre d'accès qui sont susceptibles d'être en concurrence. En pratique un accès n'est en effet pas concurrent avec la totalité des accès : un accès ayant été propagé à l'ensemble du système à une certaine date peut nécessairement être ordonné avant n'importe quel accès n'ayant pas encore été initié à cette même date. En regardant régulièrement dans le programme testé une horloge globale, il est possible de borner les dates d'initiation et de complétion de chaque accès à la mémoire. Ces bornes permettent de limiter efficacement la quantité d'accès qui sont peut-être concurrents et donc de limiter la complexité.

2.3.3.3 Utilisation de l'ordre des écritures

La deuxième catégorie d'informations très importantes est appelée l'ordre des écritures. Comme son nom l'indique, il permet d'ordonner les écritures, mais uniquement pour chaque case mémoire. Il ne donne pas d'ordre global des écritures. Cette catégorie (*write order*), ainsi que l'association des lectures (*read mapping*) ont été définis par Gibbons et Korach. Ces deux catégories d'informations permettent en fait de reconstituer entièrement l'ordre $\leq_m$. Ainsi avec ces informations ($\leq_m$) et la connaissance du programme exécuté ($\leq_d$) il suffit de construire le graphe correspondant à $\leq$ et de vérifier qu'il est acyclique.

Gibbons et Korach [GK92] donnent un algorithme en $O(N.log(N))$ pour la vérification de la SC si on connaît l'association des lectures et l'ordre des écritures. La partie de plus grande complexité de leur algorithme est un tri préalable de l'ensemble des accès permettant de construire le graphe. La vérification du graphe coûte seulement $O(N)$. Manovit et Hangal [MH05] proposent un algorithme qui effectue la vérification en $O(N)$ pour le modèle TSO. Cet algorithme est plus performant car il ne nécessite pas de trier les accès. Il repose en particulier sur la possibilité de trouver en $O(1)$ l'écriture qui est lue par une lecture. Ceci n'est pas possible avec le formalisme utilisé par Gibbons et Korach.

Des travaux récents [RHS12] proposent une solution pour vérifier la consistance à partir de deux traces issues de chaque processeur. Une trace correspond à l'ordre du programme, l'autre à l'ordre à l'entrée du système mémoire (par exemple le cache de niveau 1). La deuxième trace contient des *timestamps* permettant d'ordonner globalement les éléments qu'elle contient. En utilisant ces deux traces, ils divisent la vérification de la consistance mémoire en deux étapes. La première vérifie les réordonnancements effectués par les processeurs en $O((N/P)^6)$ entre les deux traces. La deuxième vérifie la consistance des traces au niveau des caches en $O(N \log P)$ grâce aux *timestamps*.

L'ensemble de ces travaux consiste à créer un graphe [GK92 ; MH05] ou une séquence [RHS12] contenant la totalité des accès effectués lors de l'exécution. Ainsi l'occupation mémoire est au moins $O(N)$.

2.3.4 Vérification dynamique

Les techniques présentées jusqu'ici consistent à collecter une trace de l'exécution puis à l'analyser. La trace n'est analysée qu'une fois l'exécution terminée. Il n'est donc pas possible de savoir si l'exécution est erronée avant la fin de celle-ci.

Afin de remédier à cela des techniques dites dynamiques existent. Cette vérification est appelée Vérification Dynamique de la Consistance Mémoire (VDCM). Ces techniques sont principalement utilisées dans une optique de tolérance aux fautes. Les erreurs de consistance sont détectées et éventuellement corrigées.

2.3.4.1 Vérification dynamique intégrée

Cain et Lipasti [CL02] proposent ainsi une méthode pour vérifier la consistance séquentielle d'une exécution sur un système multi-processeurs. Elle consiste à utiliser une horloge logique locale par processeur. Une horloge globale est alors construite comme un vecteur des différents horloges locales. Une valeur de l'horloge globale est associée à chaque case mémoire et à chaque processeur. Les écritures à la mémoire permettent alors de propager les horloges globales depuis un processeur vers une case mémoire. Les lectures propagent les horloges dans le sens opposé. De cette manière il est ensuite possible de vérifier qu'à chaque accès un processeur ou une case mémoire progresse bien par rapport à l'horloge globale.

Meixner et Sorin [MS05 ; MS06] proposent une méthode assez similaire. Leur but est d'implanter un mécanisme de vérification interne et dynamique dans un système multi-processeurs pour le modèle SC et des modèles plus relâchés tels que ceux de l'architecture *SPARC*. À la place d'un vecteur d'horloge, ils utilisent une horloge logique. Le principe reste le même mais il passe mieux à l'échelle. La vérification du MCM est divisée en deux parties effectuées indépendamment. La première consiste à vérifier les accès à partir des caches pour vérifier la séquentialité de ceux-ci. La seconde vérifie si les réordonnancements effectués par les processeurs sont légaux. Des modifications sont apportées aux processeurs, aux caches et aux contrôleurs de mémoires afin qu'ils puissent gérer la propagation et la vérification de l'horloge. Cette vérification impose néanmoins un surcoût au niveau de la bande passante utilisée entre les caches et la mémoire estimé à 25%. La vérification du réordonnancement est la partie la plus complexe, elle nécessite un étage de pipeline supplémentaire associé à un cache spécifique dans les processeurs. Du fait de la taille limitée de ce cache, il faut insérer régulièrement des barrières mémoire dans la SI d'un processeur afin de garantir que le cache ne déborde pas.

Dans le système de Meixner et Sorin, lorsqu'une erreur est détectée, l'accès en question est invalidé en vidant le pipeline processeur. L'accès sera alors retenté. D'autres travaux, ceux de Prvulovic, Zhang et Torrellas [PZT02], sont basés sur un système de *checkpointing*. Ils permettent de reprendre l'exécution à un point antérieur dont l'état a été sauvegardé lorsqu'une erreur est détectée.

2.3.4.2 Vérification dynamique en simulation

Les techniques précédentes peuvent évidemment être intégrées dans les simulateurs. A notre connaissance, elles n'ont d'ailleurs pas été implantées dans des systèmes réels mais uniquement testées en simulation.

D'autres techniques se destinent uniquement à une utilisation en simulation. Ainsi Shacham *et al.* [Sha+08] proposent une technique pour la vérification de la consistance d'une exécution pendant la simulation. Elle consiste à conserver une liste, appelée *scoreboard*, des écritures pouvant être lues, selon le MCM vérifié, pour chaque case mémoire. De cette manière il est possible de vérifier si une lecture a bien lu une valeur issue de cette liste. Le principe de la technique est d'affiner les règles déterminant quelles sont les écritures qui peuvent être lues au fur et à mesure

que les modèles de simulation deviennent plus précis dans le processus de développement d'un SMPMP. Dans le cas le plus simple, toutes les écritures précédentes à la même adresse peuvent être lisibles. Le surcoût dû à la vérification est en $O(CNP^2)$ où C est une constante représentant la taille maximale de la liste. Le coût de génération du maintien à jour de la liste n'est pas précisé et dépend de la complexité des règles utilisées.

Cette technique est intéressante car elle offre une méthode de vérification très simple à mettre en œuvre dans les premières phases de prototypage. De plus la vérification d'un accès est faite dès que celui-ci est effectué : il n'y a pas de latence entre le moment où l'accès est effectué et sa vérification. Néanmoins cette technique est dans la plupart des cas incomplète : si les règles ne correspondent pas exactement au MCM, il peut y avoir des erreurs non détectées.

2.4 Conclusion

Ces différents travaux ont permis de proposer des techniques permettant de réaliser une vérification de la consistance mémoire de l'exécution d'un programme sur un système muti-processeurs à mémoire partagée. La plupart des techniques reposent sur des programmes spécifiques afin de pouvoir obtenir des informations supplémentaires sur l'exécution du programme. En particulier le *read mapping* et l'ordre des écritures sont des informations très importantes puisqu'elles sont nécessaires et suffisantes pour réaliser la vérification de la consistance mémoire en temps linéaire.

Ces informations sont inadaptées aux techniques de vérification dynamique de la consistance mémoire existantes. Les techniques dynamiques décrites nécessitent de décider de la légalité d'un accès pendant sa complétion car elles sont utilisés à des fins de correction d'erreur intégrée dans le système muti-processeurs à mémoire partagée. Ainsi ces techniques sont basées sur l'extraction d'un ordre topologique, utilisant des horloges logiques, permettant cela. L'utilisation d'horloges est néanmoins limitée car elle nécessite de disposer d'ordres totaux initiaux dans les séquences d'instructions, ce qui n'existe que dans les modèles de consistance mémoire très contraints tels Consistance Séquentielle ou *Total Store Order*. Ainsi dès que le réordonnancement devient très relâché, cela devient plus compliqué. Si les écritures ne sont pas atomiques, il faut ainsi une date pour chaque case mémoire spécifique à chaque processeur.

La solution que nous proposons permet de lever ces limitations en s'appuyant sur l'observabilité additionnelle des prototypes virtuels. Nous pouvons en particulier combiner la possibilité d'extraire des informations précises, comme cela est fait avec les horloges, à un traitement unique et non distribué sur l'ensemble du système muti-processeurs à mémoire partagée comme cela se fait dans les techniques intégrées. Par ailleurs, l'intrusivité de notre solution est limitée tout en offrant une vérification complète, dynamique et efficace.

3 Méthode de vérification dynamique de la consistance mémoire

Nous abordons dans ce chapitre le problème de la VDCM. Comme nous l'avons vu dans le chapitre 2, vérifier si une exécution a été conforme à un modèle de consistance peut être très ardu. Il existe par ailleurs plusieurs approches possibles en fonction des informations disponibles sur l'exécution du programme dont on veut vérifier si elle a été conforme à un modèle de consistance.

Du fait de la quantité d'informations mises en jeu, seule une méthode dynamique peut permettre de vérifier la consistance mémoire lors de longues exécutions. Nous présentons dans ce chapitre une méthode adaptée à la VDCM basée sur l'utilisation d'un graphe dynamique.

La première partie décrit la stratégie générale adoptée pour cette méthode de VDCM. Nous y abordons en particulier le problème de la limitation de la taille du graphe ainsi que celui de la détection des violations du MCM. La seconde partie décrit comment le graphe peut être construit en suivant les ordres réels de l'exécution. Cette partie indique ainsi comment gérer les différents types de dépendance rencontrés dans un MCM. La troisième partie donne des exemples complexes de génération du graphe. Nous concluons ensuite cette partie.

Chapitre 3

3.1 Stratégie de vérification dynamique de la consistance mémoire 40

3.1.1 Utilisation d'un graphe 40

3.1.2 Création d'un graphe dynamique 41

3.1.3 Limitation de la taille du graphe 42

3.1.4 Détection des violations du modèle de consistance 43

3.1.5 Bilan sur l'utilisation d'un graphe dynamique 45

3.2 Génération dynamique de graphes partiels 45

3.2.1 Nœuds du graphe 46

3.2.2 Nœuds « *G* » 46

3.2.3 Ordre total selon une séquence 48

3.2.4 Dépendances entre deux catégories d'événements 49

3.2.5 Ordres combinant plusieurs types de dépendance 50

3.2.6 Ordres utilisant des événements supplémentaires 51

3.2.7 Bilan sur les graphes partiels 52

3.3 Génération de l'ensemble du graphe 52

3.3.1 Combinaison pour représenter l'ordre des accès à la mémoire 53

3.3.2 Combinaison pour représenter les dépendances du programme 53

3.3.3 Bilan sur les combinaisons 55

3.4 Conclusion de la génération dynamique du graphe 56

3.1 Stratégie de vérification dynamique de la consistance mémoire

L'utilisation d'un graphe est une méthode très pratique : elle est en particulier générique. Des contraintes de toutes sortes peuvent être modélisées du moment qu'elles peuvent être représentées par des relations d'ordre entre des nœuds. Un nœud représente généralement un accès, mais peut représenter toute chose que l'on jugera nécessaire de représenter.

Pour la détection des violations du MCM, l'utilisation d'un graphe est aussi très simple. Il est possible, comme nous l'avons souligné dans le chapitre 2, de vérifier la présence de cycles, et donc de violations, en temps linéaire. Contrairement à la construction d'un ordre topologique, permettant certes de trier immédiatement les accès, cette technique n'est pas complètement dépendante de l'implantation des communications dans le SMPMP.

3.1.1 Utilisation d'un graphe

3.1.1.1 Principe

L'utilisation d'un graphe pour la VCM consiste à indiquer les différentes dépendances par des arcs. Comme nous l'avons précisé dans le chapitre 2, celles-ci sont regroupées en deux catégories :

L'ordre du programme (noté $\leq_d$) regroupe les dépendances, généralement de données, qu'il y a dans chaque SI. Ces dépendances ne dépendent que du contexte local de chaque SI : chaque SI est indépendante. Les dépendances sont toutefois dynamiques : elles dépendent, bien souvent, de résultats (par exemple le calcul des adresses mémoire) qui sont spécifiques à l'exécution effectuée.

L'ordre de la mémoire (noté $\leq_m$) regroupe les dépendances qui résultent des observations de l'ordre réel des accès à la mémoire tels qu'ils ont été effectués. Comme chaque case mémoire est indépendante, les dépendances ne concernent que les accès à une même case mémoire.

Initialement : $[x] = 0$ et $[y] = 0$

SI_1	SI_2
$W_{1.1}$: [x] := 1	$W_{2.1}$: [y] := 1
$R_{1.2}$: r_1 := [y]	$R_{2.2}$: r_2 := [x]

Résultat : $(r_1, r_2) = (0, 0)$

TABLEAU 3.1 - Exemple d'exécution d'un programme

Le TABLEAU 3.1 montre un exemple d'exécution d'un programme simple. Ce programme est repris d'un exemple du chapitre 1 (cf. section 1.2.4.2). Celui-ci consiste en deux SIs. Les variables $[x]$ et $[y]$ valent initialement 0. L'exécution se déroule de la manière suivante : la SI_1 (respectivement SI_2)) écrit 1 dans la variable $[x]$ (respectivement $[y]$) et lit la valeur 0 dans la variable $[y]$ (respectivement $[x]$).

Cause	Contrainte	Graphe partiel
SI_1	$W_{1.1} <_d R_{1.2}$	W1.1(x:=1) —1→ R1.2(y=0)
SI_2	$W_{2.1} <_d R_{2.2}$	W2.1(y:=1) —2→ R2.2(x=0)
$[x]$	$R_{2.2} <_m W_{1.1}$	R2.2(x=0) —x→ W1.1(x:=1)
$[y]$	$R_{1.2} <_m W_{2.1}$	R1.2(y=0) —y→ W2.1(y:=1)

TABLEAU 3.2 - Dépendances induites de l'exécution en SC

Le TABLEAU 3.2 donne les différentes dépendances dans le modèle SC pour cette exécution et le graphe partiel résultant de chacune d'entre elles. Quatre ordres sont exprimés, l'ordre de la SI_1, celui de la SI_2, celui de la variable $[x]$ et celui de $[y]$.

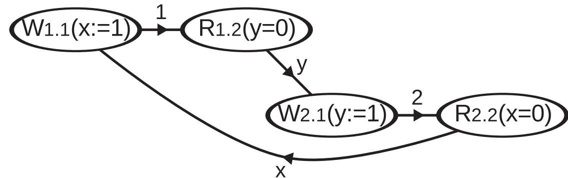


FIGURE 3.1 - Graphe complet de l'exemple d'exécution SC

Le graphe complet résultant de cette exécution est représenté dans la FIGURE 3.1. Une étiquette est attachée à chaque arc afin d'en indiquer la cause (« 1 » pour SI_1, « x » pour $[x]$, etc.). Les quatre arcs forment un cycle indiquant que le MCM, SC dans cet exemple, est violé conformément à ce que nous avions vu dans le chapitre 1.

3.1.1.2 Notations

Avant de continuer, nous définissons ici quelques notations ainsi que le vocabulaire que nous allons employer par la suite.

Nous notons « $N_1 \rightarrow N_2$ », le fait qu'il y ait un arc depuis le nœud N_1 vers le nœud N_2. Lorsqu'il est question de plusieurs arcs formant une chaîne, nous le notons par exemple « $N_1 \rightarrow N_2 \rightarrow N_3$ » pour représenter les deux arcs « $N_1 \rightarrow N_2$ » et « $N_2 \rightarrow N_3$ ».

Définition 6 (Prédécesseur)**.** Lorsqu'il y a un arc « $N_1 \rightarrow N_2$ », nous disons que N_1 est un prédécesseur de N_2. Dans le cas où « $N_1 \rightarrow N_2 \rightarrow N_3$ », N_2 est un prédécesseur de N_3 mais pas N_1. Un nœud est un prédécesseur d'un autre uniquement s'il y a un arc entre les deux. Nous utilisons l'expression explicite « prédécesseur indirect » lorsque nous voulons parler d'un nœud N_1 dont il existe un chemin formé de plusieurs arcs vers un nœud N_3.

Définition 7 (Successeur)**.** Si N_1 est un prédécesseur de N_2, nous disons inversement que N_2 est un successeur de N_1.

3.1.2 Création d'un graphe dynamique

Les dépendances permettant de construire le graphe sont issues des deux catégories : l'ordre du programme et l'ordre de la mémoire. L'ordre du programme peut être divisé en autant d'ordres qu'il y a de processeurs (ou de SIs) car ils sont indépendants. Chaque ordre partiel est directement déduit de l'ordre de la SI correspondante. De même, l'ordre de la mémoire peut être divisé en autant d'ordres qu'il y a de cases mémoire.

Notre stratégie consiste à construire ces différents ordres indépendants au fur et à mesure que l'exécution s'effectue. Les nœuds sont ajoutés au graphe uniquement lorsqu'ils deviennent nécessaires : c'est à dire lorsqu'ils deviennent partie prenante d'au moins un des ordres. Les ordres de chaque SI et case mémoire sont construits indépendamment les uns des autres. Ils ne sont reliés que par leurs nœuds qui peuvent éventuellement être présents dans plusieurs ordres. Un nœud représentant un accès atomique est ainsi présent dans l'ordre de la SI qui l'a généré et dans celui de la case mémoire accédée. Lorsqu'un nœud est traité et placé dans un ordre, celui d'une SI ou d'une case mémoire, les arcs correspondants aux dépendances de cet ordre sont ajoutés.

La FIGURE 3.2 illustre le principe général utilisé. Elle représente une exécution sur un SMPMP contenant deux processeurs qui exécutent deux SIs (SI_1 et SI_2) et qui utilisent deux cases mémoire ($[x]$ et $[y]$). Les ordres de chaque SI et de chaque case mémoire sont représentés. Pour une raison de

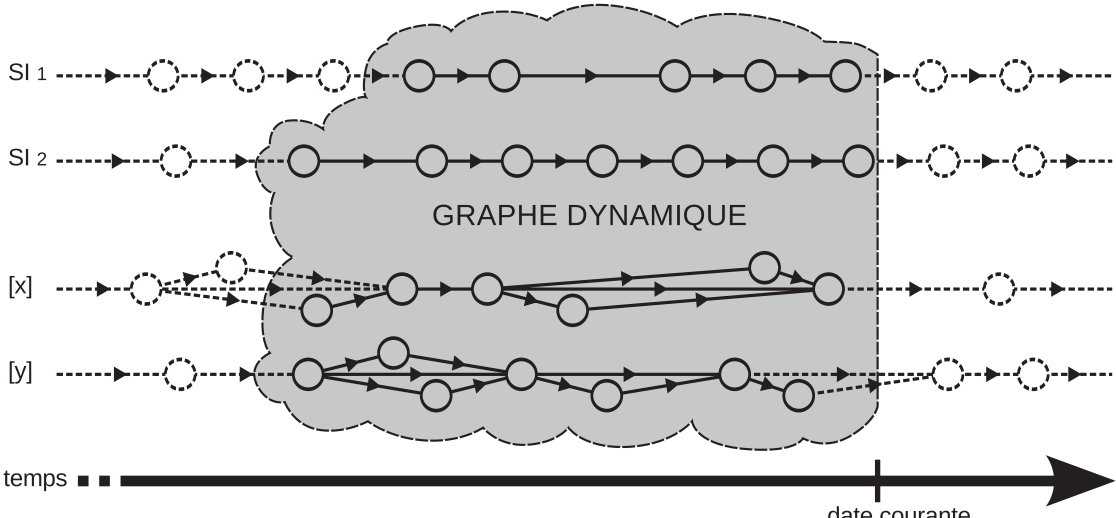


FIGURE 3.2 - Graphe dynamique d'une exécution

clarté, les nœuds sont tous distincts mais chaque nœud est en pratique à la fois dans l'ordre d'une SI et dans l'ordre d'une case mémoire : les différents ordres sont entrelacés. Dans la figure, le temps évolue de gauche à droite. L'échelle de temps est néanmoins virtuelle : elle illustre les moments où les nœuds sont traités dans leurs différents ordres. Elle ne représente que grossièrement l'évolution du temps lors de l'exécution puisqu'il n'est d'ailleurs pas garanti de pouvoir en extraire un temps global.

Dans la figure, une partie du graphe complet de l'exécution est représentée : celui-ci correspond au graphe contenant l'ensemble des nœuds nécessaires pour représenter l'exécution de son début à sa fin. On se place néanmoins à une certaine date et la zone grisée correspond au graphe dynamique à cette date. Celui-ci n'est constitué que d'une petite partie du graphe complet. Les nœuds à droite de la zone grisée (les nœuds « futurs ») n'ont pas encore été ajoutés au graphe.

Les nœuds à gauche (les nœuds « passés ») de cette zone ont par contre déjà été supprimés. Le début du graphe dynamique (c.-à-d. son bord gauche) n'est pas net car la suppression des nœuds ne se fait pas en fonction de leur date (il n'y d'ailleurs pas de notion de date pour un nœud). Les nœuds du graphe sont en effet supprimés lorsqu'ils ne sont plus nécessaires pour la VCM. Dans une solution classique, c'est à dire qui construirait le graphe complet pour ensuite y détecter les cycles, le graphe remonterait jusqu'au début de l'exécution : aucun nœud ne serait supprimé. Mais cette méthode ne passe pas à l'échelle à cause de l'explosion de la taille du graphe.

3.1.3 Limitation de la taille du graphe

Afin de limiter la taille du graphe, les nœuds doivent être supprimés lorsqu'ils ne sont plus nécessaires pour la VCM. Dans notre contexte un nœud devient inutile quand il est certain qu'il ne sera jamais impliqué dans un cycle : c'est en effet la condition, nécessaire et suffisante, d'une violation du MCM.

Dans un graphe orienté, une condition simple pour qu'un nœud ne fasse pas partie d'un cycle est qu'il n'ait aucun prédécesseur direct ou indirect. Cela découle du fait que dans un cycle, chaque nœud du cycle a un prédécesseur dans le cycle. En pratique on peut s'intéresser uniquement aux prédécesseurs (directs), puisque sans prédécesseur direct il n'y a pas de prédécesseurs indirects. Ainsi lorsque le graphe est dynamique, cela se traduit donc par les deux conditions suivantes :

1. le nœud n'a actuellement aucun prédécesseur (direct), et
2. le nœud n'en aura pas d'autres par la suite.

En supprimant les nœuds lorsque ces deux conditions sont remplies, les nœuds sont supprimés de proche en proche en partant du début du graphe : lorsqu'un nœud est supprimé, cela ouvre la possibilité de supprimer ses successeurs.

Des conditions similaires sur les successeurs garantissent aussi l'absence de cycle contenant le nœud étudié. Mais cela signifierait que les nœuds sont supprimés de proche en proche en partant de la fin de graphe qui correspond à la fin de l'exécution. Comme le graphe est construit au fur et à mesure que l'exécution se produit, les ordres sont donc traités dans l'ordre croissant. La fin du graphe n'est vraisemblablement construite qu'une fois l'exécution terminée, ce qui rend ces conditions inapplicables.

La condition sur les futurs prédécesseurs est beaucoup plus raisonnable même si elle n'est pas forcément simple à valider dans certains cas. Par exemple, une écriture W_2 succède à toutes les lectures de l'écriture précédente W_1 à la même case mémoire ; ces lectures précèdent donc l'écritureW_2 et il faudra donc s'assurer qu'aucune lecture de l'écriture précédente W_1 n'aura lieu avant de pouvoir supprimer l'écriture W_2. Ce problème sera vu plus en détail dans le chapitre 4.

Les deux conditions n'impliquant pas les successeurs ou futurs successeurs d'un nœud, il est donc tout à fait possible de supprimer un nœud alors que certains de ses successeurs n'ont pas encore été ajoutés. Une implantation de ce mécanisme devra donc porter une attention particulière à éviter que cela ne se produise ou ne pose de problèmes. Par exemple, pour une implantation utilisant des pointeurs vers des objets représentant des nœuds, une solution serait d'invalider les pointeurs restant vers un nœud lorsqu'il est supprimé.

3.1.4 Détection des violations du modèle de consistance

La détection des violations du MCM lorsqu'un graphe est utilisé se fait en recherchant si il y a des cycles dans le graphe. L'Algorithme 3.1 permet de détecter si un graphe contient au moins un cycle dans un graphe en faisant un parcours en profondeur du graphe. Le principe est de parcourir un sous-graphe jusqu'à arriver à la fin. Les nœuds visités lors du parcours sont marqués de telle sorte qu'on détecte si un nœud parcouru est visité à nouveau. Un sous-graphe parcouru exempt de cycle est aussi marqué afin que ce sous-graphe ne soit parcouru qu'une seule fois. Sa complexité est linéaire ($O(N_n + N_a)$) où N_n est le nombre de nœuds et N_a le nombre d'arcs.

3.1.4.1 Besoin d'une détection dynamique

Cet algorithme est très adapté pour détecter les cycles une fois que l'exécution est terminée, afin de vérifier que le graphe final est acyclique. Cette détection finale est nécessaire mais ne suffit néanmoins pas dans notre cas. Un problème se pose effectivement si un cycle s'est formé durant l'exécution. Il sera détecté par la détection finale, mais la taille du graphe aura probablement explosé.

Une fois un cycle formé, chacun des nœuds qui le composent ont un prédécesseur qui fait partie du cycle. Ces nœuds ne peuvent pas être supprimés car les conditions exprimées ci-dessus ne seront jamais vraies. D'une certaine manière, les nœuds du cycle se protègent mutuellement contre une suppression. Le problème majeur causé par ce phénomène est que l'ensemble des successeurs directs et indirects de ces nœuds ne pourront jamais être supprimés. Il est fort probable que cela concerne, à terme, les nœuds de tous les accès d'une ou plusieurs SIs : une barrière mémoire émise par une SI après un accès dont le nœud est dans le cycle imposerait une dépendance vers ce nœud et donc un blocage de tous les accès suivants. Par le biais des accès mémoire, le blocage peut éventuellement atteindre l'ensemble des SIs. La taille du graphe augmenterait donc inexorablement jusqu'à la fin de l'exécution.

```
1:  fonction DétectionCycle(graphe G)
2:      pour chaque nœud N ∈ G faire
3:          N.état := non-visité                              ▷ Initialisation
4:      fin pour
5:      pour chaque nœud N ∈ G faire
6:          si VisiteSousGraphe(S) = KO alors                 ▷ Un cycle a été détecté
7:              return KO
8:          fin si
9:      fin pour
10:     return OK                                             ▷ Aucun cycle dans G
11: fin fonction
12: fonction VisisteSousGraphe(nœud N)
13:     si N.état = visité alors                              ▷ Le sous-graphe a déjà été testé et
14:         return OK                                           ne contient aucun cycle
15:     sinon si N.état = en-cours alors                      ▷ Un cycle est détecté
16:         return KO
17:     fin si
18:     N.état := en-cours
19:     pour chaque S ∈ {successeurs de N} faire
20:         si VisiteSousGraphe(S) = KO alors
21:             return KO                                     ▷ Le sous-graphe contient un cycle
22:         fin si
23:     fin pour
24:     N.état := visité                                      ▷ Le sous-graphe n'a aucun cycle
25:     return OK
26: fin fonction
```

Algorithme 3.1 - Algorithme de détection des cycles dans un graphe (recherche en profondeur [Tar71])

3.1.4.2 Détection dynamique « moindre effort »

Les cycles doivent donc être détectés tout au long de l'exécution. Cela ne peut malheureusement être fait à chaque fois qu'un arc est ajouté car cela coûterait beaucoup trop cher. Nous proposons de faire une détection que l'on pourrait qualifier de partiellement dynamique. Celle-ci consiste à utiliser l'Algorithme 3.1 lorsque certaines conditions sont remplies afin de garantir que le coût global de l'algorithme est toujours linéaire.

Il faut pour cela utiliser une métrique, notée n qui représente la taille du graphe. Dans le cas général, elle peut être égale au nombre de nœuds plus le nombre d'arcs. Si le nombre d'arcs évolue linéairement avec le nombre de nœuds ($N_a = O(N_n)$), comme cela est souvent le cas, n peut être simplement le nombre de nœuds. A chaque fois que n dépasse un seuil, la détection est lancée : cela coûte $O(n)$. Si aucun cycle n'est détecté, il suffit alors de doubler le seuil pour éviter de lancer la détection trop souvent.

En se plaçant dans le pire des cas, aucun nœud n'est jamais supprimé du graphe. Ainsi, si on commence avec un seuil réglé à 1, il y aura une détection lorsque n atteint 1, puis 2, puis 4, puis 8, etc. . En considérant que N est la valeur finale, et donc maximale, de n, le coût global de ces détections est donc :

$$\text{Coût_seuil} = O\left(\sum_{n=0}^{\lfloor \log_2(N) \rfloor} 2^n\right) = O\left(2^{1+\lfloor \log_2(N) \rfloor} - 1\right) = O(2N) = O(N)\ .$$

Il faut ajouter à cela la détection finale qui coûte aussi $O(N)$. On obtient ainsi un coût global du pire cas de $O(N)$ pour la détection. Cette politique de détection ne permet pas de détecter un cycle dès sa formation, puisqu'il faut attendre que la taille du graphe dépasse le seuil courant. Cela n'est pas un problème dans notre cas, puisque ce système de VCM n'a pas vocation à être utilisé dans un mécanisme de détection et de correction d'erreurs.

Cette technique du moindre effort est très efficace si la taille du graphe se stabilise à une faible valeur lors d'une exécution ne violant pas le MCM. Dans ce cas, une fois la taille stabilisée atteinte, aucune détection intermédiaire ne sera lancée jusqu'à la fin de l'exécution à moins qu'une violation du MCM vienne déclencher un grossissement du graphe.

3.1.4.3 Détection dynamique « régulière »

Il est néanmoins possible d'améliorer la réactivité de la détection. Lorsque des éléments sont supprimés du graphe, le seuil peut être abaissé du nombre d'éléments supprimés. Ainsi même si la taille du graphe est stable, car autant d'éléments sont supprimés et ajoutés, le seuil baisse et va donc finir par passer sous la taille du graphe. Cela ne modifie en rien le coût du pire cas de la technique puisque dans ce cas aucun élément n'est jamais supprimé. Cette modification entraîne un surcoût par rapport à la détection « moindre effort », mais reste linéaire. En pratique il est possible de moduler le taux de réduction du seuil pour diminuer le coût de la détection (par exemple, une réduction du seuil de 1 toutes les 2 suppressions d'éléments du graphe). L'étude précise de la complexité de cette méthode de détection des cycles est donnée dans l'annexe A.

L'utilisation de cette technique est par ailleurs nécessaire si on souhaite s'assurer qu'un cycle ne peut rester indéfiniment dans le graphe. Dans ce cas le cycle ne sera détecté, par la méthode « moindre effort », qu'à la détection terminale effectuée à la fin de l'exécution. Ce scénario n'est néanmoins possible que si ce cycle n'entraîne pas de grossissement du graphe : cela est très improbable puisqu'il faudrait que les nœuds du cycle aient un nombre limité de successeurs directs et indirects. Typiquement, si des barrières mémoire sont régulièrement exécutées dans les SIs, cela ne peut pas se produire.

3.1.5 Bilan sur l'utilisation d'un graphe dynamique

La stratégie présentée dans cette partie permet de vérifier dynamiquement la consistance mémoire. Le coût de la détection des violations est linéaire par rapport à la taille maximale du graphe, ce qui permet d'utiliser cette stratégie sur des exécutions longues. La vérification est complète dans la mesure où l'ensemble des contraintes du MCM sont mises en place dans les ordres des SIs et des cases mémoire.

3.2 Génération dynamique de graphes partiels

La méthode décrite permet de détecter en temps linéaire, par rapport à la taille du graphe, les violations du MCM. Le graphe doit donc être construit de manière à limiter cette taille. Nous cherchons en particulier à ce qu'elle soit une fonction linéaire des opérations impliquées dans le MCM. Ajouter un nœud ou un arc coûte $O(1)$ en complexité algorithmique et en espace mémoire : il est donc de toute manière nécessaire que le graphe évolue linéairement pour pouvoir passer à l'échelle.

Dans cette partie, nous décrivons la procédure à suivre pour construire le graphe. Nous décomposons sa construction en plusieurs parties indépendantes que nous appelons des « graphes partiels ». Un graphe partiel correspond, dans la plupart des cas, à la mise en place d'un ordre partiel sur un sous-ensemble des nœuds du graphe. Chaque graphe partiel se construit à partir d'une séquence d'événements. Les événements comprennent, mais pas seulement, les accès à la

mémoire. Tout événement ayant un impact sur le MCM peut être utilisé. Une séquence permettant de construire un graphe partiel est, en pratique, extraite soit d'une SI d'un processeur soit d'une séquence d'accès à une case mémoire. Nous expliquons ainsi comment construire différents graphes partiels de manière dynamique c'est à dire au fur et à mesure que leurs séquences d'événements se produisent.

Avant d'aborder la construction de ces graphes partiels, les nœuds et principes généraux utilisés pour la construction des graphes partiels sont tout d'abord précisés.

3.2.1 Nœuds du graphe

Les nœuds du graphe peuvent représenter différentes choses. La plupart des méthodes de VCM n'utilisent que deux types de nœuds lorsque les écritures sont atomiques : ceux représentant les lectures et ceux représentant les écritures. S'il y a des écritures non atomiques, celles-ci sont représentées respectivement par P nœuds (P est le nombre de processeurs). Ainsi il y a dans ce cas $P + 1$ catégories de nœuds : les lectures et les P parties des écritures.

Dans notre cas nous utilisons des nœuds supplémentaires. Un nœud peut correspondre par exemple à une barrière mémoire ou être simplement utilisé pour des raisons pratiques. Dans cette partie, nous distinguons trois catégories principales de nœuds :

- les nœuds R (pour *read*), correspondant aux lectures ;
- les nœuds W (pour *write*), correspondant aux écritures ; et
- les nœuds G (pour *gather*), décrits ci-dessous.

3.2.2 Nœuds « G »

Nous utilisons les nœuds G comme nœuds intermédiaires pour représenter de nombreuses contraintes. Il est en théorie possible de se limiter aux nœuds R et W, mais cela n'est pas efficace. Dans de nombreux cas il est plus pratique et moins coûteux d'utiliser des nœuds supplémentaires.

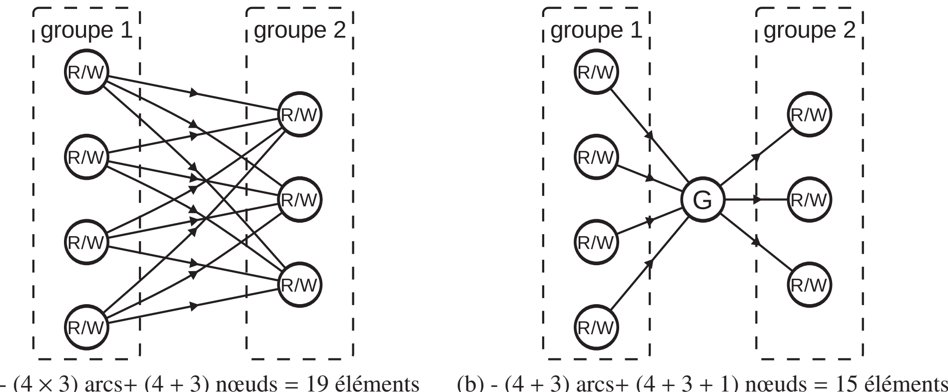
3.2.2.1 Réduction de la taille du graphe

Ils sont en particulier nécessaires pour limiter la taille du graphe, ce qui limite indirectement la complexité de la détection des cycles. Cela se manifeste lorsqu'il faut indiquer des contraintes d'ordre entre deux ensembles de nœuds, c'est à dire entre chaque couple d'un nœud du premier ensemble et d'un nœud du second. Si les ensembles sont de tailles n_1 et n_2, il faut mettre $n_1 \times n_2$ contraintes qui se caractérisent par autant d'arcs. Utiliser un nœud intermédiaire permet de réduire le nombre d'arcs à $n_1 + n_2$.

Cela est illustré dans la FIGURE 3.3. Les deux graphes sont équivalents car la relation représentée par les arcs est transitive. Cette utilisation est capitale, puisque si les deux ensembles n'ont pas une taille limitée, ne pas utiliser de nœud intermédiaire entraine l'utilisation d'un nombre quadratique d'arcs par rapport au nombre de nœuds. La taille du graphe n'aurait donc plus une évolution linéaire.

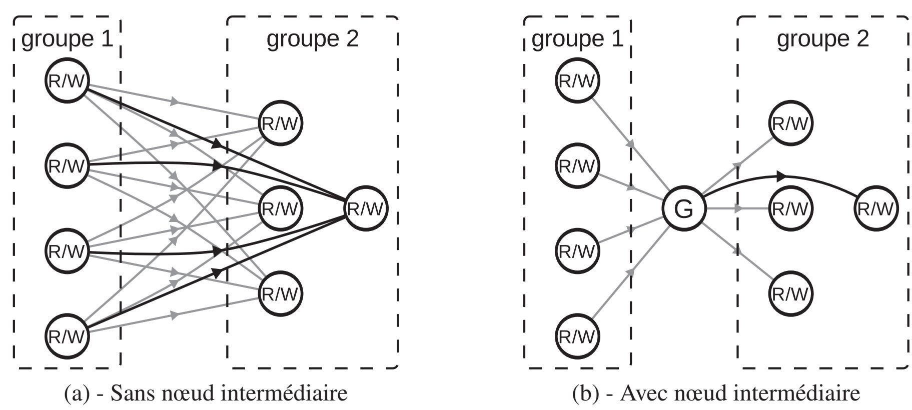
3.2.2.2 Gestion d'ensembles de nœuds

De manière générale nous utilisons un nœud intermédiaire dès qu'un nombre indéfini de nœuds vont chacun avoir un arc vers un ou plusieurs nœuds identiques. Comme la construction du graphe est dynamique, les nœuds sont traités un à un et ils ne sont vraisemblablement pas tous présents dans le graphe au moment où le premier nœud est traité. Ainsi, pour pouvoir ajouter les arcs lorsque les nœuds suivants seront traités, il faut garder une trace de tous les nœuds impliqués. En utilisant un nœud intermédiaire, seul celui-ci doit être tracé.



(a) - (4×3) arcs+ $(4 + 3)$ nœuds = 19 éléments (b) - $(4 + 3)$ arcs+ $(4 + 3 + 1)$ nœuds = 15 éléments

FIGURE 3.3 - Graphes équivalents illustrant le gain de taille obtenu avec un nœud intermédiaire



(a) - Sans nœud intermédiaire (b) - Avec nœud intermédiaire

FIGURE 3.4 - Ajout d'un nœud dans un groupe

La FIGURE 3.4 illustre ainsi l'ajout d'un nœud dans le deuxième groupe des graphes de la FIGURE 3.3. Dans la FIGURE 3.4a, pour ajouter un nœud dans l'ensemble de droite et mettre les contraintes avec l'ensemble des nœuds de gauche, il faut garder une trace des quatre nœuds de gauche et mettre un arc entre chacun de ces quatre nœuds et le nouveau. Alors qu'en utilisant un nœud intermédiaire, comme dans la FIGURE 3.4b, il suffit de garder une trace du nœud intermédiaire et de mettre un arc entre ce nœud et le nouveau nœud. On pourrait ajouter de manière similaire un nœud dans le groupe de gauche. Ainsi, au lieu de garder la trace de deux ensembles de nœuds, il suffit de garder la trace d'un seul nœud.

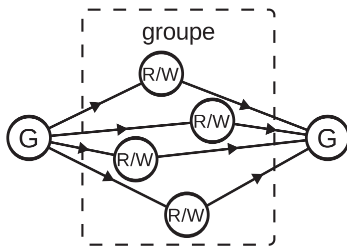


FIGURE 3.5 - Encadrement d'un ensemble de nœuds avec deux nœuds supplémentaires

De manière générale, on peut représenter un groupe de nœuds en utilisant seulement deux nœuds G. Les deux nœuds encadrent l'ensemble des nœuds du groupe, comme cela est illustré dans la FIGURE 3.5. Des nœuds peuvent être facilement ajoutés avant, après ou même dans le groupe

en gardant uniquement la trace des nœuds G. Cela permet donc de leur ajouter très facilement des contraintes communes.

Dans la suite de cette partie, nous détaillons comment créer différents graphes partiels nécessaires à la construction du graphe dynamique. Pour certains d'entre eux, nous sommes amenés à utiliser des nœuds G afin de conserver la trace d'un nombre limité de nœuds.

3.2.3 Ordre total selon une séquence

Le cas de l'ordre total est le plus simple à gérer. Pour représenter un ordre total selon une séquence, il suffit en effet de traiter la séquence dans l'ordre et d'ajouter, lors du traitement d'un événement de la séquence, un arc depuis le précédent nœud vers le nouveau.

Ce cas est illustré dans la FIGURE 3.6. Nous utilisons des graphiques similaires dans toute la suite de ce chapitre. L'axe estampillé « temps » en bas de la figure représente le traitement de la séquence. Dans la figure, chaque événement de la séquence est soit une lecture soit une écriture. Les événements sont numérotés de 1 à 7. Le graphe généré est représenté dans la partie haute de la figure et les nœuds sont placés à la verticale des événements auxquels ils correspondent. Les arcs sont étiquetés avec un numéro qui indique au cours de quel événement celui-ci a été ajouté dans le graphe : l'arc « 2 » entre l'écriture « 1 » et la lecture « 2 » est par exemple ajouté pendant le traitement de la lecture « 2 ».

L'axe « REF » (pour *référence*) correspond au fait qu'il faille conserver la trace du nœud du dernier événement pour ajouter l'arc vers le nœud suivant. Cette référence est mise à jour à chaque traitement d'un événement de la séquence pour garder la trace de celui-ci. Ces mises à jour sont représentées par les flèches pointillées allant de l'axe vers les nœuds : le point de départ de la flèche indique le moment où la mise à jour est faite par rapport au traitement des événements ; la pointe de la flèche indique le nouveau nœud qui est référencé.

Cette figure représente par exemple la mise en place des dépendances entre les accès d'une SI dans le modèle SC ou entre les accès à une même case mémoire pour laquelle l'ordre des lectures importe (par exemple dans un périphérique).

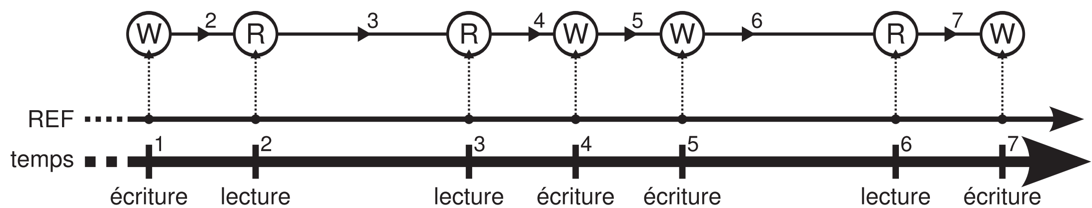


FIGURE 3.6 - Construction de l'ordre total

Un ordre total peut aussi être mis en place pour une séquence réduite. Par exemple, si un MCM impose des contraintes de type RAR ou WAW, il faut appliquer ce principe à la séquence contenant seulement les lectures ou les écritures d'une SI. La FIGURE 3.7 représente ainsi la génération de ces deux types de contraintes. Il faut alors utiliser une référence pour chaque ordre total : une pour les lectures (REF_R) et une pour les écritures (REF_W).

Pour chaque ordre total mis en place, il faut ajouter $n - 1$ arcs quand la séquence comprend n événements.

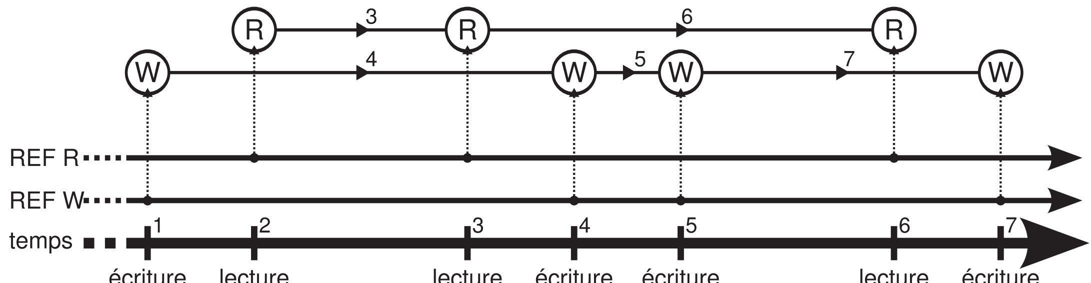


FIGURE 3.7 - Construction des ordres RAR et WAW

3.2.4 Dépendances entre deux catégories d'événements

3.2.4.1 Dépendances simples

Quand les dépendances concernent des catégories d'événements différents (par exemple les dépendances RAW ou WAR), le principe de génération du graphe est différent. Nous illustrons ce cas de figure avec les dépendances RAW, mais n'importe quel type de dépendances entre deux catégories d'événements peut être traité de la même manière et en particulier le cas des dépendances contraires WAR. Dans les dépendances RAW, il n'y a aucune dépendance entre deux lectures ou deux écritures. Cela nous oblige à utiliser des nœuds supplémentaires car il faut gérer des ensembles d'écritures et de lectures. Il faut en effet indiquer, à tout moment dans la séquence, une contrainte entre l'ensemble des écritures passées et l'ensemble des lectures futures.

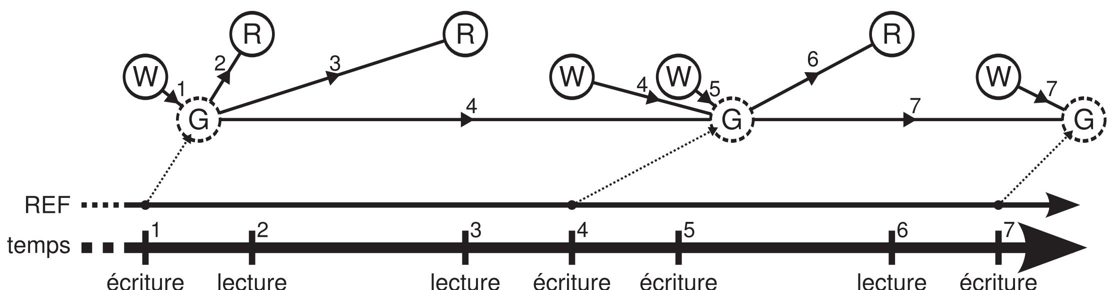


FIGURE 3.8 - Construction de l'ordre RAW

La FIGURE 3.8 illustre la génération d'un tel graphe. Une référence vers un nœud *G* est conservée, celle-ci représente l'ensemble des écritures passées. Les nœuds *R* et *W* sont toujours positionnés à la verticale de leur événement sur l'axe de la séquence. Les nœuds *G* sont placés de manière à ce que les arcs soient tous orientés de gauche à droite. Ces derniers sont néanmoins ajoutés dans le graphe aux moments où ils sont référencés, comme indiqués par le départ des flèches pointillées. Lors du traitement d'une lecture, un arc est ajouté depuis le nœud référencé. Un nouveau nœud *G* est ajouté à chaque fois qu'une écriture suivant une lecture est traitée. Dans ce cas, un arc est ajouté entre le nœud *G* précédent de manière à ce que le nouveau nœud *G* représente les écritures déjà passées. Lors du traitement de chaque écriture, un arc est de plus ajouté vers le nœud référencé.

3.2.4.2 Variantes

Il est possible de faire des variantes de ces dépendances. Par exemple, on peut imaginer séparer les écritures en deux sous-catégories d'écritures : les écritures totales et les écritures partielles. On adapte alors les règles de dépendance : une lecture ne dépend pas de toutes les écritures passées, mais seulement des écritures passées en remontant jusqu'à la dernière écriture totale.

Ce genre de dépendance représente par exemple les dépendances de données à travers un registre : une lecture de ce registre ne dépend pas des données qui ont été écrites dedans auparavant si celui-ci vient d'être réinitialisé. Les modèles qui relâchent l'ordre des accès tels que SPARC RMO ou ceux de l'architecture ARM utilisent en particulier ce genre de contraintes. On notera que dans ce cas une lecture correspond à une lecture d'un registre et donc généralement à une écriture en mémoire (instruction de type *store*) et inversement. Néanmoins dans le cadre d'un registre contenant une adresse mémoire, les instructions de type *load* ou *store* sont susceptibles de lire et utiliser celle-ci. Ce qui correspond à une lecture ou une écriture doit être déterminé en fonction du contexte.

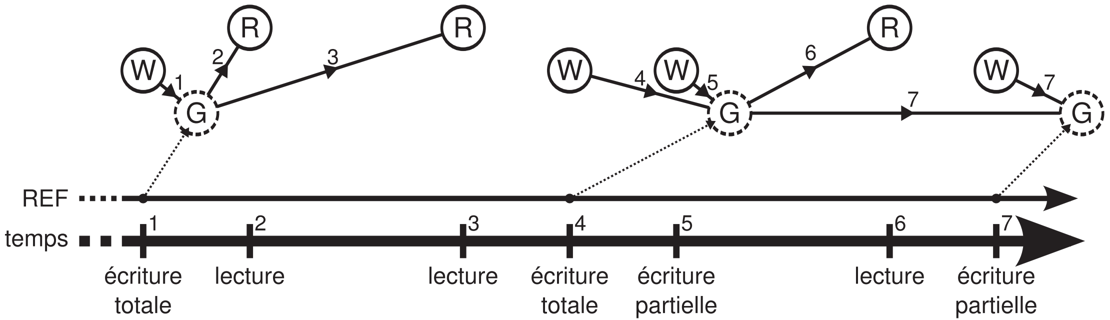


FIGURE 3.9 - Construction d'une variante de l'ordre RAW

La FIGURE 3.9 illustre la génération de cette variante. On peut voir qu'il n'y a plus d'un ordre total entre les nœuds *G* : la séquence de nœud *G* est interrompue lorsqu'une écriture totale a lieu.

Pour gérer ce genre de dépendances il faut au maximum ajouter un nœud *G* pour chaque écriture. Concernant les arcs, il y a un arc par accès (lecture ou écriture) et au plus un arc par nœud *G*. Ainsi la taille du graphe reste linéaire de la taille de la séquence : il y a au plus $2 \times n$ nœuds et $2 \times n$ arcs si la séquence contient n accès.

3.2.5 Ordres combinant plusieurs types de dépendance

Il bien sûr possible de combiner plusieurs types de dépendance. Cela peut être fait en construisant un graphe partiel de manière indépendante pour chaque type de dépendance. Néanmoins il est bien souvent plus efficace de les combiner dans un unique graphe partiel : moins de nœuds supplémentaires et moins d'arcs sont généralement nécessaires.

3.2.5.1 Dépendances « RAR + WAR »

Par exemple, dans le modèle SPARC PSO, les écritures peuvent être réordonnées entre elles et les lectures peuvent être réordonnées avec les écritures précédentes. Ainsi seules les dépendances RAR et WAR sont maintenues.

La FIGURE 3.10 illustre la génération conjointe de ces deux types de dépendance. On peut voir que contrairement à la génération seule des dépendances entre deux types différents (cf. la section 3.2.4), aucun nœud supplémentaire n'est effectivement nécessaire. Du fait de l'ordre total entre les lectures, la dernière lecture peut en effet être utilisée pour indiquer une dépendance envers toutes les lectures passées : il n'est pas nécessaire d'utiliser des nœuds *G* pour représenter l'ensemble des lectures passées. Le nœud lecture en pointillé correspond au nœud référencé initialement au début de la portion de séquence visible dans la figure.

Un seul arc par nœud est nécessaire pour représenter ces dépendances.

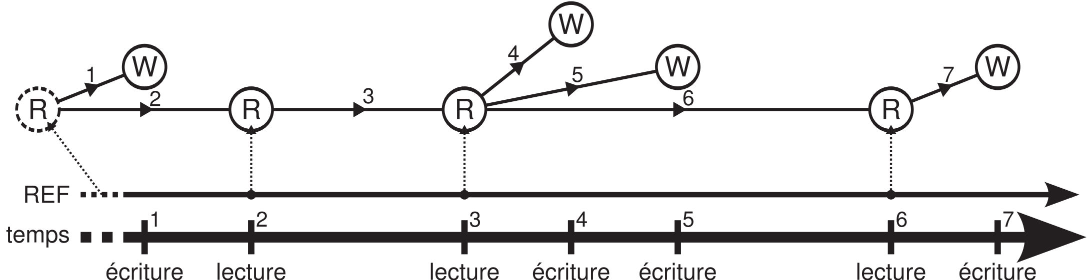


FIGURE 3.10 - Construction de l'ordre « RAR+WAR »

3.2.5.2 Dépendances « RAW + WAW + WAR »

Un cas très important est le graphe partiel de l'ordre correspondant aux accès à une case mémoire. Dans celui-ci seule la dépendance RAR n'est pas imposée : il n y a aucun impact à inverser l'ordre de plusieurs accès en lecture puisqu'une lecture ne modifie pas la valeur de la case mémoire. De manière générale, l'ordre entre plusieurs lectures est d'ailleurs inconnu, seul l'ordre entre les écritures et la position des lectures parmi les écritures sont connus.

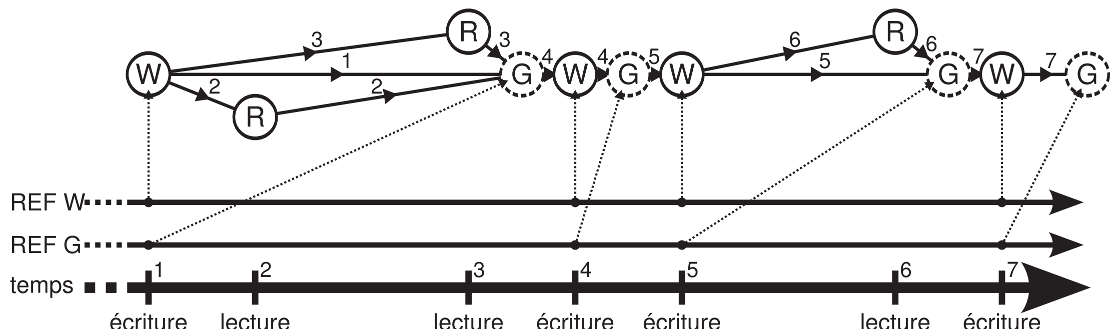


FIGURE 3.11 - Construction de l'ordre des accès à une case mémoire « RAW + WAW + WAR »

Ce cas est illustré dans la FIGURE 3.11. Pour représenter cet ordre, nous utilisons un nœud *G* avec chaque écriture. Celui-ci permet de représenter l'ensemble des accès passés jusqu'à l'écriture suivante, le nœud *G* suit toutes les lectures à l'écriture précédente. Il faut donc dans ce cas conserver deux références : une vers la dernière écriture et une vers le nœud *G* correspondant. Ces deux nœuds encadrent les lectures à la dernière écriture.

Quand une écriture est traitée, celle-ci est mise après (à l'aide d'un arc) le dernier nœud *G* référencé et un nouveau nœud *G* est ajouté après la nouvelle écriture. Lorsqu'une lecture est traitée, elle est placée entre le nœud *W* et le nœud *G* grâce à deux arcs.

La taille du graphe généré est bien linéaire en fonction du nombre d'accès : il y a un nœud supplémentaire et deux arcs par écriture ainsi que deux arcs par lecture.

3.2.6 Ordres utilisant des événements supplémentaires

La construction de graphes partiels correspondant à certains ordres nécessite de traiter des événements additionnels autres que les accès : ils ne dépendent pas uniquement des lectures et des écritures.

La FIGURE 3.12 illustre un exemple où des barrières sont traitées parmi les écritures et les lectures. Le graphe généré n'impose aucune dépendance à part qu'un accès ne peut être réordonné avec une barrière.

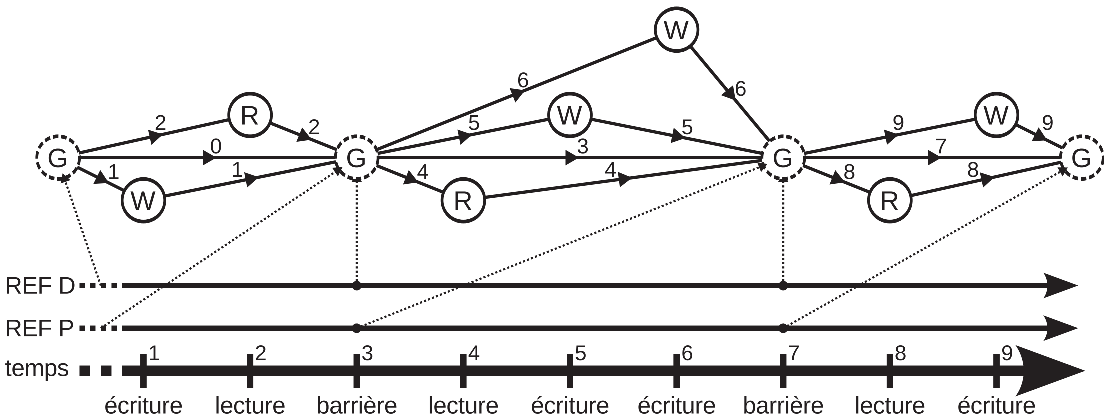


FIGURE 3.12 - Représentation des contraintes d'ordre d'une barrière mémoire

Pour gérer ce genre d'événements, il faut deux références : une pour référencer la dernière barrière (REF_D) et une pour référencer la prochaine (REF_P). Ces références gardent la trace de deux nœuds *G* qui encadrent l'ensemble des accès émis jusqu'à la prochaine barrière. Chaque accès est ainsi inséré dans le graphe entre les deux nœuds référencés. Lorsqu'une barrière est traitée, un nouveau nœud *G* est ajouté et les références sont mises à jour.

Dans ce cas, la taille du graphe est linéaire en fonction de la taille de la séquence : il y a un nœud par événement et au maximum deux arcs par nœuds.

3.2.7 Bilan sur les graphes partiels

Nous avons présenté dans cette partie le principe de génération dynamique du graphe à partir de graphes partiels. Chaque graphe partiel permet de modéliser une partie des contraintes d'ordre qu'il faut mettre en place. Nous l'avons illustré à partir d'exemples utilisant des dépendances classiques (RAW par exemple). Ils ne sont néanmoins pas exhaustifs : d'autres cas existent. Les graphes partiels qu'il faut construire dépendent entièrement du MCM que l'on veut tester.

Nous nous sommes en particulier limités, dans ces exemples, à la gestion des accès classiques (lectures et écritures). D'autres types d'accès peuvent être nécessaires. Par exemple si l'on veut vérifier que les instructions sont correctement chargées, il sera sûrement nécessaire d'introduire un troisième accès pour représenter leur chargement. Le chargement des instructions n'est en effet généralement pas traité comme les lectures des données par les MCM.

Grâce à l'utilisation des nœuds supplémentaires, notés *G*, la taille des graphes partiels correspondant à chaque ordre est linéaire en fonction de la taille des séquences utilisées pour les générer.

3.3 Génération de l'ensemble du graphe

Afin de générer entièrement les ordres $\leq_d$ et $\leq_m$ dans le graphe, plusieurs ordres simples, par exemple ceux présentés dans la partie précédente, doivent être bien souvent combinés. Chaque ordre simple construit un graphe dynamique partiel qui est compris dans le graphe dynamique global, qui est testé par la détection des cycles. Dans ce cas, les nœuds sont partagés par plusieurs ordres ou graphes partiels. Un accès est ainsi au moins dans deux ordres ou graphes partiels : un pour $\leq_d$ et un pour $\leq_m$. Les nœuds *G* sont par contre souvent spécifiques à un ordre bien précis.

Nous présentons dans cette partie quelques combinaisons possibles au sein de $\leq_d$ ou de $\leq_m$.

3.3.1 Combinaison pour représenter l'ordre des accès à la mémoire

3.3.1.1 Accès à plusieurs cases mémoire

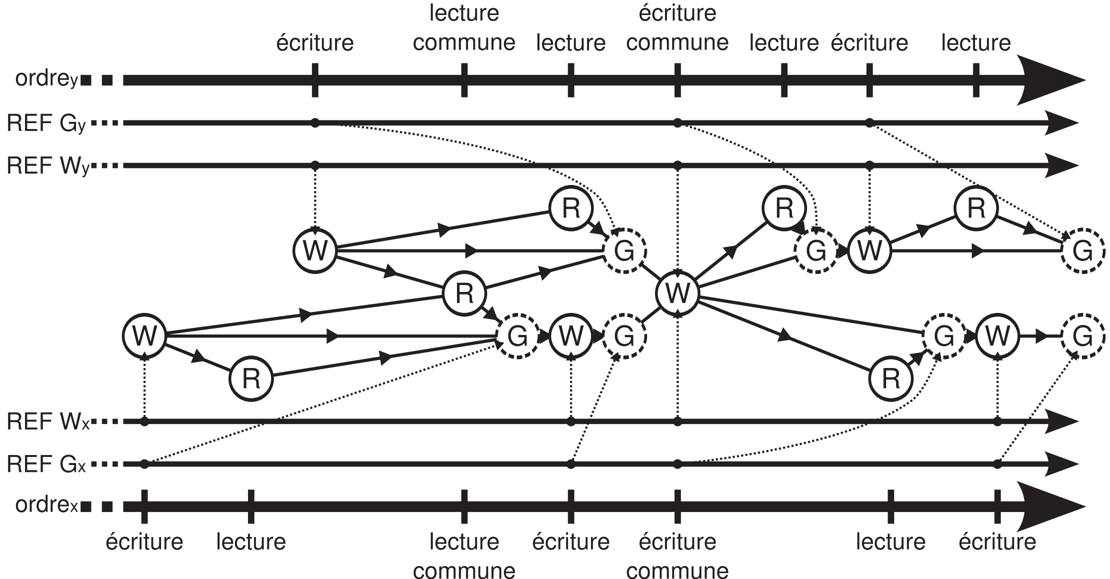


FIGURE 3.13 - Génération des graphes d'accès simultanés à plusieurs cases mémoire

Bien que les ordres des accès à la mémoire soit séparés, il peut être possible qu'un accès concerne plusieurs cases. Dans ce cas, qui est illustré dans la FIGURE 3.13, le nœud représentant l'accès est utilisé dans chacun des ordres. Mais les nœuds *G* éventuellement nécessaires ne sont par contre pas partagés. Dans cette figure, deux séquences sont représentées. Elles sont distinctes hormis pour la lecture et l'écriture *commune*. Chaque séquence crée un ordre correspondant aux dépendances RAW+WAW+WAR séparément. Seul le fait qu'elles partagent les nœuds des accès communs les relient.

3.3.1.2 Accès non atomique à une case mémoire

Afin de représenter des accès qui ne sont pas atomiques, il faut aussi combiner plusieurs ordres. La FIGURE 3.14 montre les *P* ordres nécessaires pour représenter l'ordre observé à une case mémoire lorsque les écritures ne sont pas atomiques. *P* est le nombre de processeurs. Dans ce cas il faut un couple de références pour chaque processeur.

3.3.2 Combinaison pour représenter les dépendances du programme

L'ordre d'une SI doit en effet au moins respecter que l'ordre soit correct dans un contexte mono-processeur. Dans les modèles très forts comme SC, l'ordre imposé est beaucoup plus contraint que nécessaire pour cela. Dans les modèles relâchés, il y a souvent des contraintes globales concernant uniquement les types des accès, mais aussi des contraintes concernant les dépendances de données par les registres et des dépendances spécifiques aux accès à la même case mémoire. Les figures 3.15 et 3.16 illustrent les deux catégories principales de dépendance obligatoire.

La FIGURE 3.15 illustre les dépendances qu'il faut au moins respecter entre des accès à des cases mémoire identiques. Une lecture à une variable *x* doit par exemple forcément se produire après une écriture précédente à cette même variable. Cette figure combine ainsi des ordres RAW + WAW + WAR pour deux cases mémoire (*x* et *y*). Les arcs en pointillés représentent les arcs qui

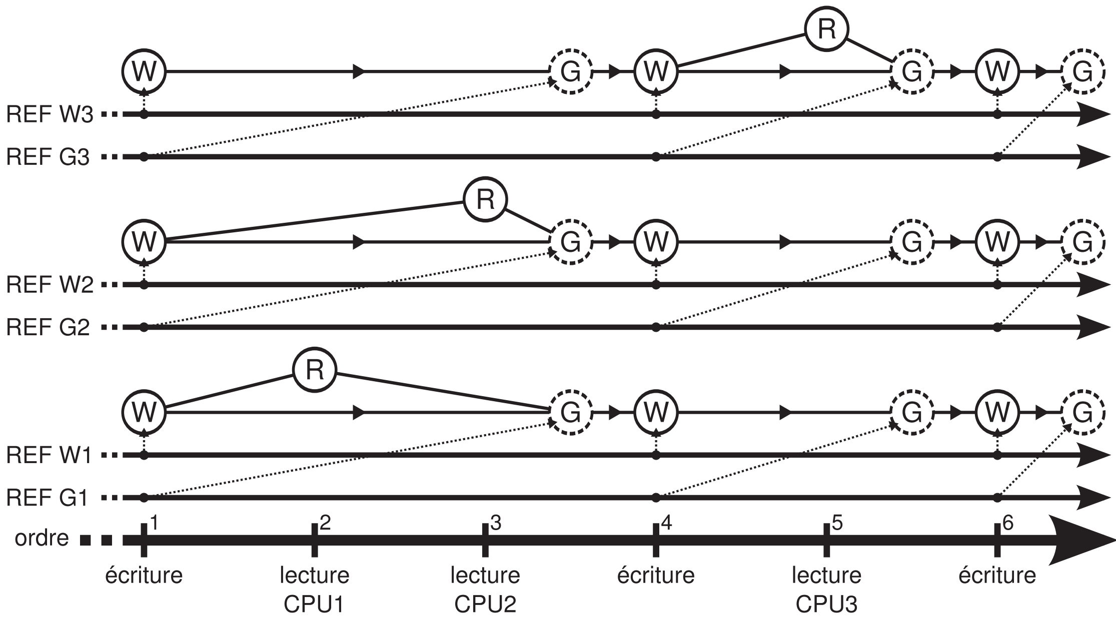


FIGURE 3.14 - Génération du graphe des accès non atomiques à une case mémoire

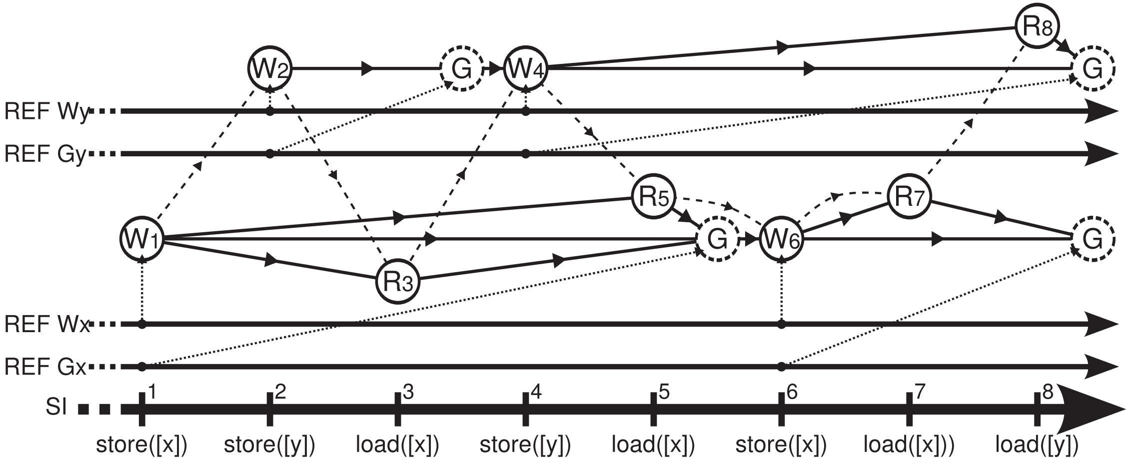


FIGURE 3.15 - Dépendances du programme via les cases mémoire

seraient mis en considérant que tous les accès sont séquentiels comme dans le modèle SC. Les dépendances sont un sous-ensemble de celles imposées par le modèle SC. Dans cet exemple chaque case mémoire est indépendante. En fonction du modèle de consistance, des ordre globaux RAR, WAW, WAR, RAW ou des barrières mémoires peuvent ajouter des contraintes supplémentaires.

L'ordre RAW + WAW + WAR pour chaque case mémoire suffit à imposer un fonctionnement correct en mono-processeur. Mais il est possible d'ajouter les dépendances RAR entre les accès à la même case au sein de la même SI pour par exemple imposer que R_5 lise une écriture au moins aussi récente que R_3. Dans la FIGURE 3.15 rien n'empêche R_5 de lire la valeur de W_1 et R_3 de lire la valeur d'une écriture insérée entre W_1 et W_6 dans l'ordre global des accès à la case mémoire x.

La FIGURE 3.16 illustre les dépendances de données à travers les registres d'un processeur dans un modèle relâché. Dans cette figure, les dépendances entre les instructions d'une même SI sont mises en place. Le nœud correspondant à une opération est numéroté avec l'indice de l'opération correspondante. Les nœuds R et W correspondent aux nœuds des accès correspondants, ils ne sont pas créés spécifiquement pour cet ordre partiel au contraire des nœuds G. Les registres sont notés r_1, r_2 et r_3. Seules les dépendances de type RAW du point de vue du contenu des registres sont

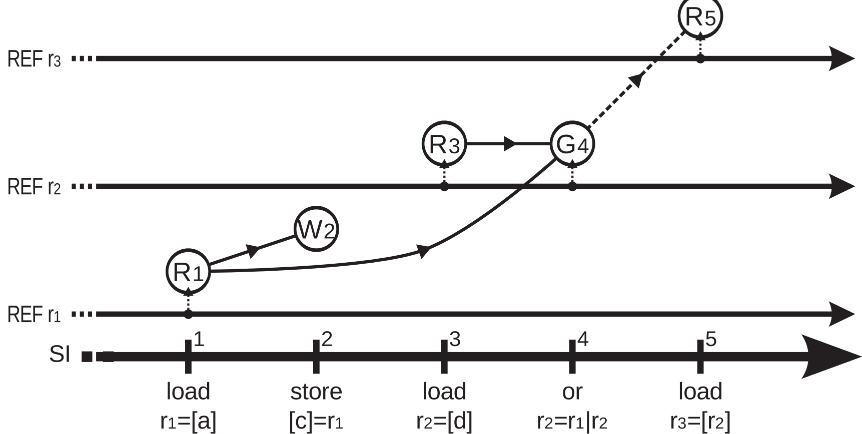


FIGURE 3.16 - Dépendances du programme via les registres

mises en place. Néanmoins le sens d'un accès du point de vue d'un registre est inversé par rapport à celui du point de vue d'une case mémoire (une lecture écrit une valeur dans un registre). Ainsi les dépendances mises en place ordonnent plutôt les écritures (en mémoire) après les lectures comme cela peut se voir entre le nœud R_1 et W_2 : l'écriture écrit la valeur du registre r_1 qui a été mise par la lecture. Pour chaque opération arithmétique, il faut ajouter un nœud supplémentaire : G_4 représente le résultat de l'opération « *or* ». Si une opération utilise plusieurs registres, il faut ajouter un arc pour chacun d'entre eux : l'opération « *or* » utilise à la fois r_1 et r_2.

L'utilisation d'un registre pour l'adresse d'un accès peut aussi être considérée comme une dépendance. Sur la figure un arc en pointillé indique la dépendance du « *load* 5 » envers le registre r_2 qui est utilisé comme adresse de la lecture.

Pour mettre en place l'ensemble des dépendances, il faut combiner les dépendances à travers les registres et celles à travers les cases mémoire.

3.3.3 Bilan sur les combinaisons

Même si les ordres partiels présentés dans la partie précédente sont simples à mettre en œuvre, il peut être nécessaire d'en combiner un très grand nombre pour pouvoir générer les contraintes d'une SI ou d'une case mémoire. Ainsi dans le cadre d'accès non atomiques, autant d'ordres partiels que de processeurs doivent être mis en place et cela pour chaque case mémoire. Inversement pour représenter les dépendances du programme d'une SI, il faut éventuellement un ordre par registre du processeur.

Mais il faut de plus, dans certains cas, un ordre partiel par case mémoire pour chaque processeur pour gérer ses dépendances spécifiques. Néanmoins chaque accès à la mémoire (et donc chaque nœud) ne sera présent que dans un nombre limité de ces ordres partiels spécifiques à une case mémoire. Ainsi même s'il peut y avoir beaucoup d'ordres partiels, un nombre très limité (et défini) d'entre eux sont utilisés pour le traitement de chaque accès.

De manière générale, nous sommes convaincus que pour chaque opération (accès mais aussi barrière mémoire par exemple) à prendre en compte dans la vérification du MCM, un nombre limité d'ordres sera impliqué. Ainsi la complexité du traitement de chacune de ces opérations sera en $O(1)$, ce qui est nécessaire pour pouvoir passer à l'échelle.

3.4 Conclusion de la génération dynamique du graphe

Nous avons présenté dans ce chapitre une méthode permettant d'effectuer une vérification dynamique de la consistance mémoire d'une exécution d'un programme sur un système muti-processeurs à mémoire partagée. Cette méthode consiste à gérer un graphe dynamique représentant une partie de l'exécution.

Le graphe est construit à partir de séquences d'événements qui correspondent aux différentes séquences d'instructions exécutées par les processeurs et aux ordres des accès de chaque case mémoire. Cette stratégie permet de construire le graphe au fur et à mesure que l'exécution se produit en temps linéaire par rapport au nombre d'événements présents dans les séquences.

La détection des cycles dans le graphe, synonymes de violations du modèle de consistance mémoire, peut aussi être effectuée de manière linéaire tout en permettant une détection réactive des cycles mais non instantanée car cela coûterait trop cher en temps de calcul.

La taille du graphe est maintenue minimale grâce à une politique de suppression des nœuds permettant de les enlever lorsqu'ils deviennent inutiles. Les conditions de l'inutilité d'un nœud nécessitent néanmoins des connaissances spécifiques sur l'exécution ; il faut par exemple pouvoir dire quand une écriture ne peut plus être lue dans le système muti-processeurs à mémoire partagée.

Le nombre de graphes partiels à construire peut néanmoins être très important : éventuellement un par couple formé d'un processeur et d'une case mémoire. Même si la gestion de chaque ordre nécessite des ressources limitées, des méthodes sont nécessaires afin de ne pas faire exploser les ressources mémoire utilisées.

4 Algorithme efficace de vérification dynamique de la consistance mémoire

Ce chapitre est dédié à la description de notre algorithme de vérification dynamique de la consistance mémoire. Cet algorithme suit la méthode du graphe dynamique décrite dans le chapitre 3 et est donc adapté à la vérification d'exécutions très longues. Ce chapitre est divisé en cinq parties.

La première partie donne tout d'abord la modélisation faite du système muti-processeurs à mémoire partagée sur lequel la vérification va être effectuée. La seconde partie est dédiée à la description de l'algorithme permettant d'atteindre une complexité algorithmique linéaire. Cet algorithme traite en particulier les accès aux périphériques ainsi que les accès atomiques, anticipés et non atomiques à la mémoire. Une étude de la complexité algorithmique et de l'occupation mémoire de l'algorithme est donnée en troisième partie.

Dans la quatrième partie, nous nous concentrons sur l'optimisation de l'occupation mémoire de cet algorithme. Des méthodes sont proposées à cette fin. Elles permettent en particulier de limiter l'occupation mémoire statique uniquement en fonction de la taille des caches et autres éléments intermédiaires de gestion des accès mémoire. Nous concluons ensuite ce chapitre dans un cinquième temps.

Chapitre 4

4.1 Représentation d'un SMPMP **58**
4.1.1 Architecture générale 58
4.1.2 Organisation de la mémoire 58
4.2 Algorithme de vérification dynamique de la consistance mémoire **59**
4.2.1 Principe général de l'algorithme 59
4.2.2 Suppression des nœuds 62
4.2.3 Exécution des instructions : ordre du programme 63
4.2.4 Complétion des accès : ordre de la mémoire 66
4.2.5 Disparition des accès 68
4.2.6 Gestion des cas particuliers 68
4.3 Bilan de l'algorithme **72**
4.3.1 Bilan sur la taille du graphe 72
4.3.2 Initialisation de l'algorithme 74
4.3.3 Occupation mémoire en pire cas 74
4.3.4 Complexité algorithmique 75
4.3.5 Bilan préliminaire 76
4.4 Optimisation de l'occupation mémoire **76**
4.4.1 Taille d'une case mémoire 76
4.4.2 Nœuds isolés 78
4.4.3 Références invalides ou isolées 79
4.4.4 Bilan des optimisations 81
4.5 Conclusion sur la vérification de la consistance mémoire **81**

4.1 Représentation d'un SMPMP

4.1.1 Architecture générale

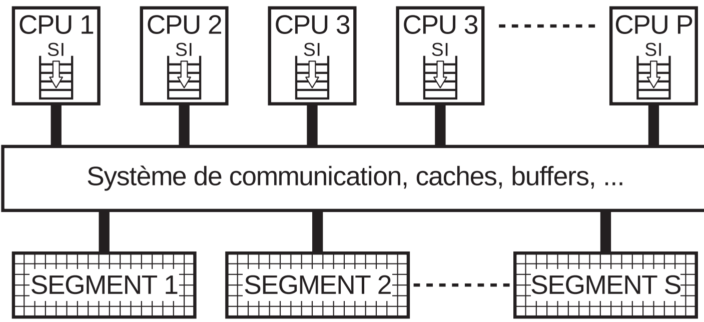


FIGURE 4.1 - Architecture d'un Système Multi-Processeurs à Mémoire Partagée

La cible de la vérification de la consistance mémoire est un MPSoC à mémoire cohérente. Un tel système est schématisé dans la FIGURE 4.1. Dans ce chapitre nous considérons que celui-ci est constitué de P processeurs et de S segments représentant la mémoire et les périphériques d'entrées/sorties. Chaque processeur est identifié par son numéro $p \in [1; P]$ et exécute une SI.

Le SMPMP peut bien sûr contenir différents composants de gestion des accès à la mémoire, en particulier des caches. Ceux-ci sont intermédiaires et ne sont vus, dans le contexte de la consistance, que comme des moyens pour effectuer les accès à la mémoire. Nous notons néanmoins C le nombre de copies de cases mémoire contenues dans ces éléments intermédiaires (caches, tampons, etc.).

4.1.2 Organisation de la mémoire

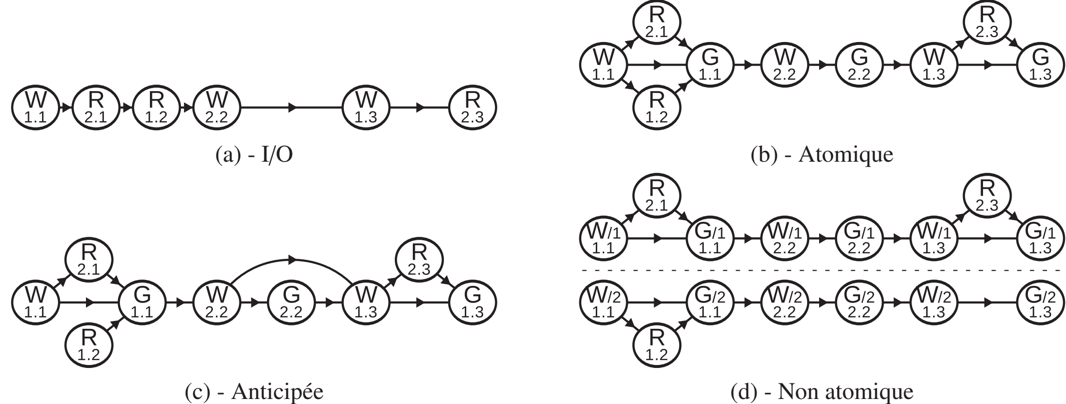
Chaque segment représente une certaine plage d'adresses mémoire et correspond à une suite contigüe de cases mémoire. Les cases mémoire d'un segment sont de taille identique, mais différents segments peuvent avoir des tailles de case différentes. La taille des cases mémoire est étudiée plus en détail dans la section 4.4. Il est néanmoins raisonnable de supposer, sans impact pour la suite, qu'une case mémoire correspond à un mot mémoire. On note M le nombre total de cases mémoire du système.

Les segments mémoire peuvent aussi différer par l'atomicité des accès qu'ils supportent. Nous supposons que l'organisation de la mémoire est statique et en particulier que les propriétés des accès à la mémoire ne changent pas et sont connus dès le début de l'exécution.

Nous considérons dans ce chapitre quatre catégories de segments. D'autres catégories existent. Néanmoins celles-ci recouvrent globalement le spectre existant des propriétés de la mémoire cohérente. En particulier, cela implique un ordre total des écritures à chaque case mémoire. Ces catégories sont décrites dans le paragraphe suivant et illustrées par la FIGURE 4.2 qui représente les contraintes résultant du même scénario d'accès à une case mémoire. Ce scénario est constitué d'accès issus de deux SIs. La séquence d'accès à la case mémoire est la suivante ($X_{p.i}, X \in \{W, R\}, p \in \{1, 2\}, i \in \{1, 2, 3\}$ représente le i-ième accès de la SI p) :

$$W_{1.1}, R_{2.1}, R_{1.2}, W_{2.2}, W_{1.3}, R_{2.3}.$$

I/O Cette catégorie correspond aux accès à des périphériques où tous les accès sont ordonnés totalement (que ce soient des écritures ou des lectures). Les accès sont tous atomiques et donc chacun représenté par un unique nœud. Du point de vue de l'ordre d'une case mémoire de cette catégorie, il n'y a pas de différence entre une lecture et une écriture. Ces accès sont communément appelés accès I/O.



(a) - I/O

(b) - Atomique

(c) - Anticipée

(d) - Non atomique

FIGURE 4.2 - Graphes partiels en fonction de la case mémoire

Atomique Cette catégorie correspond à la mémoire classique. Les accès sont tous atomiques. Les écritures sont ordonnées totalement et les lectures sont placées entre les écritures. Les lectures ne sont pas ordonnées entre elles.

Anticipée Dans cette catégorie, les lectures d'une SI peuvent anticiper les écritures de la même SI. Nous utilisons ici la représentation sans nœud additionnel de HANGAL *et al.* (cf. la section 2.2.2.2). Par rapport à la catégorie atomique, l'arc entre $W_{1.1}$ et $R_{1.2}$ est absent car ces deux accès sont de la même SI. Il y a aussi un arc supplémentaire entre $W_{2.2}$ et $W_{1.3}$ afin d'imposer leur ordre car $W_{1.3}$ est lu par $R_{2.3}$ qui suit $W_{1.3}$ dans la SI.

Non atomique Une écriture à une case mémoire de cette catégorie est non atomique (cf. la section 2.2.2.3). Elle est représentée en P nœuds : chacun correspond à l'écriture vue d'un des processeurs (dans la figure le numéro du processeur est ajouté dans le nœud). Les écritures à une même case mémoire sont ordonnées de manière identique pour tous les processeurs. Il n'y a pas d'ordre entre deux lectures lisant le résultat d'une même écriture. Dans la FIGURE 4.2d, la partie haute concerne les accès du point du vue du processeur 1 tandis que la partie basse concerne le point du vue du processeur 2.

Dans ce chapitre nous détaillons principalement le traitement de la catégorie « anticipée ». Le principe de traitement est en effet similaire pour toutes les catégories. Le traitement des autres catégories est détaillé dans l'annexe B.

Nous supposons dans ce chapitre, qu'un accès peut accéder à plusieurs cases mémoire (généralement contigües) et éventuellement de segments distincts mais dont la catégorie est identique. Il serait possible de gérer l'accès simultané à plusieurs cases mémoire de catégories différentes mais cela n'a pas d'intérêt dans notre cas. Par ailleurs les architectures interdisent généralement ce genre d'accès (lorsqu'un processeur peut en générer) ou n'en spécifient pas le comportement.

4.2 Algorithme de vérification dynamique de la consistance mémoire

4.2.1 Principe général de l'algorithme

L'algorithme décrit dans cette partie suit la méthode proposée dans le chapitre précédent. Il se décompose en trois parties :

- la construction du graphe dynamique en identifiant les arcs à placer entre les nœuds,
- la suppression des nœuds du graphe devenus inutiles afin de limiter la taille du graphe, et
- la détection des violations du MCM en détectant les cycles.

Dans cette partie, nous ne reviendrons pas sur la méthode de détection des cycles. Celle-ci a été décrite dans la section 3.1.4 du chapitre précédent. Comme nous l'avons vu dans cette section, la détection des cycles doit être lancée en fonction de métriques concernant la taille du graphe. La gestion des ces métriques ainsi que les tests nécessaires et appels déclenchant la détection des cycles ne sont pas explicitement marqués dans les algorithmes présentés dans ce chapitre afin de ne pas les alourdir. Cette gestion est implicite et une implantation de ces algorithmes doit bien entendu les insérer aux endroits nécessaires, c'est à dire lors de l'ajout ou la suppression d'éléments du graphe.

4.2.1.1 Notations utilisées

Dans la suite de ce chapitre, nous utilisons plusieurs notations. Les lettres minuscules p et m représentent respectivement un processeur et une case mémoire.

Un accès est généralement noté a. Un accès est soit une écriture soit une lecture. Dans certains cas nous utilisons néanmoins une autre lettre. On note p_a le processeur qui a émis a. Pour chaque case mémoire m accédée par a, on note a_m l'*accès partiel* à la case m correspondant à a. On note aussi $\#m_a$ le nombre de cases mémoire accédées par l'accès a.

Si a est une lecture, on note R_a le nœud correspondant. Si a est une écriture non atomique, on note $\{W_a^p | p \in [1;P]\}$ les nœuds correspondant à chacun des processeurs ; sinon on note W_a le nœud unique correspondant à l'écriture. Pour une écriture non I/O, on note G_{a_m} ou $G_{a_m}^p$ le ou les nœuds associés aux nœuds W mais qui sont spécifiques à chaque case mémoire m accédée par a.

Une instruction est notée i. Le processeur qui exécute i est noté p_i et l'ensemble des accès générés par i est noté A_i. On note SI_p la séquence des instructions exécutées par le processeur p.

On désigne l'arc entre des nœuds N_1 et N_2 par l'expression « $N_1 \to N_2$ ». Des références sont utilisées pour conserver la trace de certains nœuds, elles sont notées *réf*. Elles sont utilisées de la même manière que les nœuds, l'arc précédent peut donc être désigné par « $réf_{N_1} \to réf_{N_2}$ » en fonction du contexte. Dans les algorithmes, nous utilisons l'expression « $réf \Leftarrow N$ » pour indiquer qu'une référence *réf* conserve maintenant la trace du nœud N.

Les références sont généralement paramétrées de cette manière : $réf_{désignation}(contexte)$; le *contexte* peut par exemple être un processeur ($réf_{désignation}(p)$), un couple formé d'un processeur et d'une case mémoire ($réf_{désignation}(p, m)$) ; la *désignation* précise quelle est l'utilité de la référence au sein du contexte. Quand le contexte d'une référence est par exemple un processeur ($réf_{exemple}(p)$), cela signifie indirectement qu'il y une référence pour chaque processeur p du SMPMP. De même, lorsque le contexte est un couple formé d'un processeur et d'une case mémoire, cela signifie qu'il faut une référence pour chaque couple existant.

Un accès partiel peut être vu comme l'ensemble des références vers ses nœuds (par exemple $a_m = \{réf_W(a), réf_G(a_m)\}$ pour une écriture atomique). De même un accès peut être assimilé à l'ensemble des accès partiels : $a = \{a_m | m \text{ est accédée par } a\}$.

4.2.1.2 Entrées de l'algorithme

L'algorithme utilise les informations extraites pendant l'exécution du programme par le SMPMP. Ces informations sont traitées dynamiquement au moment où elles sont reçues, dans ce que nous appelons un événement, par l'algorithme. Il y a trois types d'événements.

Exécution d'une instruction L'algorithme a besoin de savoir quelles sont les instructions exécutées par chaque processeur ainsi que l'ordre dans lequel elles le sont. L'événement correspondant à une instruction contient les informations utiles au MCM : cela comprend en particulier la

description du ou des accès générés par l'instruction. Concernant un même processeur, les événements de chaque instruction de sa SI sont traités dans l'ordre dans lequel ils sont reçus : ils doivent donc être émis dans l'ordre de la SI. Le respect de cet ordre est raisonnable car il doit forcément exister, et être donc observable, à un endroit dans l'implantation du processeur. Ce type d'événement permet de construire l'ordre des dépendances du programme $\leq_d$.

Complétion d'un accès L'ordre des accès à la mémoire est nécessaire pour l'algorithme pour permettre une vérification en temps linéaire. L'événement de complétion d'un accès contient les informations nécessaires pour le placer dans l'ordre de sa case mémoire. Contrairement aux instructions, aucun ordre d'émission de ces événements n'est requis par rapport à l'ordre des accès à une même case mémoire : il n'y a en effet aucune garantie qu'un tel ordre soit observable lors d'une exécution.

Ainsi la construction de l'ordre des accès à une case mémoire se fait « par morceaux » : la complétion d'un accès se fait en indiquant quel est l'accès précédent dans l'ordre total de la case mémoire :

– Pour un accès I/O, chaque accès indique quel est l'accès précédent dans l'ordre de la case mémoire.
– Pour un accès classique, chaque accès (écriture ou lecture) indique quel est l'écriture précédente dans l'ordre de la case mémoire. Pour une lecture cela correspond à l'écriture lue tandis que pour une écriture cela correspond à l'écriture écrasée.

La seule contrainte d'ordre dans lequel sont émis ces événements est par rapport aux événements des instructions : les événements des instructions correspondants aux deux accès impliqués dans la complétion (celui qui est complété et celui qui sert de référence) doivent avoir été traités. Ce type d'événement permet de construire l'ordre des accès à la mémoire $\leq_m$.

Disparition d'un accès L'algorithme a aussi besoin de savoir quand un accès n'est plus accessible dans le SMPMP. Cela concerne en pratique uniquement les écritures classiques (non I/Os) car elles sont utilisées pour compléter les lectures. La nécessité de ce troisième type d'événement concerne la limitation de la taille du graphe et sera explicitée plus en détails dans la suite de cette partie.

4.2.1.3 Algorithme général

L'algorithme général de génération du graphe est présenté dans l'Algorithme 4.1. Cet algorithme correspond au traitement des différents événements qui sont issus de l'observation de l'exécution du programme. Il est présenté ici de manière très générale : il est divisé en plusieurs boucles indépendantes pour plus de clarté. Ces boucles sont en réalité entrelacées car les événements sont traités dans l'ordre où ils sont reçus.

Chaque partie de l'algorithme correspond au traitement des événements définis précédemment. Les fonctions de traitement (l'*exécution*, la *complétion* et la *disparition*) sont décrites dans la suite de ce chapitre. Durant ces trois fonctions, la détection des cycles est effectuée lorsque cela est nécessaire. La suppression des nœuds inutiles est aussi effectuée au cours de ces fonctions.

Durant ces boucles, chaque accès partiel est en particulier traité en deux ou trois étapes dans cet ordre :

1. l'*émission* de l'accès pendant l'*exécution* de l'instruction qui le génère,
2. la *complétion* de l'accès partiel, et
3. la *disparition* éventuelle de l'accès partiel.

```
pour chaque processeur p ∈ [1; P] faire
    pour chaque instruction i ∈ SI_p faire
        Exécution(i)
    fin pour
fin pour
pour chaque case mémoire m ∈ [1; M] faire
    pour chaque accès partiel a_m faire
        Complétion(a_m, r_m)        ▷ r_m est l'accès précédant a_m dans l'ordre de m
    fin pour
fin pour
pour chaque écriture non-I/O partielle e_m faire
    Disparition(e_m)
fin pour
```

Algorithme 4.1 - Algorithme général équivalent

4.2.2 Suppression des nœuds

Un nœud peut être supprimé dès lors que la certitude qu'il ne fasse pas partie d'un cycle est acquise. Ainsi, comme nous l'avons vu dans le chapitre précédent (dans la section 3.1.3), une solution permettant de supprimer les nœuds de proche en proche consiste à supprimer un nœud dès que :

- le nœud n'a pas de prédécesseurs,
- le nœud n'en aura plus d'autres.

La deuxième condition est assez difficile à remplir car cela dépend beaucoup du MCM et des contraintes qu'il faut indiquer dans le graphe dynamique. La condition peut être différente en fonction du contexte dans lequel le nœud est traité. Nous apposons donc des « marqueurs » sur les nœuds afin d'indiquer qu'il subira un traitement futur susceptible de lui ajouter un ou des prédécesseurs. Ainsi on considère la deuxième condition remplie uniquement lorsque le dernier marqueur est retiré d'un nœud. Plusieurs marqueurs peuvent être utilisés lorsque plusieurs traitements indépendants vont être effectués. Nous considérons dans la suite que tout nœud est créé avec un unique marqueur empêchant ce nœud d'être supprimé jusqu'à nouvel ordre.

Un exemple est la complétion des accès : les nœuds correspondant à un accès sont ajoutés au graphe pendant le traitement de l'instruction générant l'accès mais des prédécesseurs sont ajoutés lors de la complétion de celui-ci ; il faut donc ajouter un marqueur sur ces nœuds pour éviter qu'ils soient supprimés avant le traitement de la complétion.

L'Algorithme 4.2 contient une fonction récursive permettant de supprimer un nœud et ses successeurs si cela est possible. Cette fonction doit être appelée à chaque fois qu'un marqueur est supprimé d'un nœud.

L'ensemble des références vers le nœud supprimé sont invalidées durant cette fonction. Nous rappelons qu'elles sont utilisées dans l'algorithme pour pouvoir garder la trace d'un nœud en particulier pour y ajouter des arcs ensuite. L'invalidation d'une référence consiste à faire en sorte que toute utilisation future pour ajouter un successeur soit simplement ignorée. Une référence invalidée peut être utilisée uniquement comme source d'un arc mais pas comme destination : c'est une erreur de vouloir ajouter l'arc « $N \rightarrow réf$ » si *réf* est invalide.

Afin d'implanter efficacement les invalidations des références et les tentatives de suppression des successeurs, une solution est d'utiliser des listes chaînées, permettant un parcours en temps linéaire, de ceux-ci. Le test de chacun des successeurs implique que chaque arc du graphe va être parcouru une fois au cours de la VCM.

```
 1: fonction TentativeSuppression(nœud n)
 2:     si (n.#marqueurs ≠ 0) ∨ (n.#prédécesseurs ≠ 0) alors
 3:         return
 4:     fin si                  ▹ A ce point, les deux conditions de suppression sont remplies
 5:     pour chaque référence réf de n faire
 6:         invalider réf
 7:     fin pour
 8:     pour chaque successeur s de n faire
 9:         TentativeSupression(s)
10:     fin pour
11: fin fonction
```

Algorithme 4.2 - Algorithme de suppression des nœuds

Afin de ne pas alourdir les algorithmes présentés dans ce chapitre, les appels à cette fonction de suppression, de même que ceux pour la détection des cycles, sont implicites et ne sont pas indiqués dans la suite de ce chapitre.

4.2.3 Exécution des instructions : ordre du programme

La fonction correspondant au traitement de l'événement d'exécution d'une instruction est donnée dans l'Algorithme 4.3. Cette fonction réalise deux actions principales :

1. les nœuds correspondant à chaque accès généré par l'instruction traitée sont créés, et
2. les dépendances de la SI concernant l'instruction traitée sont mises en place.

La fonction renvoie les accès générés. Ces derniers seront ensuite utilisés pour les événements de complétion et de disparition.

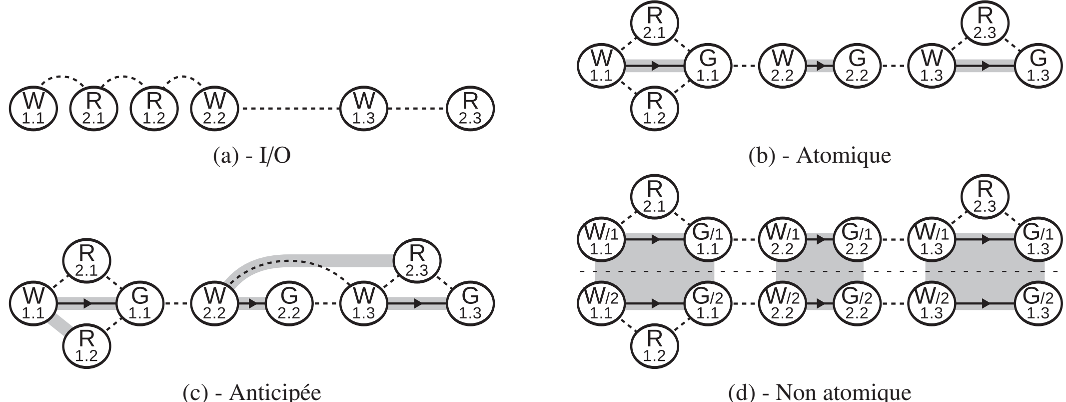
```
1: fonction Exécution(instruction i)
2:     pour chaque accès a généré par i faire
3:         Émission(a)                ▹ création des accès générés par l'instruction
4:     fin pour
5:     mise en place des dépendances de la SI
6:     return {accès générés par i}
7: fin fonction
```

Algorithme 4.3 - Traitement général d'une instruction

4.2.3.1 Émission des accès

L'émission d'un accès correspond à la création des nœuds qui le représentent. Cette opération dépend de la catégorie de l'accès (I/O, atomique, anticipée ou non atomique) et du type de l'accès (lecture, écriture).

Le principe de cette fonction est de créer le ou les nœuds R ou W d'un accès. Pour chaque écriture partielle, le ou les nœuds G sont aussi créés ainsi que les arcs entre les nœuds W et les nœuds G. La Figure 4.3 montre la partie des graphes partiels générés pour les différentes catégories de segment mémoire. Cette figure reprend le scénario de la Figure 4.2. Les traits pointillés indiquent les arcs de l'ordre de la mémoire qui ne sont pas placés lors de l'émission d'un accès. Les zones grisées regroupent les nœuds faisant partie d'un même accès partiel.



(a) - I/O

(b) - Atomique

(c) - Anticipée

(d) - Non atomique

FIGURE 4.3 - Graphes partiels créés lors de l'émission d'un accès

Rien n'empêche un nœud de faire partie de plusieurs accès partiels. Cela arrive en particulier dans la catégorie anticipée pour les nœuds *W*. Dans la plupart des cas les accès partiels sont uniquement constitués des nœuds *R* seuls ou des nœuds *W* associés aux nœuds *G*. Pour une lecture de la catégorie anticipée, le nœud *R* doit de plus être associé au nœud *W* de l'écriture précédente de la SI à la même case mémoire (cf. la FIGURE 4.3c). Ce point est discuté juste après.

Les nœuds *R* et *W* sont globaux à un accès tandis que les nœuds *G* sont spécifiques aux accès partiels. Lorsqu'un accès accède plusieurs cases mémoire, le nœud *R* ou *W* est commun à tous les accès partiels. Dans le cas d'une écriture il est donc partagé entre les accès. Cela est illustré dans la FIGURE 4.4 où une écriture est commune aux deux cases mémoire [*x*] et [*y*]. Cela ne pose pas de problème car les nœuds peuvent être référencés plusieurs fois (les accès partiels sont assimilés à des ensembles de référence).

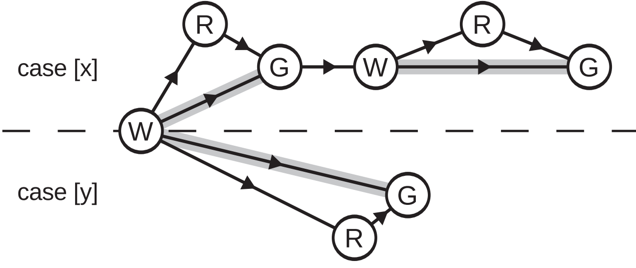


FIGURE 4.4 - Accès atomiques partiels à deux cases mémoire

L'ALGORITHME 4.4 donne la fonction d'émission pour la catégorie « anticipée ». Les algorithmes des catégories « I/O », « atomique » et « non atomique » sont dans l'annexe B. On rappelle que $\#m_a$ est le nombre de cases mémoire accédées par a. En plus de créer les nœuds *R* et *W*, des marqueurs leur sont ajoutés : un marqueur par case mémoire accédée qui sera retiré lors de la complétion de l'accès partiel correspondant. Ces marqueurs empêchent que ces nœuds soient supprimés avant que leurs accès soient complétés.

Les nœuds *G* créés n'ont logiquement qu'un seul marqueur : ils ne sont pas partagés entre les accès partiels. Le marqueur d'un nœud *G* sera supprimé lors de la disparition de l'accès partiel correspondant.

Le traitement des accès dont les écritures peuvent être anticipées est spécial. Selon la méthode que nous utilisons, il est nécessaire de connaître quelle est l'écriture précédente dans la SI au moment où l'on gère l'ordre de la mémoire de cet accès, c'est-à-dire lors de la complétion de

```
 1: fonction ÉMISSIONANTICIPÉE(accès a)
 2:     si a est une lecture alors
 3:         crée le nœud R_a
 4:         AJOUTEMARQUEURS(R_a, #m_a − 1)              ▷ −1 car il y a déjà 1 marqueur initial
 5:         pour chaque case m accédée par a faire
 6:             réf_anticip(a_m) ⇐ réf_anticip(p, m)    ▷ mémorise quelle est l'écriture précédente
 7:                                                        de la SI à la même case mémoire
 8:         fin pour
 9:     sinon
10:         crée le nœud W_a
11:         AJOUTEMARQUEURS(W_a, #m_a − 1)              ▷ −1 car il y a déjà 1 marqueur initial
12:         pour chaque case m accédée par a faire
13:             crée le nœud G_{a_m}                    ▷ le nœud a un unique marqueur initial
14:             ajoute l'arc W_a → G_{a_m}
15:             réf_anticip(p, m) ⇐ W_a                 ▷ enregistre W_a comme étant la dernière écriture à m
16:         fin pour
17:     fin si
18: fin fonction
```

ALGORITHME 4.4 - Émission d'un accès à un segment de catégorie « anticipée »

cette lecture. Cette information ne pouvant être connue que pendant l'événement d'exécution de l'instruction générant l'accès, il est nécessaire de sauvegarder cette écriture avec l'accès partiel.

Une référence ($réf_{anticip}(p, m)$) est ainsi utilisée pour conserver la dernière écriture effectuée par chaque processeur à chaque case mémoire. Ces références sont mises à jour à chaque fois qu'une écriture est émise (ligne 15). Elles sont utilisées lorsqu'une lecture est émise (ligne 6). L'écriture précédente est alors conservée dans une référence associée à l'accès partiel que nous notons ($réf_{anticip}(a_m)$). Comme cette référence ne sera utilisée que pour ajouter un successeur à son nœud référencé, il n'est pas nécessaire de lui ajouter un marqueur.

4.2.3.2 Dépendances de la SI

L'ALGORITHME 4.3 est très vague concernant la mise en place des dépendances de la SI. Celles-ci sont en effet complètement dépendantes du MCM. Si l'on désire simplement vérifier la cohérence mémoire, aucune dépendance n'a à être mise en place. A l'inverse pour vérifier le modèle SC, il faut créer un ordre total des accès émis, ce qui est simple. Entre ces deux extrêmes, de nombreuses possibilités existent, plus ou moins complexes à mettre en œuvre.

De manière générale, plus le modèle est relâché, plus il y a de cas particuliers à prendre en compte et donc de graphes partiels à construire. Comme nous l'avons vu dans le chapitre précédent, chacun de ces graphes nécessite quelques références à conserver.

Dépendances via un registre L'ALGORITHME 4.5 donne par exemple une fonction établissant des dépendances de données par les registres. Dans cette fonction, $réf_r(p)$ est une référence vers un nœud représentant la donnée contenue dans le registre r du processeur p. p est ici le numéro de la SI exécutant l'instruction en cours de traitement. o désigne une opération qui peut être par exemple un accès mémoire ou une opération arithmétique. N_o désigne le nœud correspondant à cette opération : un nœud R ou W dans le cas d'un accès mémoire ou simplement un nœud G créé spécifiquement pour représenter l'opération dans la SI dans le cas d'une opération arithmétique.

Dans le cas où le nœud est créé, son marqueur initial est enlevé (ligne 14) car ce nœud ne sera utilisé ensuite que dans cette même fonction et donc uniquement comme source d'un futur arc (ligne 8).

```
 1: fonction DépendanceRegistre(opération o)
 2:     si o est un accès alors
 3:         N_o désigne le nœud correspondant à l'accès
 4:     sinon
 5:         crée un nœud N_o                          ▹ le nœud a un marqueur initial
 6:     fin si
 7:     pour chaque registre r lu par o faire
 8:         ajoute réf_r(p) → N_o
 9:     fin pour
10:     pour chaque registre r écrit par o faire
11:         réf_r(p) ⇐ N_o
12:     fin pour
13:     si o n'est pas un accès alors
14:         EnleveMarqueur(N_o)                       ▹ ainsi N_o n'a plus de marqueur et
15:     fin si                                        sera donc supprimé dès que possible
16: fin fonction
```

Algorithme 4.5 - Mise en place des dépendances via des registres

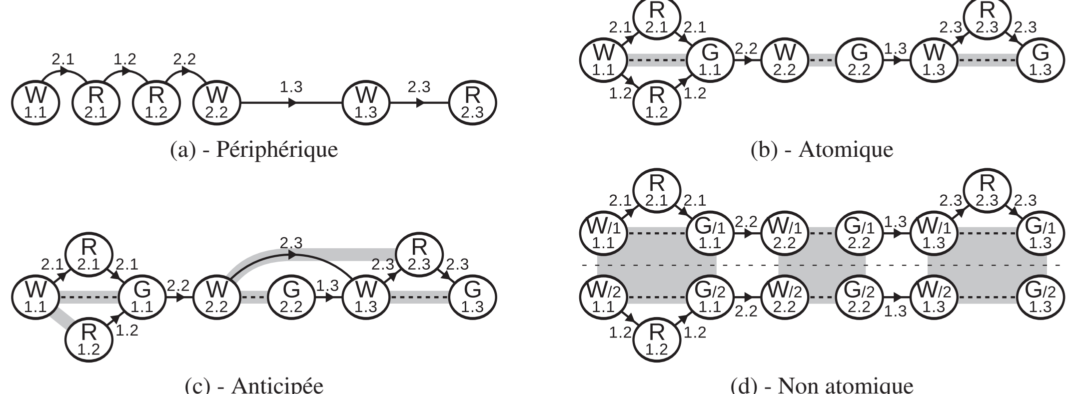
Dépendances via une case mémoire Afin de gérer les dépendances de données d'une SI à travers les cases mémoire, il est nécessaire de construire un graphe partiel par case mémoire. Nous ne donnons pas d'exemple d'algorithme car suivant les cas (catégorie de l'accès, modèle de consistance mémoire), l'ordre partiel n'est pas forcément le même. L'ordre partiel peut aller de l'ordre total entre tous les accès à une même case mémoire à uniquement un ordre WaW+WaR si la catégorie est anticipée et si le MCM n'impose pas les dépendances RaR (l'architecture ARM n'impose pas d'ordre entre deux lectures à la même adresse par exemple).

La manière de construire ces ordres a été décrite dans le chapitre précédent. Le point important est que la mise en place de ces ordres partiels impose de conserver un grand nombre de références pour chaque processeur (une ou deux pour chaque case mémoire, en fonction de l'ordre partiel).

Autres dépendances Les dépendances à travers les registres et les cases mémoire sont les principales sources de contraintes d'ordre à mettre place. Toutefois, il faut par exemple aussi gérer les barrières mémoires et les contraintes globales entre accès (c.-à-d. celles qui sont indépendantes des cases accédées) s'il y en a.

4.2.4 Complétion des accès : ordre de la mémoire

La Figure 4.5 illustre les arcs qui sont mis en place lors de la complétion pour les différentes catégories étudiées. Elle possède les mêmes bandes grises que la Figure 4.3 qui surlignent les accès partiels et le groupement d'une lecture anticipée avec l'écriture précédente. Chaque arc est étiqueté avec l'identifiant de l'accès dont le traitement de la complétion provoque son ajout au graphe. Les traits pointillés sont à la place des arcs qui ont été mis en place pendant l'émission des accès. La complétion permet de finaliser la création de l'ordre de la mémoire.



(a) - Périphérique

(b) - Atomique

(c) - Anticipée

(d) - Non atomique

FIGURE 4.5 - Graphes partiels créés lors de la complétion d'un accès

La complétion de la catégorie anticipée est décrite dans l'ALGORITHME 4.6. Le traitement des autres catégories est détaillé dans l'annexe B. Celui-ci consiste à mettre en place l'accès partiel dans l'ordre de la mémoire. Chaque accès partiel composant un accès doit être complété car dans le cas général ils ne sont pas complétés par rapport au même accès référent.

1: **fonction** COMPLÉTIONANTICIPÉE(accès a_m, accès référent r_m)
Prérequis : a_m n'a pas déjà été complété
Prérequis : r_m est une écriture qui n'a pas encore *disparue*
2: **si** a est une lecture **alors**
3: ajoute $R_a \rightarrow G_{r_m}$
4: **si** a est issue d'un autre processeur que r **alors**
5: ajoute $W_r \rightarrow R_a$
6: **fin si**
7: **si** $réf_{anticip}(a_m) \neq W_r$ **alors**
8: $réf_{anticip}(a_m) \Leftarrow W_r$
9: **fin si**
10: ENLÈVEMARQUEUR(R_a)
11: **sinon**
Prérequis : r_m n'a pas encore été utilisé comme référent pour compléter une écriture
12: ajoute $G_{r_m} \rightarrow W_a$
13: ENLÈVEMARQUEUR(W_a)
14: **fin si**
15: **fin fonction**

ALGORITHME 4.6 - Complétion d'un accès « anticipé »

Durant cette fonction un marqueur est enlevé du nœud principal de l'accès (lignes 10 et 13), ainsi lorsque tous les accès partiels d'un accès sont complétés, son nœud n'aura plus de marqueur. Ce nœud pourra alors être supprimé lorsqu'il respectera les conditions de suppression (cf. la section 4.2.2.

Trois conditions doivent être remplies et préalablement être vérifiées à l'entrée de cette fonction.

1. l'accès a_m ne doit pas avoir été déjà complété ;
2. l'accès r_m ne doit pas avoir disparu ; et
3. si l'accès a_m est une écriture, l'accès référent r_m ne doit pas avoir déjà servi à compléter un tel accès.

Les deux premières conditions correspondent aux hypothèses sur les événements de *disparition* et *complétion*. La troisième condition correspond aux contraintes de la cohérence mémoire : elle impose respectivement un ordre total des écritures à une case mémoire.

Lors de la complétion d'une lecture, le nœud W de l'accès référent est utilisé ; néanmoins seul un successeur lui est ajouté (ligne 5). Ce n'est donc pas grave si ce nœud a déjà été supprimé (si l'accès référent a déjà été complété). Par contre un prédécesseur est ajouté au nœud G de l'accès référent lors de la complétion d'une lecture (ligne 3) : ce nœud ne doit donc pas être supprimé tant que des lectures peuvent l'utiliser comme référent (c'est à dire jusqu'à l'événement de disparition du référent).

Pour la complétion des écritures, seul un successeur est ajouté au nœud G (ligne 12), il est donc théoriquement possible de supprimer ce nœud si aucune lecture ne sera ensuite complétée avec l'accès référent. Mais en pratique, une écriture ne disparaît (c.-à-d. n'est plus lisible par aucune lecture) qu'après avoir été complétée (c.-à-d. écrasée par une autre écriture). Il est ainsi inutile de chercher à supprimer ces nœuds G avant la disparition des écritures.

4.2.5 Disparition des accès

Le traitement de l'événement de disparition des accès est donné dans l'Algorithme 4.7. Sa seule fonction est d'enlever le marqueur du ou des nœuds G de l'accès partiel traité.

```
1: fonction DisparitionAnticipée(écriture a_m)
2:     EnlèveMarqueur(G_{a_m})
3: fin fonction
```

Algorithme 4.7 - Disparition d'un accès « anticipé »

Une fois que l'événement de disparition d'une écriture est traité, cette écriture partielle n'a plus d'existence au sein de l'algorithme. Les accès qui n'ont pas d'événement de disparition (les lectures par exemple) n'ont plus d'existence dès qu'ils ont été complétés. Le fait qu'un accès partiel n'a plus d'existence ne signifie pas que ses nœuds ont forcément été supprimés du graphe mais uniquement que la structure de donnée éventuellement utilisée pour représenter cet accès partiel n'existe plus.

4.2.6 Gestion des cas particuliers

Nous avons abordé jusqu'à présent le cas général de la VCM. L'ordre du programme et de la mémoire sont en effet mis en place respectivement lors du traitement des événements d'*exécution* et de *disparition*. Seules les écritures et les lectures sont par ailleurs gérées précédemment.

Nous donnons dans cette section deux exemples qui nécessitent des traitements particuliers : la gestion des accès à la mémoire ayant une sémantique spéciale et la gestion des barrières mémoires quand les accès sont non atomiques.

4.2.6.1 Gestion des accès spéciaux

Jusqu'à présent, nous n'avons considéré que les accès classiques : les lectures et les écritures. Même si ce sont les principaux accès émis, il existe des variantes de ceux-ci. Il est en effet extrêmement pratique d'un point vue logiciel de pouvoir faire une lecture suivie d'une écriture en ayant la garantie qu'une autre écriture ne se soit pas intercalée entre les deux. Cette propriété est en effet très utile pour gérer les synchronisations entre plusieurs parties du programme. Cela consiste à faire ce qu'on appelle un échange (*swap*). Plusieurs solutions matérielles sont possibles pour faire cela.

L'architecture peut fournir un accès spécial *swap* qui permet de le faire ou se baser sur deux accès que nous appelons *Load Linked* (LdL) et *Store Conditional* (StC). Ces accès sont appelés accès exclusifs dans l'architecture ARM : *load exclusive* et *store exclusive*. L'architecture peut d'ailleurs très bien fournir une instruction *swap* et l'implanter en utilisant deux accès. Si le *swap* est réellement supporté par l'architecture celui-ci est forcément correct et peut être géré exactement de la même manière qu'une écriture : le nœud W représente à la fois la lecture de l'écriture précédente et la nouvelle écriture. Il n'y alors aucune vérification supplémentaire à faire.

Si deux accès distincts sont utilisés, il faut par contre vérifier qu'ils n'ont pas été interrompus. Le principe des LdL et StC est de faire l'échange à l'aide de deux accès dérivés de la lecture et de l'écriture classique.

- Le LdL est une lecture ordinaire qui indique qu'elle sera suivie d'un StC.
- Le StC est une écriture conditionnée, elle n'a effectivement lieu que si aucune écriture n'a eu lieu depuis le dernier LdL à la même case.

Le système doit alors garantir, qu'en cas de succès du StC, aucune écriture ne s'est intercalée entre les deux accès. En cas d'échec, la StC n'écrit rien. L'information d'échec ou de succès est transmise au processeur émetteur.

Le cas du *swap* peut être géré en considérant que le *swap* est en fait un couple formé d'une LdL et d'une StC. Nous traitons donc ici uniquement ce dernier cas qui est plus général. Il est important de noter que l'information de l'échec ou non du StC doit être connue dès sa phase d'*émission*. Nous ne nous occupons ici de vérifier que le succès du StC car seules les conditions de succès sont importantes : la complexité d'implantation de ces accès entraîne une implantation limitée. Il est souvent possible qu'un échec d'un StC ne soit pas provoqué par une écriture intercalée depuis le dernier LdL.

Vérifier qu'un StC réussi est correct consiste donc à vérifier s'il a bien écrasé une écriture lue par le précédent LdL (dans la même SI) à la même case mémoire. Le problème n'est pas de vérifier que c'est bien le LdL précédent dans la SI, cela peut être garanti facilement en ordonnant les StCs, les LdLs et les autres accès à la même adresse lors du traitement de l'exécution des instructions les générant. Le problème est de vérifier qu'un StC réussi écrase bien une écriture lue par un LdL du même processeur.

Le point commun entre un LdL et un StC réussi est qu'ils sont complétés avec la même écriture : l'un la lit tandis que l'autre l'écrase. Le principe est donc d'utiliser cette écriture (lue ou écrasée) pour conserver les informations nécessaires à la vérification. Les ajouts à faire dans les fonctions du traitement de complétion et de disparition sont donnés dans l'Algorithme 4.8.

Ces deux fonctions doivent être ajoutées au traitement de la complétion et de la disparition. Elles ne font que vérifier qu'un LdL du même processeur a bien lu l'écriture écrasée par chaque StC. Ces fonctions passent à l'échelle car une écriture e_m ne sert à compléter une écriture (et donc un StC) qu'une seule fois. Ainsi les étiquettes ne sont parcourues (ligne 10) au maximum qu'une seule fois.

Le principe est que si un StC n'écrase pas une écriture lue par un LdL, la référence ($réf_{\mathrm{sc}}(e_m)$) de son nœud sera valide lors de la disparition de l'écriture écrasée. L'erreur pourra alors être signalée. Le nœud ainsi référencé ne peut en aucun cas être supprimé avant cette disparition puisqu'il suit le nœud G_{e_m} dans l'ordre de la mémoire et que ce nœud ne peut être supprimé avant la disparition de e_m.

Il faut noter que la méthode décrite conserve dans tous les cas une complexité moyenne et une occupation mémoire en $O(1)$ car chaque LdL partiel provoque l'ajout d'une unique étiquette et l'ensemble la contenant ne sera testé qu'une seule fois. Si les étiquettes sont stockées dans une liste, le coût du parcours de cette liste peut en effet être rattaché aux différents LdL qui y ont mis une étiquette.

```
1: fonction ComplétionLlSc(accès a_m, écriture e_m)
2:     selon(a_m)
3:         quand LdL alors
4:             si réf_sc(e_m) est invalide alors
5:                 ajoute l'étiquette ll_p à l'accès e_m      ▹ p est le numéro de la SI ayant émis a_m
6:             sinon
7:                 invalide réf_sc(e_m)                       ▹ réf_sc(e_m) suit bien un LdL de la même SI
8:             fin si
9:         quand StC alors
10:            si e_m n'a pas d'étiquette ll_p alors           ▹ p est le numéro de la SI ayant émis a_m
11:                réf_sc(e_m) ⇐ W_a
12:            fin si
13:    fin selon
14: fin fonction

15: fonction DisparitionLlSc(écriture e_m)
16:     si réf_sc(e_m) est valide alors
17:         erreur                          ▹ L'accès correspondant à réf_sc(e_m) n'est pas correct
18:     fin si
19: fin fonction
```

Algorithme 4.8 - Traitement des accès LdL et StC

Néanmoins l'occupation mémoire de l'algorithme peut tout de même exploser à cause de ce cas. Le nombre d'étiquettes de LdL stockés dans un accès e_m n'est théoriquement pas limité. Les étiquettes sont supprimées en même temps que l'accès partiel e_m, c'est à dire lors de sa disparition. Bien qu'il soit peu probable que ce nombre explose lors d'un fonctionnement normal, cela est possible : par exemple si une SI effectue des LdL en boucle à une case mémoire qui n'est jamais écrasée (par exemple dans le cadre d'une tentative de prise de verrou échouant éternellement car on a oublié de le déverrouiller). Pour palier à ce cas, il faut éviter de dupliquer une étiquette ll_p si celle-ci est déjà présente pour borner le nombre d'étiquettes à p. Une solution est d'utiliser une liste puis, lorsque la taille de la liste atteint P, de passer à un tableau indiquant la présence (ou non) de chacune des P étiquettes.

4.2.6.2 Synchronisation et non-atomicité

Afin de synchroniser différentes parties d'un programme, les architectures définissent des instructions spécifiques comme les barrières. Ainsi une des attentes par rapport à ces barrières est qu'une fois la barrière effectuée par une SI, toute autre SI qui se synchronise avec cette barrière (en effectuant aussi une barrière par exemple) doit en particulier avoir une vision au moins aussi récente des cases mémoire que celle de la première SI lors de sa barrière. En pratique cela signifie que toute lecture faite par la deuxième SI doit lire une donnée au moins aussi récente que celle lue ou écrite par la première SI avant la barrière. Cela permet par exemple de garantir que des données protégées par un verrou soient correctement propagées d'une SI à l'autre lorsque le verrou change de propriétaire.

L'Algorithme 4.9 donne l'algorithme permettant de gérer les barrières mémoire comme expliqué dans la section 3.2.6. Cela consiste principalement à conserver un nœud représentant la dernière barrière exécutée et un nœud représentant la prochaine.

```
1: fonction ExécutionBarrière
2:     EnlèveMarqueur(réf_{dernière barrière}(p))          ▷ enlève le marqueur initial
3:     réf_{dernière barrière}(p) ⇐ réf_{prochaine barrière}(p)
4:     réf_{prochaine barrière}(p) ⇐ nouveau nœud           ▷ ce nœud a un marqueur initial
5:                                  qui sera supprimé lors de la prochaine barrière
6:     ajoute réf_{dernière barrière}(p) → réf_{prochaine barrière}(p)
7: fin fonction
```

Algorithme 4.9 - Gestion des barrières mémoires

Prenons l'exemple de deux SIs, la première effectue une lecture R_1, lisant l'écriture W, puis une barrière B_1 ($R_1 < B_1$) tandis que la seconde effectue une barrière B_2 puis une lecture R_2 à la même adresse que R_1 ($B_2 < R_2$) et supposons que B_2 suit B_1 ($B_1 < B_2$). On obtient donc la séquence suivante « $R_1 < B_1 < B_2 < R_2$ ». Les barrières sont notées B_p dans cet exemple : elles doivent être représentées dans le cas général par un nœud spécifique mais sont dans certains cas simplement intégrée à des accès (par exemple les accès exclusifs de l'architecture ARM sont des barrières). $B_1 < B_2$ découle en pratique de contraintes dues à des accès à la mémoire effectués à des fins de synchronisation.

Si l'écriture est atomique, aucun problème ne se pose car $W < R_1$ grâce à l'ordre de la mémoire (R_1 lit W), il y a donc forcément $W < R_2$ et R_2 lit donc une valeur au moins aussi récente que R_1.

Par contre si l'écriture est non atomique, l'ordre de la mémoire impose seulement $W^1 < R_1$ et rien sur la sous écriture W^2. Il faut donc imposer des contraintes supplémentaires afin de garantir $W^2 < R_2$ et obliger R_2 à lire une valeur récente. En pratique on cherche donc à imposer $\{W^i < B_1, \forall i \in [1;P]\}$, mais cela est difficile car W n'est pas forcément issu de la même SI que B_1.

De manière similaire, si la deuxième SI fait une écriture W_2 et non une lecture, on cherche à imposer $\{B_2 < W_2^i, \forall i \in [1;P]\}$ pour empêcher que R_1 puisse avoir lu W_2^1. Cette contrainte est néanmoins plus simple à mettre en place car W_2 et B_2 sont effectuées par la même SI.

```
 1: fonction ÉmissionNonAtomique(accès a)
 2:     si a est une lecture alors
 3:         ajoute réf_{dernière barrière}(p) → R_a → réf_{prochaine barrière}(p)
 4:         pour chaque m accédée par a faire
 5:             réf_{barrière}(a_m) ⇐ réf_{prochaine barrière}(p)
 6:             AjouteMarqueur(réf_{prochaine barrière}(p))   ▷ car le nœud reçoit un prédécesseur
 7:         fin pour                                           lors de la complétion
 8:     sinon
 9:         pour chaque i ∈ [1; P] faire
10:             ajoute réf_{dernière barrière}(p) → W_a^i → réf_{prochaine barrière}(p)
11:         fin pour
12:     fin si
13: fin fonction

14: fonction ComplétionLectureNonAtomique(lecture l_m, écriture e_m)
15:     pour chaque i ∈ [1; P] faire
16:         ajoute W_e^i → réf_{barrière}(l_m)
17:     fin pour
18:     EnlèveMarqueur(réf_{barrire}(l_m))        ▷ enlève le marqueur mis lors de l'émission
19: fin fonction
```

Algorithme 4.10 - Synchronisation quand les écritures sont non atomiques

Le problème est principalement que les écritures lues, que l'on doit contraindre, sont inconnues au moment où une lecture est traitée pour l'exécution de l'instruction correspondante. Quand l'écriture est atomique, l'ordre est implicite car il découle des dépendances de la mémoire. Lorsque cet ordre est restreint comme pour les écritures non atomiques ou anticipées, il faut donc l'imposer explicitement. L'Algorithme 4.10 donne ainsi les ajouts à apporter aux fonctions d'exécution, d'émission et de complétion. Dans ce cas, comme l'écriture lue d'une lecture n'est connue que lors de sa complétion, l'ajout des contraintes est déporté dans la complétion de la lecture.

L'algorithme ne décrit que le cas des accès non atomiques, les autres doivent bien sûr aussi être gérés et donc en particulier placés entre les deux références conservant les barrières comme cela est fait pour les accès non atomiques (lignes 3 et 10). Une référence ($réf_{\text{barrière}}(l_m)$) est utilisée pour conserver la trace de la prochaine barrière au moment de l'émission d'une lecture (ligne 5) et l'utiliser lors de la complétion de celle-ci (ligne 16).

4.3 Bilan de l'algorithme

Dans cette partie, nous faisons le bilan de l'algorithme présenté précédemment. Nous traitons, ici, de la complexité de celui-ci ainsi que de son occupation mémoire.

Les grandeurs utilisées dans cette parties sont les suivantes :

- P est le nombre de processeurs,
- S est le nombre de segments,
- M est le nombre de cases mémoire,
- C est le nombre de cases mémoire dans les caches et autres tampons,
- I est le nombre d'instructions exécutées,
- N est le nombre d'accès partiels générés,
- U est le nombre de mises à jour et invalidations des caches.

Il est important d'avoir à l'esprit que ces nombres ont des ordres de grandeur complètement différents. Nous les classons en quatre catégories :

- P et S sont des paramètres architecturaux plutôt petits.
- C est d'une grandeur moyenne.
- M est un grand nombre.
- I, N et U sont du même ordre de grandeur. Ce ne sont pas des paramètres architecturaux et sont supposés bien plus grands que tous les autres nombres. Ces grandeurs sont estimées assez proches. Cela est assez logique pour le nombre d'instructions I et le nombre d'accès N mais pas forcément pour U. U représente le nombre de mises à jour ou invalidations des caches. Il nous semble raisonnable de supposer que ce nombre soit lié à N. L'implantation d'un système MPSoC aura en effet tout intérêt à éviter l'explosion de ce nombre qui représente des opérations et éventuellement des communications entre caches.

En conséquence, on peut raisonnablement supposer que

$$\{P, S\} \ll C \ll M \ll \{I, N, U\}.$$

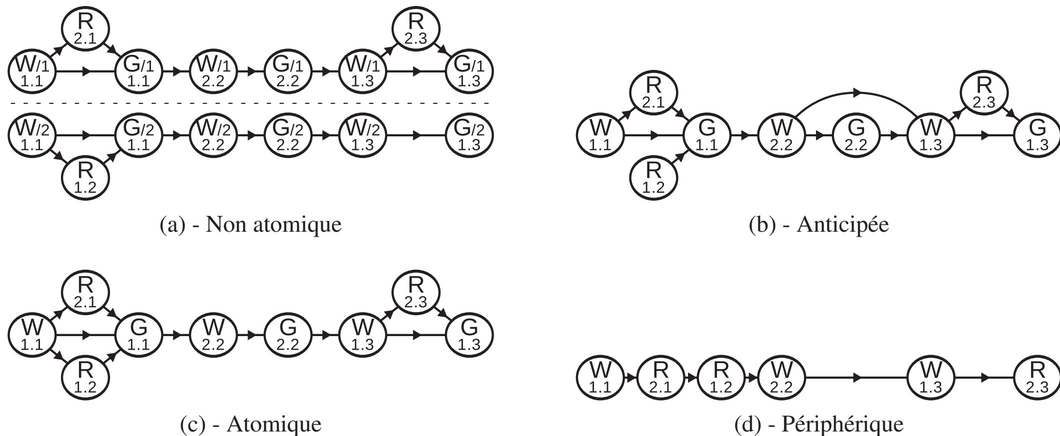
Dans l'algorithme présenté, nous avons donc cherché avant tout à avoir une complexité linéaire par rapport à I, N, U tout en limitant au maximum l'occupation mémoire en supprimant les nœuds du graphe.

Nous introduisons aussi le terme na qui vaut 1 si il y a des écritures non atomiques et 0 sinon. Ainsi P^{na} vaut P ou 1 en fonction de l'atomicité des écritures.

4.3.1 Bilan sur la taille du graphe

La taille du graphe impacte directement l'occupation mémoire ainsi que la complexité à travers la fonction de suppression des nœuds.

Chaque accès peut accéder à une ou plusieurs cases mémoire. Néanmoins nous faisons l'hypothèse que le nombre de cases accédées est borné par un paramètre architectural. En considérant qu'une case mémoire est un octet, cette borne serait probablement 4 dans un système 32 bits car les accès ne peuvent pas, la plupart du temps, y dépasser un mot de 32 bits. Il faut noter que cette borne ne limite pas pour autant la largeur des instructions de *load* et *store* d'une architecture. Dans l'architecture SPARC par exemple, la limite est fixée à 8 octets : au delà, une instruction génère plusieurs accès.



(a) - Non atomique

(b) - Anticipée

(c) - Atomique

(d) - Périphérique

FIGURE 4.6 - Ordre de la mémoire pour chaque catégorie

4.3.1.1 Cas des accès I/Os, atomiques ou anticipés

Nous nous concentrons dans un premier temps sur le cas où il n'y a pas d'accès non atomiques. Le graphe est construit au fur et à mesure que les événements suivants sont traités : *exécution* d'une instruction (comprenant l'*émission* des accès), *complétion* d'un accès partiel et *disparition* d'un accès partiel. Chaque événement provoque la création d'un nombre borné de nœuds et d'arcs.

Chaque accès est en effet représenté par un seul nœud R ou W. Pour chaque écriture partielle, il faut de plus un nœud G pour représenter l'ordre observé des accès. Cet ordre nécessite au plus deux arcs par accès partiels comme le rappelle la FIGURE 4.6.

La mise en place des ordres des dépendances du programme est plus compliquée et dépend principalement de l'architecture et de son modèle de consistance. Néanmoins nous sommes convaincus que le nombre de nœuds et d'arcs supplémentaires nécessaires est borné pour chaque instruction ou accès partiel.

En particulier nous considérons que chaque nœud fait partie d'un nombre fini de graphes partiels, tel que ceux décrits dans le chapitre précédent. Les principaux graphes partiels sont :

– les dépendances RAR, WAR, RAW ou WAW globaux sur tous les accès ;
– les dépendances des barrières mémoires ;
– les dépendances à travers les registres ; et
– les dépendances à travers les cases mémoire ;

Les trois premiers points donnent un nombre total limité de graphes. Le dernier point n'est pas limité mais chaque accès partiel n'apparait par construction que dans un seul de ces graphes.

En conséquence le nombre de nœuds et d'arcs évolue de manière linéaire avec le nombre d'instructions et d'accès : $O(I + N)$.

4.3.1.2 Cas des écritures non atomiques

Nous considérons maintenant le cas où les écritures peuvent être non atomiques. Le raisonnement est similaire au cas précédent hormis le fait que chaque nœud peut être remplacé par P nœuds à cause de la non atomicité. Ainsi le nombre de nœuds et d'arcs évolue en $O((I + N) \times P)$.

De manière plus générique, le nombre de nœuds, ainsi que le nombre d'arcs, évolue en $O((I + N) \times P^{na})$.

4.3.2 Initialisation de l'algorithme

Un point important de l'algorithme qui n'a pas été abordé jusqu'ici est son initialisation. Il est nécessaire de stocker de nombreuses références pour établir les différents ordres composant les dépendances et les observations. Ces références doivent donc être initialisées. Cette opération n'est pas négligeable, il faut au moins une référence spécifique à chaque case mémoire.

L'ordre d'une case mémoire étant RAW+WAW+WAR, il est nécessaire de disposer d'un nœud G pour chaque écriture. Il faut donc gérer le cas de ce qu'on appelle l'écriture initiale : il est en effet possible de lire une case mémoire avant qu'elle n'ait été écrite par un processeur même si la valeur retournée n'est pas forcément définie. S'il n'est pas nécessaire de représenter cette écriture par un nœud W, le nœud G qui lui correspond est indispensable pour établir les dépendances entre les lectures éventuelles effectuées avant la première écriture. Cela est illustré dans la FIGURE 4.7.

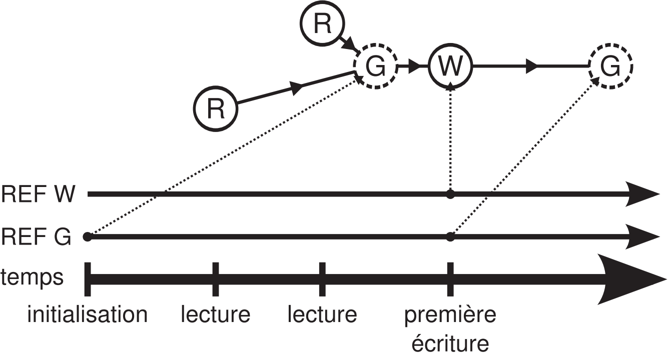


FIGURE 4.7 - Début d'un ordre RAW+WAW+WAR

Il faut logiquement créer un nœud G initial pour chaque ordre RAW+WAW+WAR. De manière générale, il faut créer un nœud pour chaque référence utilisée pour recevoir des prédécesseurs. En pratique, on peut créer les nœuds uniquement à la première utilisation d'une telle référence. Cela permet de ne pas avoir de nœuds inutiles dans le graphe et évite en plus le surcoût que ces nœuds auraient lors de la détection des cycles.

4.3.3 Occupation mémoire en pire cas

L'occupation mémoire qui peut être nécessaire pour l'algorithme est très importante. Nous la divisons en plusieurs catégories.

Architecture La complétion des lectures nécessite de connaître le contenu du système. Il faut en effet savoir quelles sont les écritures dont les données sont dans les caches et les segments mémoire. Pour chacune des cases contenues dans ces composants, il faut donc stocker les références correspondant aux écritures. Il faut bien sûr aussi stocker les propriétés des différents segments et processeurs. L'occupation résultante est donc $O(S + P + (C + M) \times P^{na})$.

Ordres des observations de la mémoire Au sein même de l'algorithme de nombreuses références sont nécessaires. Chaque ordre basique nécessite en effet de conserver quelques références (généralement une ou deux). Comme nous l'avons vu il faut un ordre par case mémoire pour représenter les observations de la mémoire. Ceci représente donc $O(M \times P^{na})$.

Ordres des dépendances du programme Les ordres des dépendances du programme sont encore plus nombreux que ceux des observations. Chaque processeur utilise des ordres globaux mais éventuellement aussi un ordre spécifique à chaque case mémoire. Ces références représentent donc $O(M \times P^{1+na})$.

Graphe Dans le pire des cas, si aucun nœud n'est supprimé, le graphe occupe une place en $O((I + N) \times P^{na})$.

Au total, l'occupation mémoire nécessaire dans le pire des cas est donc :

$$O(S + (C + M \times P + I + N) \times P^{na})$$

4.3.4 Complexité algorithmique

La complexité de l'algorithme présenté dépend de plusieurs facteurs. Nous la présentons en différentes catégories.

Certains coûts sont fortement dépendant des structures de données utilisées. Nous utilisons M_k pour représenter le coût de l'accès aux références d'une case mémoire à partir de son adresse. Nous utilisons M_i pour représenter le coût d'initialisation de la structure de données associée. En fonction des structures de données utilisées (tables, listes, arbres, etc.), M_i et M_k peuvent varier de $O(1)$ à $O(M)$.

De manière similaire nous utilisons C_i et C_k pour représenter le coût d'initialisation et le coût d'accès aux références représentant le contenu d'un cache.

Initialisation de l'architecture Les structures qui correspondent à l'architecture doivent être initialisées. En particulier il faut initialiser les références correspondant au contenu des caches et segments. Les références nécessaires pour gérer l'ordre du programme ne doivent pas être oubliées, d'autant plus qu'il faut éventuellement initialiser les références d'un ordre partiel pour chaque couple (processeur, case mémoire). Ceci représente donc au pire cas $O(P + S + (C_i + M_i \times P) \times P^{na})$ opérations. Comme nous l'avons relevé précédemment, l'initialisation de l'ordre d'une case mémoire peut en pratique être fait lors de la première utilisation de celle-ci à condition d'avoir les structures de données appropriées. M_i peut donc être très réduit en pratique.

Graphe Le traitement des instructions et des accès provoque l'ajout de nombreux nœuds et arcs parfois en plusieurs étapes. Mais chaque accès ou instruction pris séparément a un coût global borné. Ainsi le coût de la création du graphe est en $O((I + N \times (M_k + C_k)) \times P^{na})$. M_k et C_k représente les coûts d'accès aux structures de données.

Cycles La détection des cycles dépend directement de la taille du graphe. Ainsi son coût est en $O((I + N) \times P^{na})$.

Gestion des caches Les mises à jour et invalidations des caches permettent de gérer le contenu des caches et coûtent donc $O(U \times C_k \times P^{na})$. Ces événements sont aussi un moyen de pouvoir dire quand une écriture disparaît. Il est par exemple possible de conserver le nombre de fois qu'une écriture est dans un cache et de mettre à jour ce nombre en fonction de ces événements : quand ce nombre vaut 0 et que l'écriture n'est pas en mémoire centrale, alors elle a disparu.

Au total, la complexité algorithmique est donc :

$$O(P + S + (C_i + M_i \times P + I + N \times (M_k + C_k) + U \times C_k) \times P^{na})$$

4.3.5 Bilan préliminaire

L'algorithme présenté permet de vérifier la consistance mémoire de manière efficace. Il permet de traiter en temps linéaire les instructions et les accès : la complexité est, si l'on considère les ordres de grandeur, $O((I + N \times (M_k + C_k)) \times P^{na})$

M_k et C_k sont très importants dans l'équation car ils sont en facteur de N. Il est donc primordial de choisir des structures de données adaptées. $M_k = O(M)$ n'est par exemple pas raisonnable : une implantation devra faire attention à ne pas dépasser $M_k = O(log(M))$. En utilisant une table pour chaque segment, on peut par exemple facilement obtenir $M_k = O(log(S))$.

La complexité, bien que correspondant au pire cas, est réaliste au contraire de l'occupation mémoire. Nous nous basons en effet sur l'hypothèse que le graphe aura une taille réduite du fait de la suppression des nœuds au fur et à mesure de l'exécution. Si on met le pire cas de côté et qu'on considère que le graphe est de taille T_G, l'occupation mémoire est alors :

$$O(S + (C + M \times P) \times P^{na} + T_G)$$

En faisant l'hypothèse que T_G n'est pas très grand devant M, M n'est pas négligeable. Il devient alors intéressant d'essayer de ne pas avoir une occupation mémoire linéairement dépendante de M.

4.4 Optimisation de l'occupation mémoire

Cette partie concerne l'optimisation de l'algorithme afin de réduire, en particulier, l'occupation mémoire. L'occupation mémoire du pire cas ne peut bien sûr pas être améliorée. Mais cela n'empêche pas d'adapter l'algorithme pour réduire l'occupation que nous appelons dynamique : celle à un moment donné lors de l'exécution de l'algorithme. Nous nous intéressons à supprimer la plus grosse partie de l'occupation mémoire dépendant des paramètres de l'architecture : celle qui dépend du facteur M, le nombre de cases mémoire. Cette partie s'articule autour de quatre points.

Cases mémoire L'augmentation de la taille d'une case mémoire permet de diminuer le nombre d'accès partiels considérés pour chaque accès.

Nœuds isolés De nombreux nœuds sont conservés dans le graphe car ils peuvent encore recevoir des nouveaux prédécesseurs. Néanmoins certains sont inutiles pour la recherche de cycles car ils sont isolés : ils n'ont pas d'arcs (sortant ou entrant).

Références invalides De nombreuses références sont utilisées bien qu'elles ne référencent aucun nœud puisque ces nœuds ont été supprimés.

Contenu de la mémoire Afin de pouvoir gérer la complétion des lectures, il faut conserver des références vers les écritures correspondant aux données dans la mémoire et les caches.

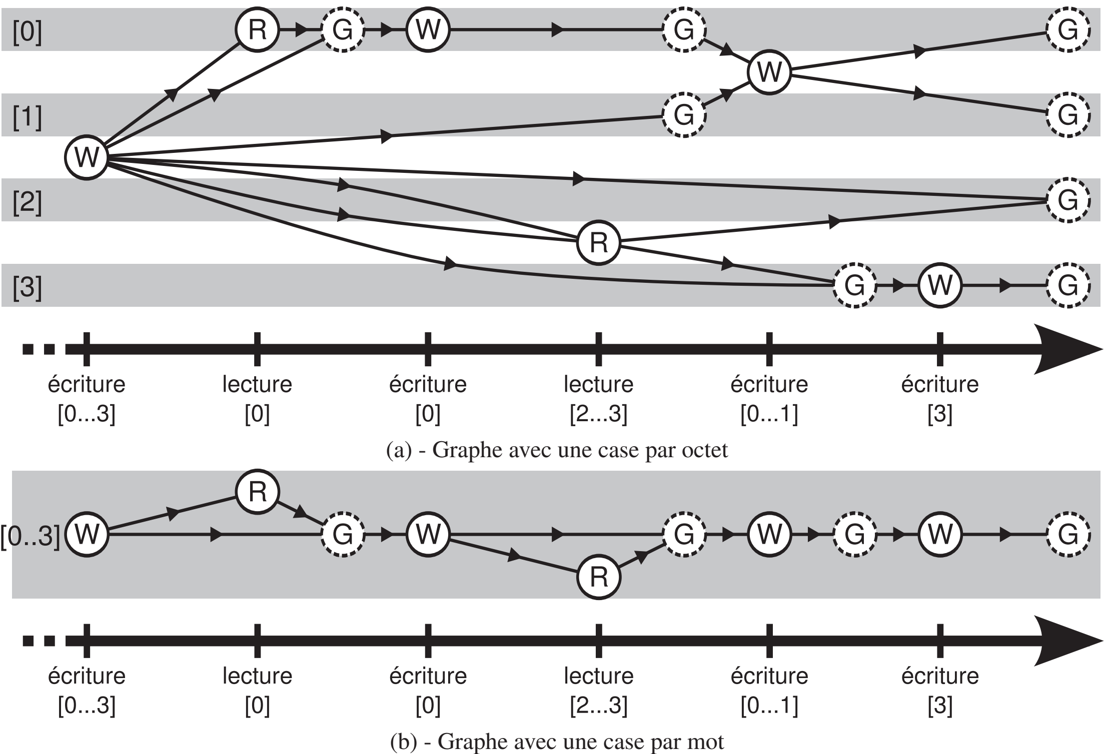
4.4.1 Taille d'une case mémoire

Jusqu'à présent, nous n'avons pas vraiment explicité ce qu'était une case mémoire. Dans le cas général, on peut considérer que c'est le plus petit élément accessible : un octet par exemple. La conséquence directe de l'utilisation de cases mémoire minimales est la séparation d'un accès en plusieurs accès partiels, qui sont gérés indépendamment les uns des autres et ont chacun des nœuds dédiés. Ces nœuds doivent être référencés pour pouvoir y ajouter des arcs, ce qui implique des structures de données volumineuses.

Si on veut représenter exactement les conditions imposées par la consistance mémoire des architectures, les cases mémoire doivent rester petites. Utiliser des cases mémoire plus larges ajoute des dépendances supplémentaires entre des accès qui sont théoriquement indépendants. Considérons par exemple que les cases mémoire sont les mots alignés de 4 octets. Dans ce cas

tout accès, qu'il accède 1 ou 4 octets, est géré de la même manière. Ainsi des écritures à des octets séparés de ce mot vont être ordonnées alors qu'elles ne le seraient pas si les cases mémoire correspondaient à un octet.

La différence entre ces deux cas est illustrée dans la FIGURE 4.8. Cette figure représente les accès au mot correspondant à la plage d'adresse $[0\ldots3]$ composée de 4 octets. La FIGURE 4.8a montre le graphe généré si une case correspond à un octet. La FIGURE 4.8b montre le graphe généré pour les mêmes accès si une case correspond à un mot de 4 octets. On peut remarquer qu'il y a beaucoup moins de nœuds G et d'arcs dans le deuxième graphe. Les nœuds R et W sont beaucoup plus contraints dans le deuxième graphe.



(a) - Graphe avec une case par octet

(b) - Graphe avec une case par mot

FIGURE 4.8 - Impact de la taille d'une case mémoire sur le graphe

D'une certaine manière, si une case mémoire représente un mot entier, alors tout accès à ce mot, même s'il n'accède qu'à une portion, est considéré comme accédant à la totalité de la case. La case mémoire est ainsi traitée comme si elle était atomique. Cela correspond à la notion de largeur de cohérence. L'implantation faite d'une architecture et de son modèle de consistance peut en effet toujours être plus contrainte que spécifiée. En pratique, la cohérence ne se limite même pas à un mot mais concerne les lignes entières de cache.

Nous avons tout intérêt à vérifier le modèle le plus proche possible de ce qui est implanté, surtout si cela réduit le coût algorithmique. Ainsi, il faut, autant que possible, utiliser des cases mémoire larges. Dans notre représentation, nous considérons qu'un segment est une suite contigüe de cases de même taille. Mais chaque segment peut avoir sa propre taille de cases mémoire. On peut généralement classer les segments en deux catégories.

1. Les segments susceptibles d'être mis en cache sont composés de cases de la taille d'une ligne de cache.

2. Les segments non cachés sont composés d'une seule case mémoire de la taille du segment. Ces segments sont souvent aussi des segments correspondants aux périphériques.

L'augmentation de la taille des cases mémoire entraîne une diminution de leur nombre. Ainsi les grandeurs M et C qui représentent un espace mémoire donné sont réduites.

4.4.2 Nœuds isolés

4.4.2.1 Le problème des nœuds isolés

La politique de suppression des nœuds que nous utilisons dans l'algorithme permet de supprimer les nœuds quand il est certain qu'ils ne seront pas impliqués dans un cycle. Néanmoins certains nœuds sont susceptibles de rester dans le graphe tout en n'étant connectés à aucun autre nœud.

Ces nœuds sont très courants, en particulier dans les ordres spécifiques à une case mémoire, que ce soit l'ordre des observations de la mémoire ou un ordre spécifique à un couple d'un processeur et d'une case mémoire servant à l'établissement de l'ordre du programme.

Ainsi, à partir du moment où une case a été accédée, un nœud G sera forcément dédié à l'ordre de la mémoire pour cette case mémoire et pour le restant de l'exécution. Ce nœud correspond au nœud G de la dernière écriture (éventuellement l'écriture dite initiale) qui a accédé à cette case. Il ne peut être supprimé car une éventuelle lecture à cette case provoque l'ajout d'un arc vers ce nœud. S'il n'y a pas d'accès pendant un certains laps de temps, alors les nœuds précédant celui-ci seront supprimés du graphe, mais le dernier nœud G perdurera. On rappelle dans la FIGURE 4.9 l'ordre des accès d'une case mémoire. Le nœud G dont il est question est celui le plus à droite dans le graphe.

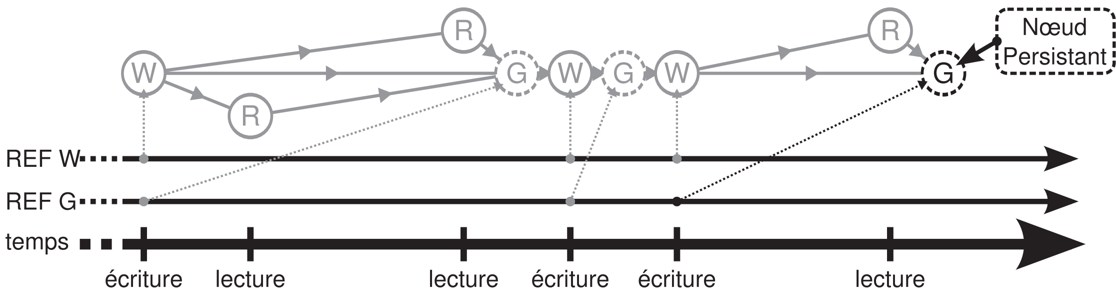


FIGURE 4.9 - Persistance d'un nœud G dans l'ordre d'une case mémoire.

Le graphe va donc inévitablement grossir à chaque fois qu'une nouvelle case mémoire est accédée. Cette constatation est valable pour chaque graphe partiel qui implique de conserver un nœud G pour y ajouter des prédécesseurs jusqu'à un éventuel futur événement. Cela comprend donc les ordres observés de chaque case mémoire, les ordres des dépendances du programme associés à un couple (processeur, case mémoire) et les ordres des dépendances du programme associés à seulement un processeur. Ceux qui concernent uniquement un processeur ne sont pas très significatifs. Le nombre de cases mémoire est en effet supérieur de plusieurs ordres de grandeur au nombre de processeurs.

Quand un tel nœud G est créé, des prédécesseurs lui sont généralement ajoutés. Mais du fait de la suppression des nœuds de proche en proche, ceux-ci vont finir par être supprimés. Et si aucun successeur n'est ajouté à ce nœud, il va finir par devenir *isolé*. Nous disons qu'un nœud est isolé s'il n'a ni prédécesseur ni successeur. Si un tel nœud reste dans le graphe, c'est qu'il peut encore recevoir des prédécesseurs. Cela n'empêche pas que tant qu'il est isolé, il est inutile du point de vue de la VCM : il ne peut pas être dans un cycle.

4.4.2.2 Suppression temporaire des nœuds isolés

La solution consiste donc à supprimer temporairement ces nœuds, par exemple en détectant dans la fonction de *suppression* (cf. l'ALGORITHME 4.2 à la page 63) si un successeur du nœud supprimé devient isolé. Il suffit ensuite d'invalider temporairement les références vers ce nœud pour indiquer que le nœud a été supprimé temporairement. Lorsqu'une référence est utilisée le nœud est alors recréé et l'ensemble des références vers ce nœud est remise à jour.

Le problème de cette solution est son coût temporel. D'une part, il faut parcourir toutes les références d'un nœud à chaque fois qu'un nœud est temporairement supprimé. D'autre part, cela nécessite de conserver un lien entre toutes les références du même nœud pour pouvoir remettre en place l'ensemble des références lorsqu'une d'entre elles est utilisée. Ainsi à chaque fois qu'un nœud isolé est supprimé ou remis, cela coûte $O(\text{nombre de références du nœud})$ et si le nœud change régulièrement de statut cela peut coûter très cher.

La solution est en fait de ne supprimer les nœuds isolés que s'ils n'ont qu'une seule référence. De cette manière, le coût de la suppression temporaire $O(1)$ peut être associé à la suppression de l'arc qui rend le nœud isolé. De même sa remise en place peut être associée à l'ajout de l'arc ou de la référence qui la provoque.

De plus, un nœud est de manière générale référencés une seule fois : par la référence associée au graphe partiel auquel il appartient. Cette condition supplémentaire n'est donc pas très contraignante. Seuls les nœuds G utilisés pour l'ordre de la mémoire sont éventuellement référencés à plusieurs endroits quand un accès est présent dans plusieurs caches ou dans un cache et la mémoire. Mais cela représente un nombre limité d'accès, $O(C)$, et donc de nœuds.

Une autre condition qui rend impossible la suppression temporaire d'un nœud est si celui-ci contient des informations importantes qui ne peuvent être reconstituées si elles sont perdues. Le nombre de marqueurs dont dispose un nœud isolé fait partie de ces informations. Dans la plupart des cas, on supprimera un nœud s'il n'a qu'un unique marqueur.

Un exemple d'informations importantes concerne non pas la VCM mais la recherche de la cause d'une violation du MCM. Si l'on veut pouvoir analyser le graphe pour y trouver la cause d'une violation, on peut par exemple vouloir conserver des informations sur l'instruction ayant provoqué l'ajout de certains nœuds. Parmi ces informations, la plus flagrante est le type du nœud (R, W, G, etc.) qui ne sert dans ce document qu'à faciliter la lecture des graphes et des algorithmes.

Les nœuds isolés ne pouvant pas être supprimés peuvent par contre être mis de côté pour éviter d'être testés lors de la détection des cycles.

4.4.3 Références invalides ou isolées

L'éviction des nœuds isolés permet surtout de réduire la taille du graphe au minimum : seuls les nœuds étant liés à au moins un autre nœud par un arc sont représentés dedans. Mais une référence vers un nœud supprimé temporairement ou définitivement bien qu'invalide a un impact similaire sur l'occupation mémoire que le nœud : un nœud coûte $O(1)$ mais une référence aussi.

4.4.3.1 Références utilisées pour l'ordre du programme

Par exemple, si des graphes partiels spécifiques à chaque case mémoire sont nécessaires pour construire l'ordre d'une SI, une fois les nœuds isolés supprimés, la majorité des références utilisées pour ces graphes seront invalides. Seules les références concernant les cases mémoire récemment accédées ne le seront pas.

Les structures de données associées à un processeur contiennent éventuellement un grand nombre de références car de nombreux graphes partiels doivent être construits, en particulier si le MCM est très relâché. Ces références sont en effet une des principales sources de l'occupation mémoire $O(M \times P)$. Il est donc primordial de supprimer les références invalides.

De manière similaire aux nœuds isolés, ces références seront reconstruites lorsqu'elles seront nécessaires. De même que pour les nœuds, si les références contiennent des informations additionnelles importantes, elles ne doivent pas être supprimées. En particulier, il faut pouvoir lors de la reconstruction savoir pourquoi la référence était invalide : le nœud référencé était-il temporairement ou définitivement supprimé ? Pour toutes les références que nous avons utilisées dans les exemples de graphes partiels de l'ordre du programme, cette information est connue car les références conservent toujours le même genre de nœud. Une référence conserve en effet :

1. soit un nœud qui reçoit uniquement des successeurs et qui sera supprimé définitivement dès que possible,
2. soit un nœud qui reçoit éventuellement des prédécesseurs et qui a donc un marqueur et qui ne sera donc pas supprimé définitivement.

Par exemple, si l'on désire vérifier que les valeurs lues et écrites correspondent bien lors de la complétion d'un accès, il faut conserver dans les accès partiels la valeur courante ou lue de la case mémoire, ce qui permettra de vérifier qu'elles concordent lors des complétions. Cela empêchera donc toute suppression temporaire des accès partiels. Une telle vérification n'est néanmoins nécessaire que si on ne fait pas confiance en la véracité des événements traités par l'algorithme ou en la transmission des valeurs des accès dans les communications dans le SMPMP.

La suppression de ces références impacte néanmoins fortement les structures de données de l'algorithme utilisées pour stocker ces références. Bien qu'il soit très rapide, il est par exemple inadapté d'utiliser un tableau de références indexé directement par les adresses des cases mémoire pour les stocker si l'on veut que la suppression des références soit utile. Dans ce cas, une structure d'arbre ou de table de hachage sera privilégiée.

4.4.3.2 Références utilisées pour l'ordre de la mémoire : les accès partiels

Dans l'algorithme décrit, il n'y a pas de références explicites utilisées pour construire l'ordre de la mémoire. Comme nous l'avons dit dans la section 4.2, les accès partiels peuvent néanmoins être considérés comme des références.

Ces accès partiels sont utilisés pour les événements de complétion et de disparition. Ceux contenus dans les caches et les segments doivent être conservés. Comme un accès partiel peut être présent dans plusieurs caches, ils sont peut-être même dupliqués.

Néanmoins, la plupart d'entre eux sont uniquement présents dans leur segment respectif car les caches et les tampons intermédiaires ont une capacité limitée. Par exemple, le nœud W de la dernière écriture à une case mémoire finira par être supprimé et son G deviendra isolé quand les nœuds des lectures lisant cette écriture seront supprimés. Lorsque les nœuds G d'un tel accès partiels sont supprimés (car isolés), il est très intéressant de supprimer l'accès partiel. De cette manière, on peut espérer supprimer temporairement la majorité des accès partiels contenus dans les segments.

De manière similaire aux références des graphes de l'ordre du programme, les structures de données doivent être adaptées et seuls les accès partiels ne contenant aucune information importante peuvent être supprimés. Dans l'algorithme, des informations telles que des références additionnelles ou des étiquettes sont utilisées dans les accès partiels. En particulier les étiquettes utilisées pour gérer les accès spéciaux (cf. la section 4.2.6.1) sont cruciales pour le traitement de ceux-ci.

4.4.4 Bilan des optimisations

Les diverses optimisations présentées dans cette partie, et principalement la suppression des références invalides, permettent de supprimer les structures de données qui ont un coût proportionnel au plus important des paramètres architecturaux : le nombre de cases mémoire, M. L'occupation mémoire, *dynamique*, à un instant de l'algorithme est ainsi estimée à :

$$MEM_{\text{dynamique}} = O(S + (C + K_S.P) \times P^{na} + T_G)$$

T_G dans cette équation représente la taille du graphe utile : c'est à dire le nombre d'accès partiels ayant des nœuds dans le graphe (en n'incluant pas ceux dont les nœuds et références ont été supprimés temporairement). K_S représente un coût statique qui est introduit par l'utilisation de structures spécifiques pour conserver l'état des segments de la mémoire. Un tableau entrainera $K_S = M$, une table de hachage K_S = constante, un arbre $K_S = 1$, etc. .

L'expression de la complexité algorithmique n'est pas changée. Le traitement d'un accès est néanmoins plus lourd puisqu'il faut gérer la suppression et la restauration des nœuds et des références.

$$\text{Complexité} = O(P + S + (C_i + M_i \times P + I + N \times (M_k + C_k) + U \times C_k) \times P^{na})$$

L'occupation mémoire en pire cas reste inchangée.

$$MEM_{\text{pire cas}} = O(S + (C + M.P + I + N).P^{na}).$$

On remarquera que si $K_S = M$ et si $T_G = (I+N) \times P^{na}$ dans l'occupation mémoire, on retrouve l'occupation mémoire en pire cas.

En supprimant les parties négligeables en prenant en compte les différents ordres de grandeur ($\{P, S\} \ll C \ll M \ll \{I, N, U\}$), on obtient :

$$\text{Complexité} = O((I + N.(M_k + C_k) + U.C_k) \times P^{na}),$$

$$MEM_{\text{dynamique}} = O((C + K_S.P) \times P^{na} + T_G)$$

et

$$MEM_{\text{pire cas}} = O((I + N) \times P^{na}).$$

4.5 Conclusion sur la vérification de la consistance mémoire

Dans ce chapitre, nous avons présenté un algorithme de vérification dynamique de la consistance mémoire passant à l'échelle. Celui-ci est adapté à une exécution en parallèle de l'exécution du programme testé. Cela permet donc d'éviter le stockage de nombreuses informations qui empêchent d'utiliser un algorithme non dynamique lors d'exécutions longues. Il est ainsi adapté à une utilisation pendant la simulation d'un système muti-processeurs à mémoire partagée pour vérifier que ce dernier exécute son programme en respectant un certain modèle de consistance mémoire.

Sa complexité est linéaire en fonction d'événements observables pendant la simulation :

1. l'exécution d'une instruction,
2. la complétion d'un accès à la mémoire, et
3. la disparition des écritures dans le système.

L'occupation mémoire est de plus modérée. À part la partie du graphe strictement utile qui est inévitable car elle ne contient que des nœuds ayant au moins un arc, les structures de données nécessaires dépendent de paramètres architecturaux relativement petits : le nombre de segments mémoire, de processeurs et la taille des caches.

L'algorithme peut être utilisé quel que soit l'atomicité des écritures tant que le modèle de consistance mémoire vérifié inclut la cohérence mémoire. Si les écritures ne sont pas atomiques, il y a néanmoins un surcoût d'un facteur du nombre de processeurs à la fois en complexité et en occupation mémoire.

Les événements de l'algorithme correspondent à des opérations réelles se produisant dans le système muti-processeurs à mémoire partagée : l'exécution d'une instruction, la complétion d'un accès et les mises à jour des caches. Ils sont observables, en particulier lors de l'utilisation d'un prototype virtuel. Ils ne sont néanmoins pas indépendants : le traitement de l'exécution par l'algorithme produit des objets, appelés accès partiels. Ces accès partiels doivent être donnés à nouveau à l'algorithme pour le traitement de leur complétion et de leur disparition. Il faut pouvoir transmettre ces objets entre les opérations effectuées concernant le même accès dans le système muti-processeurs à mémoire partagée.

5 Environnement de trace et d'analyse pour la simulation

Comme nous l'avons vu dans les précédents chapitres, et en particulier dans le chapitre 4 : pour réaliser efficacement une analyse complexe, telle que la vérification de la consistance mémoire, il faut disposer de beaucoup d'informations. Dans ce chapitre, nous définissons ainsi un environnement permettant l'analyse efficace du déroulement d'une simulation d'un système muti-processeurs à mémoire partagée.

Nous définissons précisément dans la première partie le cahier des charges que doit remplir cet environnement. Celui-ci est subdivisé en un système de trace et une bibliothèque d'analyse.

Dans la deuxième partie nous explicitons le système de trace proposé. Celui-ci permet d'extraire des informations à propos de différents points du système muti-processeurs à mémoire partagée simulé tout en permettant d'inclure les recoupements entre ces différentes informations.

La troisième partie est ensuite dédiée à la définition d'une bibliothèque permettant de construire des analyses complexes en utilisant les informations tracées. Ce système utilise en particulier les recoupements présents dans la trace pour permettre de traiter les informations efficacement.

Nous concluons finalement ce chapitre dans la quatrième partie.

Chapitre 5

5.1 Objectifs de l'environnement de trace et d'analyse 84
5.1.1 Extraction d'informations d'un SMPMP 84
5.1.2 Objectifs 85
5.1.3 Environnement unifié de trace et d'analyse 85
5.2 Système de trace 86
5.2.1 Exemple de SMPMP 86
5.2.2 Principe général 86
5.2.3 Relations entre les événements 87
5.2.4 Conclusion sur le mécanisme de trace 90
5.3 Bibliothèque générique d'analyse 90
5.3.1 Principe général 90
5.3.2 Ordonnancement des traitants 92
5.4 Conclusion sur l'environnement de trace et d'analyse 95

5.1 Objectifs de l'environnement de trace et d'analyse

5.1.1 Extraction d'informations d'un SMPMP

5.1.1.1 Extraction depuis un système réel

Dans un système réel, les moyens d'obtenir de l'information sont très limités. Seule une observation des entrées et sorties d'une puce est en effet possible. Pour un système intégré, tel un MPSoC, elles sont extrêmement limitées. Afin de palier à ce problème, et pour permettre d'analyser le comportement d'un système intégré, une méthode universelle est utilisée : la chaîne de scan. Celle-ci permet en particulier de figer le système et d'extraire bit par bit les données de l'ensemble des registres intégrés à cette chaîne. C'est un processus extrêmement lent et inadapté à des analyses de haut niveau.

En utilisant une interface similaire, des méthodes ont ainsi été mises en place pour extraire des informations sur les processeurs afin de pouvoir déboguer les programmes exécutés. En particulier des mécanismes permettant de paramétrer quelles informations sont tracées en temps réel par un port de sortie dédié du système intégré. Cette trace peut être ensuite collectée par un système extérieur et analysée. Parmi ces systèmes, on trouve par exemple le système *EJTAG* de l'architecture MIPS et le système *Embedded Trace Macrocell* de l'architecture ARM. Pour les systèmes massivement parallèles, il y a néanmoins des problèmes à cause du goulet d'étranglement que forment le ou les ports de sortie utilisés pour la trace [Gao+12].

5.1.1.2 Extraction depuis un prototype virtuel

Dans un prototype virtuel, les mêmes systèmes peuvent être utilisés. Mais un prototype virtuel a l'avantage de ne virtuellement pas avoir de limites quant à la quantité d'informations qu'il est possible de tracer. En utilisant des portes dérobées, il est possible de gagner en efficacité par rapport aux mécanismes réels utilisés pour l'analyse. Par exemple il est possible d'intégrer directement un serveur pour le débogueur *GDB* dans les modèles des processeurs [Pou+09], évitant ainsi de simuler un mécanisme tel l'*EJTAG* pour obtenir un résultat similaire.

Une manière facile d'obtenir des informations depuis un prototype virtuel est de tracer, sans conditions, l'ensemble des signaux simulés du système. Ainsi *SystemC* [SystemC] intègre des mécanismes permettant de le faire. Cela a néanmoins un désavantage majeur, la totalité des informations disponibles sont tracées et il est nécessaire d'effectuer un filtrage de celles-ci avant de pouvoir analyser celles qui sont utiles.

Par ailleurs, même si énormément d'informations sont tracées, il faut probablement les recouper ensuite entre elles pour pouvoir les analyser. Ainsi dans le travail de Rambo, Henschel et Santos [RHS12], le réordonnancement effectué par un processeur est vérifié entre deux points du pipeline de celui-ci. Pour cela, une trace est extraite de chacun de ces deux points contenant les n accès effectués. Vérifier le réordonnancement entre les deux traces peut être fait en $O(n)$, mais le coût total est $O(n^6)$ car il faut associer en paire les accès des deux traces avant de pouvoir le faire.

Afin de pouvoir analyser efficacement ce qu'il se passe lors d'une simulation d'un SMPMP, il est donc nécessaire de disposer d'un système de trace global utilisé dans l'ensemble du système. Il faut en effet pouvoir extraire des informations concernant différents mécanismes du SMPMP simulé : l'exécution des instructions par les processeurs, la transmission des communications par le système de communications, etc.. Afin de pouvoir être analysées efficacement, il ne faut pas reporter dans l'analyse le soin de recouper les informations extraites.

5.1.2 Objectifs

L'extraction d'informations se fait dans l'optique de réaliser des analyses complexes concernant le déroulement d'une simulation d'un SMPMP. L'algorithme de VCM décrit dans les chapitres précédents est un exemple d'une telle analyse. Des informations sont nécessaires à propos de ce qu'il se produit dans beaucoup de composants matériels du SMPMP : les processeurs à propos des instructions exécutées, les segments mémoire ou les périphériques à propos des accès à ceux-ci, les caches à propos des données qu'ils contiennent, etc.. Mais des informations sont aussi nécessaires entre ces différents composants puisque, dans cet algorithme, on cherche à relier l'émission d'un accès lors d'une instruction avec la complétion de celui-ci : un accès n'est pas simplement complété par un segment mémoire, il est émis par un processeur puis transféré jusqu'à sa destination.

Ainsi nous recherchons trois objectifs dans l'extraction d'informations depuis la simulation :

– des informations doivent pouvoir être extraites à propos d'événements se produisant en différents points du prototype virtuel ;
– des informations à propos des relations entre ces différents événements doivent pouvoir être extraites aussi ; et
– ces informations doivent être extraites en minimisant l'impact sur la simulation, en particulier l'impact fonctionnel.

5.1.3 Environnement unifié de trace et d'analyse

Dans la suite de ce chapitre, nous décrivons donc un environnement permettant de réaliser ces analyses. Il est représenté de manière très générale dans la Figure 5.1. Le principe général de celui-ci est d'extraire des informations de la simulation d'un SMPMP et de fournir un support pour réaliser des analyses de ces informations.

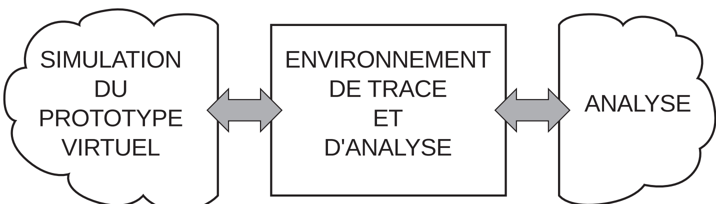


Figure 5.1 - Vue globale de l'architecture de l'environnement

Une des caractéristiques majeures de l'extraction d'informations est l'impact de cette extraction sur le système cible. En particulier le degré de l'impact sur le comportement fonctionnel du système est crucial. Dans un système réel, lorsqu'on ne désire pas modifier ce comportement, seule une observation des signaux disponibles est possible. Afin d'augmenter les informations disponibles et leur pertinence, les systèmes intègrent des fonctionnalités dédiées uniquement à générer des traces qui peuvent alors être collectées par un système externe. Ces fonctionnalités ont un coût d'intégration et sont donc limitées, elles se concentrent généralement sur les processeurs.

Dans un système virtuel, les mêmes observations peuvent être réalisées, et en particulier avec les même fonctionnalités intégrées de trace. Mais cela implique de travailler avec les informations observables et donc du niveau d'abstraction de la modélisation du prototype virtuel. Pour éviter une reconstruction fastidieuse vers des informations de plus haut niveau, nous nous basons sur une particularité des modèles de simulation formant les prototypes virtuels : ils peuvent dans de nombreux cas être modifiés sans que leur comportement fonctionnel ne le soit. Dans la suite de ce chapitre, nous appelons cela l'instrumentation des modèles.

Cette instrumentation a bien sûr un impact, au moins sur la vitesse de simulation du prototype virtuel. Mais il est en contrepartie possible d'extraire des informations bien plus ciblées de celles qui sont simplement observables.

5.2 Système de trace

5.2.1 Exemple de SMPMP

Afin d'illustrer les différents points développés dans ce chapitre, nous nous appuyons sur des exemples concrets. Dans ce cadre, nous utilisons à chaque fois le même SMPMP en exemple. Celui-ci est représenté dans la FIGURE 5.2. Il comprend plusieurs processeurs ayant chacun son propre cache pour les données. La partie concernant les instructions n'est pas étudié dans les exemples. Chaque cache est relié à un système de communication commun auquel est relié un banc de mémoire partagée.

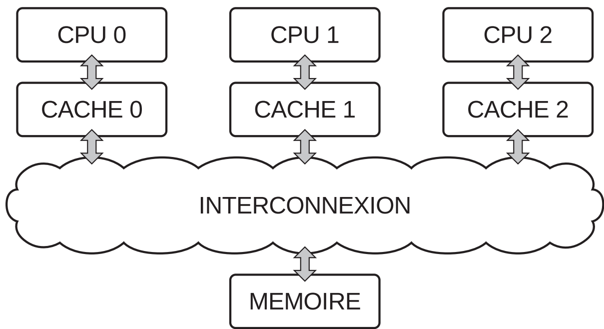


FIGURE 5.2 - Système matériel utilisé pour les exemples

5.2.2 Principe général

5.2.2.1 Utilisation de ports de trace

Un système multi-processeurs est un système massivement parallèle. La modélisation qui en est faite dans un prototype virtuel l'est aussi. Celle-ci est en effet composée de nombreux composants matériels qui réalisent chacun différentes opérations en même temps pendant la simulation. Ce parallélisme est très souvent virtuel, au moins en partie, mais nous ne faisons aucune hypothèse sur celui-ci.

Ainsi dans le but de récolter des informations en différents points du prototype virtuel, plusieurs « ports de trace » sont utilisés aux points utiles. Du point de vue de l'environnement, le prototype virtuel est uniquement composé des parties traçant des informations à travers un port de trace. Les parties ne traçant aucune information sont inconnues et ignorées de l'environnement. Chacune de ces parties est appelée un composant qui est associé à un port de trace. Ces composants forment une vue abstraite du prototype virtuel simulé. Les ports de trace sont indépendants les uns des autres et sont séquentiels. Chaque port de trace permet ainsi à son composant de tracer une séquence ordonnée d'*événements* contenant des informations. L'indépendance des ports de trace est nécessaire pour ne pas ajouter de contraintes de synchronisation si la simulation est faite de manière parallèle.

5.2.2.2 Architecture générale

La FIGURE 5.3 représente une vue d'ensemble du mécanisme de trace. Les événements sont transmis depuis la simulation vers l'environnement à travers les différents ports de trace. Ils

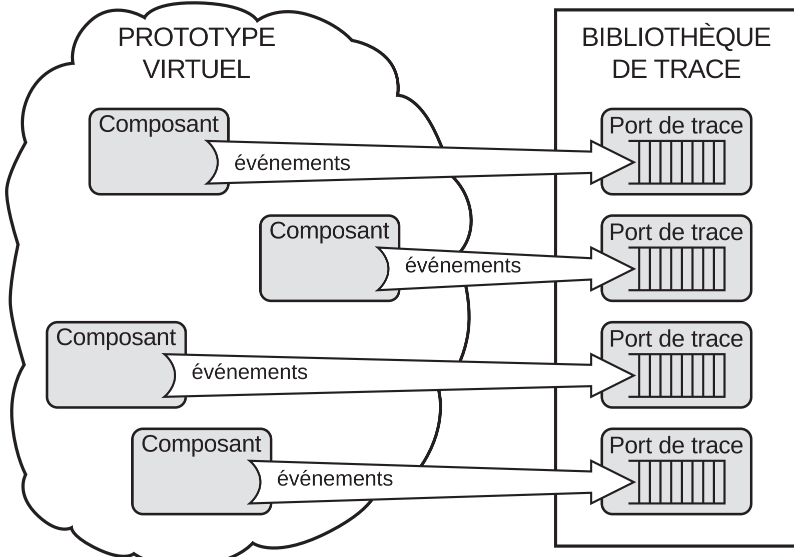


FIGURE 5.3 - Architecture du mécanisme de trace

peuvent ensuite être analysés en parallèle de la simulation afin de ne pas l'impacter plus que ce qui est nécessaire pour extraire les événements. Éventuellement, les événements peuvent être stockés pour une analyse ultérieure.

Ce à quoi correspond un composant, et le port de trace associé, dépend à la fois du prototype virtuel et des objectifs de l'analyse. De même le contenu des événements est spécifique aux objectifs visés.

Par exemple, si l'objectif est d'étudier le logiciel exécuté par le système simulé, c'est ce que font les processeurs qu'il va être nécessaire de tracer. Une possibilité est alors de représenter chaque processeur par un composant générant, à travers son port de trace, un événement par instruction exécutée. Cela n'est néanmoins possible que si la modélisation d'un processeur dans le prototype virtuel le permet : si celui-ci est modélisé à grain très fin, il sera peut-être nécessaire d'utiliser plusieurs composants (et donc plusieurs ports de trace) par processeur pour obtenir les informations voulues. Ces ports de trace peuvent représenter par exemple les différents étages du pipeline où des informations voulues sont accessibles.

Pour la VCM, il faut tracer des événements au moins au niveau des processeurs (pour avoir des informations sur les accès générés) et au niveau des segments mémoire (pour avoir des informations sur la complétion des accès).

5.2.3 Relations entre les événements

5.2.3.1 Des relations réelles de cause à effet

Chaque port trace séquentiellement des événements indépendamment des autres ports. Néanmoins les événements de différents composants peuvent être reliés pour une raison de cause à effet. Par exemple, la mise à jour d'une cellule mémoire par un composant de mémorisation est reliée à la requête d'écriture initiée par un processeur. Pour la VCM, ces deux informations sont nécessaires mais ne suffisent pas : il faut savoir quelle est la requête qui provoque la mise à jour.

Il faut donc un système permettant de pouvoir reconstruire les relations de cause à effet. Ces relations sont très dépendantes du contexte dans lequel elles sont utilisées. En particulier, un événement peut être la cause ou la conséquence de plusieurs événements qui ne proviennent pas forcément des mêmes composants.

La Figure 5.4 illustre ainsi un scénario de notre plate-forme d'exemple. Le scénario présenté dans celle-ci correspond à l'utilisation d'un protocole de cohérence de type *write through invalidate* : chaque écriture atteint le segment mémoire ciblé. Néanmoins un regroupement des accès contigües est réalisé pour réduire l'utilisation du système de communication. Dans cet exemple, on peut voir plusieurs écritures du CPU_0 qui sont regroupées en une seule requête envoyée au segment mémoire qui envoie alors deux invalidations aux caches des CPU_1 et CPU_2. Les relations de cause à effet sont représentées par des flèches inversées (c.-à-d. allant de la conséquence vers la cause). La requête d'écriture envoyée par le cache possède plusieurs causes : les requêtes d'écriture du CPU_0. La mise à jour effectuée par le segment mémoire a plusieurs conséquences : les invalidations des caches 1 et 2.

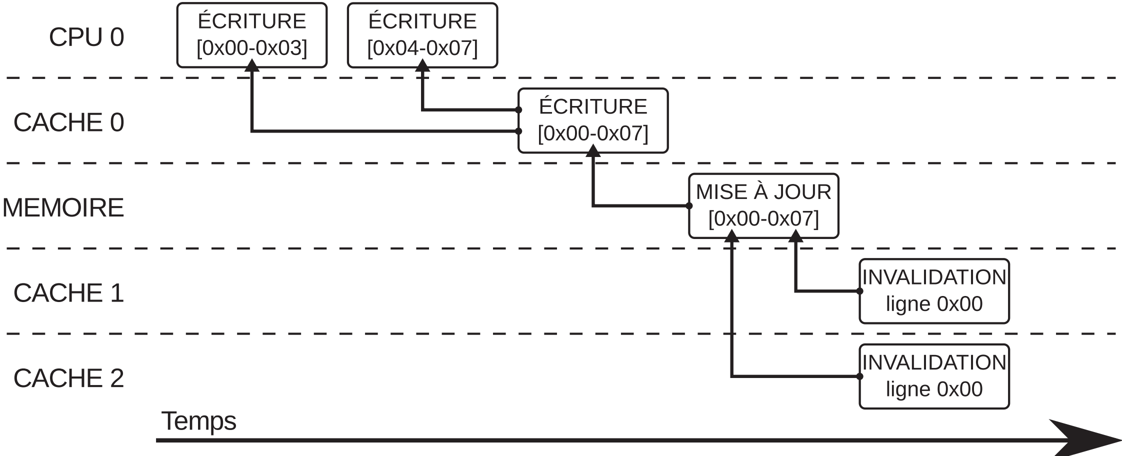


Figure 5.4 - Exemples de relations de cause à effet

La figure est présentée selon l'ordre temporel logique allant de gauche à droite. Néanmoins à partir des événements tracés, aucun temps global ne peut être reconstruit (à moins que les événements contiennent les informations nécessaires). Les seuls ordres temporels extraits sont locaux à chaque composant : les événements d'un port de trace sont séquentiels.

5.2.3.2 Utilisation d'identifiants

Si certaines relations peuvent être retrouvées facilement, ce n'est pas forcément le cas pour toutes. Par exemple, si la requête envoyée depuis un cache vers la mémoire possède un numéro de paquet, il peut être inséré dans les événements et permettre de reconstruire la relation *a posteriori*. Néanmoins afin d'éviter d'avoir à effectuer une coûteuse opération de recherche et reconstruction des relations entre les événements, il est nécessaire que les relations de cause à effet puissent être présentes dans les traces.

Afin de gérer des relations, il est nécessaire de pouvoir identifier les événements. Le système de trace associe, à cette fin, un unique identifiant à chaque événement. Pour indiquer les relations, nous proposons qu'un événement contienne les identifiants des événements dont il est la conséquence : un événement « conséquence » référence un événement « cause » (c'est pour cette raison que les flèches sont inversées dans la Figure 5.4). Si la relation entre deux événements est réelle, il y a nécessairement une transmission d'informations dans le prototype virtuel entre le composant émettant l'événement « cause » et le composant émettant l'événement « conséquence ». Il n'y a par contre aucune garantie sur l'existence d'une communication depuis le composant émettant l'événement « conséquence » vers le composant émettant l'événement « cause ».

Il est donc possible de transmettre des informations supplémentaires en modifiant les canaux de communication utilisés. Afin de ne pas modifier le comportement du prototype virtuel, un canal

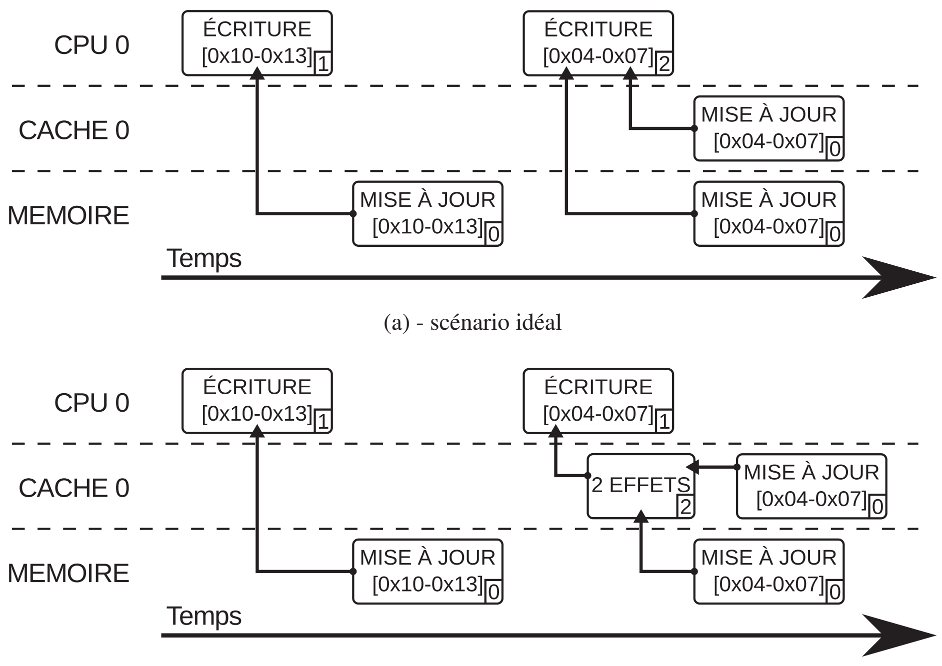
parallèle au canal de communication du prototype doit être mis en place. Ce canal sert à transmettre l'identifiant en même temps que les données habituelles. Il en résulte un « élargissement » du canal de communication initial. Par exemple, ce canal peut consister à ajouter des signaux transportant les identifiants à côté de ceux transportant les données dans un système de communication.

5.2.3.3 Absence de réciprocité des relations

Les relations ne sont pas réciproques, seul le sens conséquence vers cause est en effet représenté. Les différents ports de trace étant indépendants, il n'est pas impossible qu'un événement soit traité avant d'autres événements qu'il a causés. Ainsi le système de trace ne peut pas savoir si tous les événements causés par un autre ont été reçus. C'est une information qu'il est capital d'obtenir pour éviter, par exemple, de devoir conserver des informations sur un événement uniquement pour le cas où une nouvelle conséquence de cet événement serait traitée. Dans cette optique, les événements tracés doivent contenir l'information du nombre d'événements dont ils sont la cause.

Cette contrainte est forte mais pas bloquante. Dans de nombreux cas, il n'est en effet pas possible pour un composant, générant un événement, de savoir exactement combien d'autres événements seront provoqués parce qu'il ne dispose pas de cette information : le nombre de conséquences peut être dynamique. Néanmoins, celui-ci sait vraisemblablement qu'il y en aura au moins un certain nombre. Une solution est alors d'utiliser des événements intermédiaires pour « changer » le nombre de conséquences d'un événement. L'événement intermédiaire est une conséquence d'un événement initial et peut indiquer avoir autant de conséquences que nécessaire.

La Figure 5.5 illustre ce cas. Dans cet exemple, une écriture d'un processeur peut éventuellement provoquer une mise à jour du cache du processeur (si celui-ci possède la ligne accédée). S'il n'est pas possible d'indiquer quel sera le nombre de conséquences de l'écriture initiale, il est possible de mettre 1 par défaut car il y aura au moins la mise à jour de la mémoire. Un événement intermédiaire est alors nécessaire pour indiquer qu'il y aura en fait 2 conséquences. Dans cette figure, le nombre de conséquences de chaque événement est indiqué dans son coin inférieur droit.



(a) - scénario idéal

(b) - utilisation d'un événement intermédiaire

Figure 5.5 - Événement ayant un nombre dynamique de conséquences

5.2.4 Conclusion sur le mécanisme de trace

Ce mécanisme de trace permet d'extraire des informations du prototype virtuel par le biais d'événements générés en différents points, appelés composants, de ce prototype. Ces composants sont globalement indépendants les uns des autres ainsi que les événements qu'ils génèrent. Néanmoins comme nous l'avons soulevé, il est nécessaire de mettre en relation les événements de différents composants pour pouvoir analyser certains problèmes. Un système d'indication des relations de cause à effet est utilisé à cette fin. Ce système nécessite néanmoins de transmettre des identifiants spécifiques à l'environnement de trace avec les communications réelles se déroulant entre les composants matériels du prototype virtuel.

Les définitions de ce que représentent les composants et les événements sont étroitement liées avec le prototype virtuel et les objectifs de l'analyse. Idéalement les composants correspondent au niveau d'abstraction le plus haut permettant de réaliser les analyses voulues. L'implantation du prototype virtuel peut néanmoins rendre cette abstraction trop compliquée à mettre en place. En pratique les composants représentés dépendent en effet des possibilités existantes pour collecter les informations qui forment les événements. Il peut en effet être plus simple et efficace de subdiviser certains composants idéaux en plusieurs composants. Ces composants tracent alors des événements reliés par des relations de cause à effet permettant d'associer ensuite leurs contenus.

5.3 Bibliothèque générique d'analyse

Dans cette partie nous définissons un environnement d'analyse à partir des traces collectées par le mécanisme précédemment présenté. Cet environnement a pour objectif de permettre une analyse du système simulé complet : c'est à dire portant sur les événements générés par l'ensemble des composants. Les relations de cause à effet, introduites explicitement dans la trace, permettent d'envisager une gestion efficace de celles-ci.

Comme nous l'avons décrit dans la partie précédente, un événement est identifiable de manière unique et contient au moins ces deux informations :

- le nombre d'événements dont il est la cause, et
- les identifiants des événements dont il est la conséquence.

5.3.1 Principe général

Cet environnement générique se présente sous la forme d'une bibliothèque permettant de construire un système d'analyse spécifique traitant les événements générés par les ports de trace des composants. Ce système d'analyse est donc formé de deux parties : une bibliothèque générique intégrée à l'environnement et un programme spécifique utilisant cette bibliothèque. Ce programme, développé spécifiquement pour l'analyse à effectuer, est appelé par la suite « analyseur ».

Le principe général consiste à effectuer une analyse non pas avec une unique fonction centralisant la totalité de l'analyse mais de manière « distribuée » sur les événements : chaque événement est traité par une ou plusieurs fonctions que nous appelons des « traitants ». Ces traitants sont exécutés et ordonnancés par la bibliothèque en fonction de règles paramétrées par l'analyseur.

5.3.1.1 Les composants

La première étape de l'analyse correspond à une déclaration, de la part de la bibliothèque des différents composants traçant des événements via leurs ports de trace respectifs. Ces composants réalisent une abstraction du SMPMP simulé et peuvent représenter des éléments complètement différents : par exemple des processeurs, des caches, des segments mémoire, des éléments du sys-

tème de communication, etc.. Il est donc nécessaire d'indiquer à l'analyseur quel port de trace correspond à quelle partie du SMPMP : chaque composant peut avoir des caractéristiques particulières importantes pour l'analyseur qui doivent donc lui être transmises.

Une solution, qui ne limite en rien les possibilités, est par exemple que chaque composant génère un événement initial via son port de trace. Cet événement n'a aucune cause et aucune conséquence et contient uniquement les caractéristiques du composant. Par exemple, si un composant représente un cache, on peut penser que la taille des lignes, leur nombre et l'associativité sont des informations utiles.

5.3.1.2 Informations accessibles

Certaines informations sont rendues accessibles lors du traitement d'un événement afin d'effectuer l'analyse. La complexité pour obtenir ces informations à partir d'un événement doit être en $O(1)$.

Contenu de l'événement Le contenu de l'événement est évidemment nécessaire puisque c'est par rapport à celui-ci que l'analyse s'effectue.

Données de l'analyseur spécifiques à l'événement L'analyseur peut avoir besoin d'associer des données supplémentaires à un événement. En particulier, comme un événement peut être traité en plusieurs fois, cela peut permettre de transmettre des informations d'un traitant à l'autre.

Données de l'analyseur spécifiques au composant De même, des données peuvent être associées au composant qui a généré l'événement.

Événements « cause » Les relations sont une partie capitale de la trace. Les événements, qui sont référencés comme ayant causé l'événement en cours de traitement, sont donc accessibles.

5.3.1.3 Traitement centré sur les événements

Une fois les composants déclarés, l'analyse des événements est effectuée. Nous proposons une solution permettant d'analyser dynamiquement la trace. Les événements ont ainsi une certaine durée de vie : un événement apparaît quand il est tracé par son composant et disparaît une fois qu'il n'est plus utile, c'est à dire quand l'analyseur n'a plus besoin de lui. Entre ces deux moments, l'événement peut être traité en une ou plusieurs étapes.

Chacune des étapes du traitement est réalisée par ce que nous appelons un « traitant » : une fonction ayant un seul argument, l'événement traité. La subdivision du traitement d'un événement en plusieurs étapes est le moyen utilisé pour permettre de gérer les probables contraintes de l'analyseur par rapport aux relations de cause à effet et aux ordres dans lesquels sont tracés les événements par leurs composants respectifs. En laissant la possibilité de paramétrer des contraintes d'ordonnancement de ces traitants, il est en effet possible de laisser une grande flexibilité à l'analyseur.

Par exemple, supposons que les processeurs tracent les instructions dans les ordres dans lesquels ils les exécutent et qu'un périphérique trace les accès auxquels il répond dans l'ordre dans lequel il les traite. Ainsi pour traiter les instructions ayant accédé au périphérique dans l'ordre des accès à ce périphérique, un analyseur va devoir utiliser les relations de cause à effet entre les instructions et les accès pour pouvoir disposer des informations sur les événements des instructions lors du traitement des événements des accès au périphérique. Ainsi le traitement d'un accès au périphérique ne pourra se faire que quand l'événement de l'instruction le provoquant aura été traité (et que l'on disposera donc des informations de son contenu). Comme les événements sont tracés par des composants différents, seules les relations de cause à effet entre ces deux événements vont permettre de mettre en place ces contraintes d'ordonnancement entre les traitants.

Il y a quatre catégories de traitants qui sont exécutées dans l'ordre dans lequel elles sont listées ci-dessous. Trois des catégories comprennent un unique traitant.

Traitant de la réception La réception d'un événement est exécutée dès que l'événement est reçu par la bibliothèque d'analyse. Ce traitant est obligatoirement exécuté et permet par exemple d'initialiser des données spécifiques à l'événement. Lors de ce traitant, aucun accès aux événements « cause » n'est permis car ceux-ci n'ont pas forcément déjà été reçu par la bibliothèque. Les réceptions des événements tracés par un même composant sont exécutées dans l'ordre où ils ont été tracés.

Traitant initial Dans ce traitant l'ensemble des données définies à la section précédente, et en particulier les événements « cause », sont accessibles. Les événements « cause » ont tous été reçus par la bibliothèque et leurs réceptions respectives ont été exécutées.

Traitants additionnels L'analyseur peut déclarer des traitants additionnels pour un événement qui est exécuté après le traitant initial. Ce sont les seuls traitants optionnels.

Traitant de la suppression La suppression d'un événement a lieu lorsque tous ses autres traitants ont été exécutés. Elle permet alors de libérer la mémoire qu'il utilise. Ce traitant permet d'effectuer, juste avant la suppression effective, les éventuelles opérations de nettoyage des données spécifiques associées par l'analyseur à l'événement.

5.3.1.4 Définition des traitants

Initialement, seule la réception doit être définie. Cela doit se faire pendant la déclaration des composants avant que le traitement des événements commence. Ainsi la réception est commune à tous les événements d'un même composant. Cela n'empêche pas de différencier le traitement fait à l'intérieur de celle-ci en fonction des événements.

Les autres traitants sont spécifiques à chaque événement, ils peuvent être ajoutés à partir de sa réception. Aucun traitant additionnel n'est initialement défini, par contre le traitant initial et la suppression existent même si aucune fonction ne leur est affectée. Si aucune fonction ne leur est affectée, ils seront effectués silencieusement : ils n'auront aucun effet visible.

5.3.2 Ordonnancement des traitants

5.3.2.1 Contraintes initiales

Concernant un même événement, les traitants sont exécutés dans l'ordre de la liste. Les différents traitants additionnels ne sont pas, par défaut, ordonnés entre eux. En plus de ces contraintes qui sont locales aux traitants d'un même événement, des contraintes sont nécessaires pour garantir l'accès aux événements « cause » lors du traitant initial :

- le traitant initial d'un événement ne peut être exécuté avant que les réceptions des événements « cause » soient exécutées ; et
- les événements « cause » d'un événement ne peuvent être supprimés (et leurs suppressions exécutées) avant que le traitant initial de cet événement soit exécuté.

En dehors de ces contraintes, il n'y a aucune autre règle initiale d'ordonnancement. En particulier l'accès aux événements « cause » n'est pas garanti pendant la réception, les traitants additionnels et la suppression. L'accès aux événements « cause » n'est par ailleurs jamais garanti. Afin de pouvoir mener à bien une analyse, il est possible et vraisemblablement nécessaire d'ajouter des contraintes additionnelles entre les traitants.

On remarquera qu'aucun interblocage n'est possible avec ces contraintes initiales dans l'exécution des traitants si les événements sont consistants. Nous entendons par consistant le fait que le nombre d'événements référençant un même événement corresponde au nombre de conséquences

qu'il déclare. Comme il n'y a aucune contrainte entre un traitant et le même traitant d'un événement « cause », l'absence de tout interblocage est garanti même si les relations de cause à effet forment un cycle parmi les événements.

5.3.2.2 Contraintes additionnelles

Les contraintes initiales sont minimales. D'autres peuvent être définies par l'analyseur et contraindre l'ordonnancement des traitants.

Verrouillage Le premier type de contrainte est le plus basique : il permet de bloquer l'exécution d'un traitant jusqu'à nouvel ordre. Un système n'offrant qu'un seul degré de verrouillage (c'est à dire si l'état du traitant est booléen : verrouillé ou déverrouillé) permet d'implanter n'importe quel autre type de contrainte. Par exemple, en associant un compteur, on peut gérer le fait de devoir débloquer un traitant autant de fois qu'il a été bloqué (comme les marqueurs du chapitre précédent empêchant la suppression d'un nœud).

Des systèmes plus complexes peuvent aussi être mis en place. Afin d'éviter à l'analyseur de devoir implanter des mécanismes classiques qui sont sûrement utiles dans de nombreux cas, nous pensons que certains de ceux-ci doivent être intégrés dans la bibliothèque, en particulier le verrouillage à plusieurs degrés décrit précédemment.

L'ordre entre deux traitants Cette contrainte consiste à empêcher un traitant d'être exécuté tant qu'un autre traitant ne l'a pas été. Cette contrainte est bien sûr utilisable quel que soit les événements des traitants : que les deux événements diffèrent ou non.

Ce type de contrainte est particulièrement intéressant dans le cadre des relations de cause à effet car il permet de faire en sorte que des traitants d'événement soient exécutés dans l'ordre (ou l'ordre inverse) des relations de cause à effet. Les ALGORITHMES 5.1 et 5.2 donnent ainsi ce qu'il faut exécuter pour créer des ordres d'exécution « cause puis effet » et « effet puis cause ». Dans ces algorithmes, on note $\{E_{init}, E_{add:?}, E_{suppr}\}$ les différents traitants d'un événement E.

L'ALGORITHME 5.1 montre ainsi comment construire un arbre d'exécution de traitants respectant l'ordre « cause puis effet ». Les traitants contraints sont les traitants additionnels notés $E_{add:CpE}$ de chaque événement. Pour cela, les fonctions CAUSEPUISEFFET_RÉCEPTION et CAUSEPUISEFFET_INITIAL doivent être exécutées respectivement lors du traitant de réception et du traitant initial de chaque événement. La première fonction ajoute le traitant additionnel à chaque événement et la deuxième ajoute les contraintes d'exécution. La ligne 7 n'est pas obligatoire. Elle est uniquement nécessaire pour garantir que les événements « cause » d'un événement E ne soient pas supprimés avant que le traitant additionnel ne soit exécuté.

```
1: fonction CAUSEPUISEFFET_RÉCEPTION(événement E)
2:     AJOUTETRAITANT(E_add:CpE)
3: fin fonction

4: fonction CAUSEPUISEFFET_INITIAL(événement E)
5:     pour chaque événement C parmi les causes de E faire
6:         ORDONNE(C_add:CpE, E_add:CpE)      ▹ C_add:CpE sera exécuté avant E_add:CpE
7:         ORDONNE(E_add:CpE, C_suppr)         ▹ E_add:CpE sera exécuté avant C_suppr
8:     fin pour
9: fin fonction
```

ALGORITHME 5.1 - Réception d'un ordre d'exécution « cause vers effet »

L'ALGORITHME 5.2 donne les fonctions similaires pour la mise en place des contraintes inverses. Cela est un peu plus compliqué car il faut forcer le blocage du traitant additionnel (ligne 4) jusqu'à ce que l'ensemble des contraintes le concernant aient été mises en place (ligne 13). Il n'y a initialement aucun contrainte empêchant un traitant additionnel d'un événement d'être exécuté avant le traitant initial d'une de ses causes. *nb_cons* représente dans cette fonction un champ de donnée spécifique à l'analyseur associé à chaque événement.

```
 1: fonction EFFETPUISCAUSE_RÉCEPTION(événement E)
 2:     AJOUTETRAITANT(E_add:EpC)
 3:     si E a des conséquences alors
 4:         BLOQUE(E_add:EpC)
 5:         E.nb_cons := nombre de conséquences deE
 6:     fin si
 7: fin fonction

 8: fonction EFFETPUISCAUSE_INITIAL(événement E)
 9:     pour chaque événement C parmi les causes de E faire
10:         ORDONNE(E_add:EpC, C_add:EpC)        ▷ E_add:EpC sera exécuté avant C_add:EpC
11:         C.nb_cons := C.nb_cons − 1
12:         si C.nb_cons = 0 alors               ▷ C_add:EpC est débloqué quand toutes
13:             DÉBLOQUE(C_add:EpC)                 ses contraintes ont été ajoutées.
14:         fin si
15:     fin pour
16: fin fonction
```

ALGORITHME 5.2 - Réception d'un ordre d'exécution « effet vers cause »

Ce genre de contraintes (« cause puis effet » ou l'inverse) permet par exemple d'utiliser les traitants contraints pour faire remonter de l'information d'un événement initial (une instruction par exemple) vers un événement final (le traitement d'un accès par un périphérique par exemple) quel que soit le nombre d'événements intermédiaires ou dans le sens inverse.

Les FIFOs La deuxième catégorie de contraintes très utiles concerne plutôt le deuxième type d'informations que l'on peut tirer de la trace des événements : la séquence d'événements tracés par un même composant. L'ordre de cette séquence est respecté lors de l'exécution des traitants de réception, mais il peut être nécessaire d'imposer cet ordre à d'autres traitants, par exemple les traitants initiaux. L'ordre peut bien sûr être imposé uniquement sur un sous-ensemble des événements tracés par un composant.

L'ALGORITHME 5.3 donne une fonction à exécuter pendant le traitant de réception des événements d'un composant. Les traitants initiaux de ces événements seront exécutés suivant l'ordre des traitants de réception et donc l'ordre de la trace. Dans cette fonction, on utilise une FIFO stockée dans les données spécifiques des composants. L'accès à cette FIFO est fait à la ligne 2.

```
 1: fonction EFFETPUISCAUSE_RÉCEPTION(événement E)
 2:     INSERTFIFO(E.composant.fifo, E_init )
 3: fin fonction
```

ALGORITHME 5.3 - Maintien de l'ordre de la trace lors de l'exécution des traitants initiaux

Exemple de la vérification de la consistance mémoire Dans le cadre de la VCM, l'ensemble de ces contraintes est nécessaire. Les FIFOs sont ainsi nécessaires pour imposer les ordres de traitement des instructions par rapport à l'ordre des traces des processeurs. Par ailleurs, pour transporter les informations des accès depuis leur émission par les processeurs jusqu'à leur complétion, des ordres de type « cause puis effet » sont nécessaires. Inversement, pour rapatrier les informations depuis un banc mémoire vers un cache ayant fait une requête pour remplir une de ses lignes, des ordres de type « effet puis cause » sont nécessaires.

5.3.2.3 Complexité de l'environnement

Cet environnement consiste simplement à exécuter des traitants en respectant des contraintes d'ordre. En utilisant des structures de données appropriées, une implantation de cet environnement peut avoir une complexité totale en $O(N_T + N_C)$. Pour chaque traitant, il suffit en effet par exemple de conserver deux informations : 1. le nombre de traitants restant à exécuter avant, et 2. la liste des traitants à exécuter après.

Cet complexité nécessite en particulier de pouvoir trouver en $O(1)$ un événement par rapport à l'identifiant utilisé pour le référencer dans la trace. Dans cette équation, N_T représente le nombre total de traitants exécutés et N_C le nombre de contraintes d'exécution incluant les contraintes initiales. Cette complexité dépend donc beaucoup de l'analyseur. Dans le cas où le nombre de traitants par événement ainsi que le nombre de contraintes d'exécution sont bornés par des constantes, la complexité est en $O(N_E)$ où N_E est le nombre d'événements. On notera que la complexité algorithmique de l'environnement ne réalisant aucune analyse est de $O(N_E + N_R)$, où N_R représente le nombre de relations de cause à effet.

L'occupation mémoire en pire cas est de même limitée par $O(N_T + N_C)$.

5.4 Conclusion sur l'environnement de trace et d'analyse

Dans cette partie, nous avons défini un environnement générique permettant de réaliser des analyses diverses à partir des événements tracés depuis la simulation. Ces analyses peuvent prendre en compte les différentes relations de cause à effet exprimées dans les événements.

La complexité de celle-ci est entièrement dépendante de la trace et de l'analyseur. Il est en particulier raisonnable de supposer que le coût de l'environnement soit dans la majorité des cas en $O(C.N_E)$ et plus précisément que chaque événement soit traité en $O(C)$ où C est une constante. Cela n'est possible que si le nombre de relations de cause à effet est borné pour chaque événement. Comme ces relations correspondent à des liens réels, il est néanmoins probable que celles-ci dépendent de paramètres architecturaux fixes du système simulé.

Aucun interblocage n'est possible avec les contraintes initiales d'ordonnancement des traitants. Mais l'analyseur peut tout à fait en générer en paramétrant des contraintes supplémentaires. De manière similaire à la détection des cycles dynamiques décrite dans le chapitre précédent, il est possible de détecter les interblocages pour un coût amorti en $O(1)$ par événement.

L'occupation mémoire en pire cas est du même ordre que la complexité algorithmique. Si aucune contrainte d'ordre n'est ajoutée par rapport aux contraintes initiales, un événement n'est conservé que jusqu'à ce que l'ensemble de ses conséquences soient tracées. Cela entraîne que l'occupation mémoire est limitée dans ce cas en fonction des caractéristiques du système simulé.

6 | Expérimentations

Nous présentons dans ce chapitre les expérimentations faites afin d'évaluer les propositions décrites dans les chapitres précédents.

Nous détaillons dans un premier temps les différentes implantations réalisées pour effectuer ces expérimentations. Dans une seconde partie nous décrivons les différentes expérimentations réalisées. La troisième partie est dédiée à l'analyse des expérimentations. Nous y étudions en particulier l'impact de l'algorithme de VCM sur une simulation d'un prototype virtuel ainsi que son occupation mémoire.

Nous concluons finalement ce chapitre.

Chapitre 6

6.1 Implantations **98**
- 6.1.1 Instrumentation d'un prototype virtuel 98
- 6.1.2 Implantation de l'environnement de trace et d'analyse 99
- 6.1.3 Implantation de l'algorithme de vérification de la consistance mémoire 102

6.2 Description des expérimentations **105**
- 6.2.1 Plate-forme matérielle simulée 105
- 6.2.2 Logiciel embarqué sur la plate-forme 107
- 6.2.3 Résultats des simulations 107
- 6.2.4 Détection des violations du modèle de consistance mémoire 110

6.3 Évaluation du coût de la vérification de la consistance mémoire **111**
- 6.3.1 Impact sur les simulations 111
- 6.3.2 Coût de la vérification de la consistance mémoire 112
- 6.3.3 Occupation mémoire 115

6.4 Conclusion **118**

6.1 Implantations

6.1.1 Instrumentation d'un prototype virtuel

Afin de mener à bien les expérimentations, nous avons instrumenté des modèles de composants de la bibliothèque de composants *SoCLib* [SoCLib]. *SoCLib* est une bibliothèque de composants écrits en *SystemC* [SystemC] développée pour le prototypage virtuel de MPSoCs. Nous avons instrumenté des modèles de composants *cycle accurate* afin qu'ils tracent les événements voulus indiqués ci-dessous. Cette instrumentation a été faite en ayant pour objectif deux analyses :

1. la VCM telle qu'elle est décrite dans le chapitre 4, et
2. l'analyse du programme exécuté par le MPSoC simulé dans le cadre du projet *Decopus* [1].

Les événements tracés ont donc été choisis par rapport à ces objectifs. Dans cette section, nous listons les événements tracés par chaque composant instrumenté. La description complète des événements et de leurs relations est donnée dans l'annexe C.

6.1.1.1 Instrumentation des composants

Processeur Le modèle de processeur instrumenté est un ISS simulant le jeu d'instructions *MIPS32*. Ce modèle trace trois types d'événements :

– une « instruction » pour chaque instruction exécutée ;
– un « accès » pour chaque accès à la mémoire ou à un périphérique ;
– une « exception » pour chaque exception prise (cet événement n'est pas utile pour la VCM).

Chaque processeur trace ses instructions dans l'ordre de sa SI et un accès est tracé avant l'instruction qui le génère. Cet accès est par ailleurs une conséquence de l'instruction qui la génère.

Périphériques Les modèles des machines à états utilisées pour le traitement des accès aux périphériques adressables (mémoires ou autres) ont été instrumentés. Ils tracent un seul type d'événement :

– un « acquittement » pour chaque accès traité provenant d'un processeur.

Les acquittements sont tracés dans l'ordre où ils sont effectués par chaque périphérique.

Caches Les modèles des caches ont été instrumentés car ceux-ci sont utilisés pour répondre à certains accès et contiennent des copies de certaines écritures. Ainsi on cherche à connaître quelles sont les écritures contenues dans chaque cache. Pour cela, ils tracent :

– un « accès » pour chaque invalidation de ligne ou lecture (suite à un défaut de cache) effectuée ;
– un « acquittement » pour chaque accès d'un processeur pris en compte par le cache. La prise en compte d'une lecture signifie la lecture depuis une copie contenue dans le cache tandis que la prise en compte d'une écriture indique une mise à jour de cette copie.

Les accès et acquittements d'un cache sont tracés dans l'ordre correspondant à l'évolution de son contenu. Ainsi l'interprétation des accès dans l'ordre de la trace permet de faire évoluer le contenu du cache et les acquittements sont interprétés en fonction de celui-ci. En particulier, l'accès informant de l'invalidation d'une ligne doit être tracé avant l'accès envoyé à la mémoire pour récupérer une autre ligne en remplacement. Ceci doit être le cas, même si l'invalidation est une conséquence de l'émission de l'accès vers la mémoire (car il faut libérer une ligne du cache pour stocker la nouvelle).

1. Decopus : Decopus fait partie du projet Nano 2012.

Tampon d'écriture Le modèle du tampon d'écriture a été instrumenté pour gérer les réponses à des lectures et le regroupement de plusieurs écritures. Il trace les événements suivants :

– un « acquittement » pour chaque écriture prise en compte et chaque lecture répondue.

– un « accès » pour chaque écriture propagée vers la mémoire. Dans le cas du regroupement de plusieurs écritures, cet accès les représente tous.

De manière similaire à un cache, les événements sont tracés dans l'ordre permettant de faire évoluer le contenu du tampon d'écriture en interprétant les événements. Ainsi un accès en écriture acquitté par un tampon d'écriture est considéré comme contenu dedans jusqu'à ce que le tampon trace un accès incluant l'accès initial.

Les relations de cause à conséquence entre les événements tracés vont de manière générale des instructions vers les acquittements avec des intermédiaires en particulier les accès. Les relations et les intermédiaires sont précisément décrits dans l'annexe C.

6.1.1.2 Interfaces instrumentées

Afin de pouvoir relier les accès et leurs acquittements respectifs, certaines interfaces entre les différents composants ont été modifiées.

Interface processeur – cache Les interfaces entre le processeur et les caches (de données et d'instructions) ont été modifiées pour pouvoir transférer l'identifiant de l'événement de l'accès du processeur en même temps que ses caractéristiques (adresse, type, donnée). Dans chacune des interfaces, un champ contenant l'identifiant a donc été ajouté.

Interface de communication Dans la bibliothèque *SoCLib*, le protocole de communication utilisé entre les différents composants (caches et périphérique) est le protocole VCI [VCI]. Afin de pouvoir transférer l'identifiant d'un accès correspondant à un paquet transmis par ce protocole, nous avons étendu la description des signaux des interfaces et des *flits* transportés. Un champ a ainsi été ajouté dans les *flits* transportés par le système de communication.

Cette modification est très localisée puisque dans la bibliothèque, les différents composants d'interconnexions existants partagent la même description des interfaces et des *flits*. Il n'a pas été nécessaire de modifier chaque modèle d'interconnexion.

Interface tampon d'écriture – cache Le cache de donnée et le tampon d'écriture associés à un processeur sont regroupés dans un même objet logiciel *SystemC* dans le bibliothèque *SoCLib*. Chacun de ces composants utilise néanmoins un port de trace dédié car ces deux composants représentent des composants ayant des sémantiques différentes. Par contre, il n'a pas été nécessaire de réellement modifier d'interface pour que le tampon d'écriture et le cache de données puissent échanger des identifiants.

6.1.2 Implantation de l'environnement de trace et d'analyse

6.1.2.1 Architecture logicielle

L'environnement de trace et d'analyse implanté est caractérisé par deux *Application Programming Interface*s (APIs). La première est l'API de *trace* qui permet à un prototype virtuel de tracer des événements. La seconde est l'API d'*analyse* qui permet de construire une analyse des événements tracés.

L'architecture logicielle est illustrée dans la FIGURE 6.1. Le prototype virtuel instrumenté est simulé par un programme (le simulateur) exécuté sur la machine hôte. Ce programme utilise une bibliothèque logicielle pour effectuer la trace. Cette dernière est construite spécifiquement pour une analyse donnée à partir de l'environnement d'analyse et l'analyseur développé.

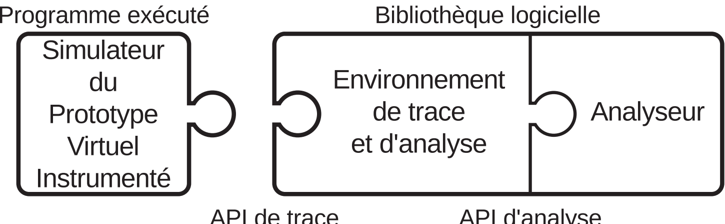


FIGURE 6.1 - Architecture logicielle de la simulation d'un prototype virtuel instrumenté

La pile d'appel va de la gauche vers la droite : le prototype virtuel simulé appelle les fonctions de l'environnement pour tracer les événements. Nous avons implanté deux versions de cet environnement. La première est strictement séquentielle : l'analyse est faite dans le même *thread* que la simulation. La seconde est concurrente : l'analyse est faite dans un *thread* spécifique permettant ainsi de limiter l'impact sur la vitesse de simulation.

6.1.2.2 Système de trace

Notre implantation du système de trace est limitée à un simulateur non parallèle. L'implantation du système de trace n'est pas capable de gérer des événements provenant de plusieurs *threads* tel que ce serait le cas avec une implantation parallèle de la simulation.

L'API de trace est simple, elle contient en particulier une fonction permettant à un composant d'ouvrir un *port de trace*. Ce port de trace lui permet ensuite de récupérer des événements vides pour les remplir et finalement les tracer. L'ordre dans lequel chaque composant trace les événements a une importance pour certains événements. Un processeur trace par exemple les instructions exécutées dans l'ordre de son programme.

Chaque événement vide est fourni par le système de trace car il lui attribue son identifiant unique permettant de mettre en place les relations de cause à conséquence entre événements. Seul l'ordre de la trace compte, l'ordre dans lequel les événements vides ont été récupérés n'a aucune importance. De plus un composant peut avoir plusieurs événements en cours de remplissage au même moment. Par exemple, un accès et l'instruction correspondante dans le cas d'un processeur.

Les événements Un événement est implanté comme une liste chainée de conteneurs génériques de taille fixe. Initialement un événement vide n'est qu'un unique conteneur. Si le conteneur ne suffit pas, une fonction permet à un composant de rajouter des conteneurs un par un à un événement. Les conteneurs sont alloués par l'environnement de trace et d'analyse.

Un événement vide est donné au prototype virtuel qui le remplit et le renvoie à l'environnement. Celui-ci est ensuite traité par l'analyseur. Une fois le traitement terminé, les conteneurs formant un événement sont recyclés et utilisés pour d'autres événements.

Les relations entre événements L'environnement n'a pas connaissance du contenu des événements. Il n'a besoin que du nombre de conséquences et de la liste des causes de chaque événement. Ainsi seul l'analyseur et le prototype virtuel ont besoin de connaitre la sémantique du contenu des conteneurs.

L'API définit des fonctions permettant d'obtenir l'identifiant, de définir le nombre de conséquences et d'indiquer les causes d'un événement. Les identifiants sont implantés comme des pointeurs vers les événements. Cela permet un accès très efficace lors de l'analyse, puisqu'il s'agit d'un simple déréférencement.

6.1.2.3 Environnement d'analyse

L'environnement d'analyse permet de construire une bibliothèque de trace réalisant une analyse. Cette bibliothèque est compilée à partir d'un noyau commun (l'environnement de trace et d'analyse) et une partie développée spécifique : l'analyseur.

Dans notre implantation, l'analyseur peut associer des structures de données additionnelles aux événements et aux ports de trace qui représentent les différents composants matériels simulés. Il doit aussi implanter quatre fonctions principales qui sont appelées à des moments stratégiques durant l'analyse.

Déroulement de l'analyse L'analyse se déroule en cinq phases. Quatre d'entre elles correspondent à l'appel de fonctions qui doivent être implantées par l'analyseur :

1. la première fonction est appelée au début et permet éventuellement d'initialiser des données ;
2. la seconde fonction est appelée ensuite pour déclarer à l'analyseur chaque port de trace ;
3. la troisième fonction est appelée une fois que tous les ports de trace sont déclarés ;
4. les traitants des événements sont appelés pendant cette phase, c'est le gros de l'analyse ;
5. la quatrième fonction est appelée après le traitement des événements pour finaliser l'analyse.

L'environnement d'analyse fournit de plus les fonctions permettant de définir les différents traitants d'un événement. Il fournit aussi celles pour mettre en place les contraintes entre ces traitants (contrainte d'exécution entre deux traitants et mise d'un traitant dans une FIFO).

Les traitants Nous avons implanté une version limitée de l'environnement d'analyse défini au chapitre 5. Pour nos analyses, et en particulier la VCM, il n'est en effet pas nécessaire de pouvoir mettre en place un nombre variable de traitants additionnels. Ainsi seuls trois traitants (le traitant initial et deux traitants additionnels prédéfinis) sont paramétrables par l'analyseur dans notre implantation. Les traitants additionnels sont nommés « aller » et « retour ».

Pour chaque événement, ils sont exécutés dans cet ordre :

1. Le traitant « initial » permet d'initialiser le traitement de l'événement. Concernant les événements issus d'un même port de trace (c'est à dire d'un même composant matériel), ceux-ci sont exécutés dans l'ordre dans lequel ils ont émis à travers ce port de trace.
2. Le traitant additionnel « aller » est ensuite exécuté. Il est exécuté après les traitants « aller » des causes mais avant ceux des conséquences.
3. Le traitant additionnel « retour » est finalement exécuté. Il est exécuté après les traitants « retour » des conséquences mais avant ceux des causes.

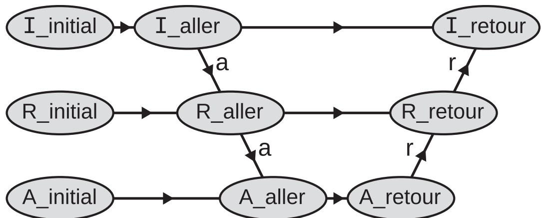


FIGURE 6.2 - Ordres entre les traitants paramétrables par l'analyseur

La FIGURE 6.2 montre les contraintes entre les différents traitants pour trois événements : 1. une instruction (notée *I*), 2. un accès (conséquence de l'instruction) (noté *R* car l'événement correspond à la requête de l'accès), et 3. un acquittement (conséquence de l'accès) (notée *A*). Dans cette figure les nœuds représentent les différents traitants listés ci-dessus. Pour les traitants de l'instruction, ils sont ainsi notés *I_traitant*. Les arcs représentent les différentes contraintes :

– les contraintes entre les traitants d'un même événement (arcs non étiquetés) ;
– les contraintes entre les traitants « aller » (arcs étiquetés avec un « a ») ; et

– les contraintes entre les traitants « retour » (arcs étiquetés avec un « r »).

6.1.2.4 Traitement général d'un événement

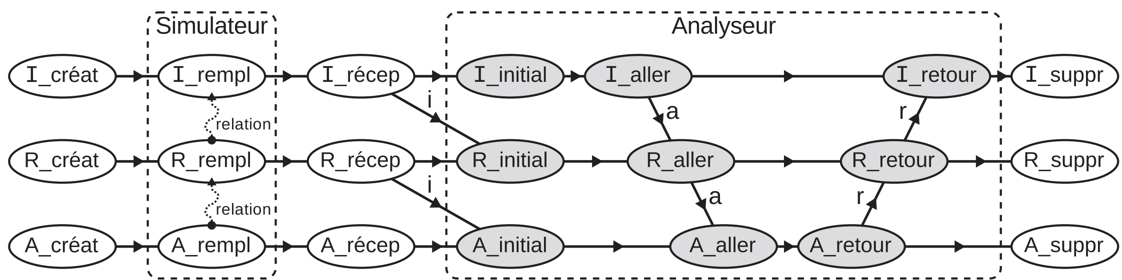


FIGURE 6.3 - Traitements effectués sur trois événements durant leurs durées de vie

La FIGURE 6.3 montre l'ensemble des traitements effectués sur les trois événements de la FIGURE 6.2. Les parties où l'analyseur et le simulateur interviennent sont indiquées par des cadres pointillés.

Cette figure inclut les traitants paramétrables, qui sont grisés, mais aussi les traitants de réception (*récep*) et de suppression (*suppr*). Le traitant de réception d'un événement est exécuté dès que l'événement est reçu par la bibliothèque de trace. Il initialise en particulier les données de l'événement spécifique à l'analyseur. Le traitant de suppression d'un événement consiste uniquement à recycler les conteneurs de l'événement. Les arcs étiquetés avec un « i » sont les contraintes qui garantissent que les causes d'un événement ont bien été tracées avant d'exécuter le traitant initial : il est ainsi possible d'accéder à celles-ci depuis ce traitant.

La figure inclut aussi un traitement de création (*créat*) et un traitement de remplissage (*rempl*) pour chaque événement. Ces traitements ne sont pas des traitants dans le sens utilisé ici mais correspondent à certaines actions effectuées :

– la création correspond au moment où un événement vide est fourni au simulateur du prototype virtuel ;
– le remplissage correspond aux actions effectuées par le prototype virtuel instrumenté.

6.1.3 Implantation de l'algorithme de vérification de la consistance mémoire

6.1.3.1 Utilisation de l'environnement de trace et d'analyse

Nous explicitons dans cette section le principe général de la vérification à partir de l'environnement d'analyse. La FIGURE 6.4 montre le traitement de cinq événements : elle reprend les trois événements des précédents exemples. Ce scénario comprend l'exécution par un processeur d'une instruction (I) générant un accès (R). Cet accès est une lecture qui est acquittée (A) par un cache car il contient la ligne accédée. En plus de ces trois accès, on considère que le cache a effectué auparavant un accès (RL) pour lire une ligne depuis la mémoire. Cette dernière a répondu à la requête avec l'acquittement (AL).

Selon les règles des événements décrits au début de ce chapitre, la requête du processeur est tracée avant l'instruction et la requête du cache est tracée avant l'acquittement qu'il fait ensuite. Les traitants initiaux de ces événements sont donc exécutés en respectant ces deux ordres : R_{initial} avant I_{initial} et RL_{initial} avant A_{initial}. Ces contraintes sont indiquées avec des arcs étiquetés « t ».

Ordre entre les phases d'exécution et de complétion La gestion de l'accès correspondant à la requête R dans l'algorithme de VCM se fait en deux étapes : l'exécution de l'instruction correspondante puis la complétion. La solution utilisée est d'effectuer le traitement de l'exécution dans le

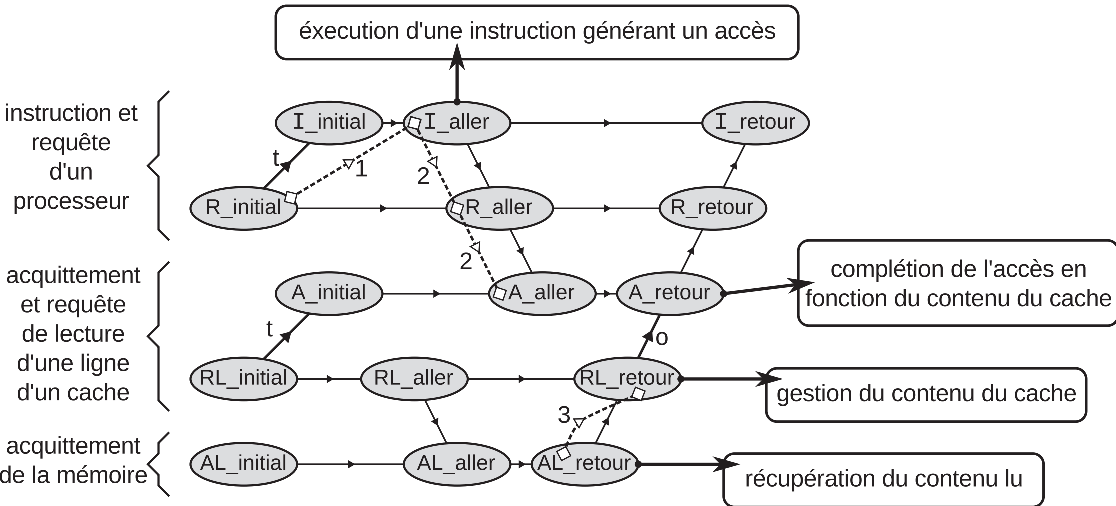


FIGURE 6.4 - Localisation des traitement VCM parmi les traitants

traitant I_{aller} et d'effectuer le traitement de la complétion A_{retour}. Les contraintes entre ces traitants garantissent que l'exécution soit faite avant la complétion. Par ailleurs il est possible de transférer de proche en proche les informations sur l'accès à compléter de I à A (les flèches pointillées étiquetées « 2 » dans la figure).

Ordre entre les phases d'exécution d'une même SI Pour imposer que les exécutions des instructions d'un même processeur soient faites dans l'ordre du programme, il faut ajouter des contraintes additionnelles sur les traitants « aller » des instructions. Il suffit en fait que ceux-ci soient exécutés dans le même ordre que leurs traitants initiaux. Ces derniers sont en effet exécutés dans l'ordre de la trace qui correspond à l'ordre d'exécution des instructions.

Grâce au fait que R_{initial} est exécuté avant I_{initial}, il est possible de transférer les informations nécessaires à propos de l'accès de R à I (flèche pointillée « 1 »). La requête R étant une conséquence de I, elle possède son identifiant et peut donc accéder à I. Ainsi lors du traitant I_{initial}, les données concernant l'accès effectué sont connues et il est donc possible d'effectuer la phase d'exécution de l'instruction I.

Sans la règle imposée sur l'ordre de trace entre une requête et l'instruction correspondante, il aurait été nécessaire d'ajouter d'autres contraintes entre les traitants ou un autre traitant additionnel pour transférer les informations.

Phases de complétion Pour les phases de complétion des accès, le besoin est de connaître l'accès (écriture normale ou accès I/O) qui va servir de référent lors de la complétion. Ainsi il faut pouvoir disposer du contenu (du tampon, cache ou segment mémoire) du composant qui réalise l'acquittement. Pour cela il faut traiter, dans ces composants, les différents événements impactant ce contenu dans le bon ordre : celui dans lequel les événements ont été tracés.

Pour les caches, le contenu dépend en particulier du chargement de lignes depuis la mémoire (événements RL et AL). Comme les copies des écritures qui sont chargées ne peuvent être connues qu'au niveau de la mémoire, il faut transférer celles-ci entre la mémoire et le cache (flèche pointillée « 3 »). La chaîne des traitants « retour » est utilisée à cette fin. C'est pourquoi la gestion du contenu du cache ne peut être faite que dans les traitants « retour » (RL_{retour}).

Pour effectuer les complétions dans le bon ordre par rapport à cette gestion, l'ordre des traitants « retour » est imposé par des contraintes additionnelles (flèche « o ») pour correspondre à celui de la trace, en particulier dans les caches. Il serait dans certains cas possible d'effectuer certaines complétions dans des traitants « aller ». Il est néanmoins plus simple et préférable de réaliser

tous les traitements concernant un même port de trace dans le même type de traitant. Dans le cas contraire, il faudrait en effet alors mettre des contraintes supplémentaires entre des traitants « aller » et « retour », ce qui doit être fait avec précaution : cela peut en effet facilement provoquer des cycles dans le graphe d'exécution des traitants si les événements sont reliés, même indirectement, par des relations de cause à conséquence.

Phases de disparition Les phases de disparition des écritures sont gérées grâces aux événements tracés par les caches (invalidations et chargements de lignes). Ces événements permettent de maintenir le nombre de copies existantes pour chaque écriture dont la donnée est contenue dans un cache.

6.1.3.2 Description de l'implantation réalisée

Nous avons implanté l'algorithme de VCM décrit dans le chapitre 4 permettant de vérifier différents MCMs. En particulier la vérification du modèle SC et un modèle de type TSO (similaire à celui de l'architecture *x86*) ont été implantés. Ce dernier autorise uniquement le réordonnancement des écritures avec les lectures qui les suivent. Il permet à un processeur de lire de manière anticipée ses écritures, par exemple depuis un tampon d'écriture, sans que cela n'ajoute des contraintes sur les accès qui vont suivre.

Ce modèle correspond à ce qui est implanté dans les plateformes matérielles que nous allons simuler pour nos expérimentations. Les dépendances du programme qui sont à vérifier sont les suivantes :

- les dépendances WAW, RAR et WAR quelles que soient les cases mémoire accédées,
- les dépendances RAW mais uniquement pour les accès I/Os, LDLs, STCs, et
- les barrières mémoire.

La gestion de la lecture anticipée des écritures non-I/Os nécessite, comme nous l'avons souligné dans la section 4.2.3.1, de garder une trace de la dernière écriture de chaque processeur à chaque case mémoire. Cela peut nécessiter une grande quantité de mémoire.

Bien que les dépendances de chacun des processeurs soient assez limitées, le problème majeur auquel on peut avoir à faire face dans la VCM concerne l'occupation mémoire et il est présent dans ce MCM. Il faut en effet conserver une trace de certains nœuds pour chaque couple formé d'un processeur et d'une case mémoire.

Nous avons aussi implanté la vérification d'un modèle plus relâché de type RMO (de *SPARC*) dans lequel toutes les dépendances sont relâchées mises à part les dépendances de données à travers les registres et les cases mémoire. Cela nécessite de prendre en compte les mouvements et opérations de données dans chacun des registres des processeurs et la plupart des instructions doivent être donc traitées. Bien que nous ne disposons pas de plateforme matérielle implantant un modèle aussi relâché, cela permet d'évaluer le coût de la gestion de ces dépendances au niveau de la VCM.

Optimisation de l'occupation mémoire Afin d'avoir une occupation mémoire limitée, nous avons implanté les mécanismes décrits dans le chapitre 4. En particulier les nœuds et les références sont supprimés temporairement lorsque cela est possible. Ils sont alors restaurés quand ils sont nécessaires pour effectuer une opération avec le nœud supprimé.

Cases mémoire L'algorithme a été implanté en utilisant des cases mémoire de la taille des lignes de cache pour les segments de mémoire classique. Pour les périphériques, une case mémoire correspond à un segment entier.

L'utilisation de cases mémoire de la taille d'une ligne de cache a posé un problème pour la vérification des accès LdL et StC. Par rapport à l'algorithme 4.8 présenté dans le chapitre 4, des adaptations ont été nécessaires car la taille des cases mémoire (une ligne de cache) ne correspond pas au grain de ces accès (un mot de 32 bits). Les étiquettes utilisées pour stocker l'information d'un LdL ont été étendues pour indiquer aussi quel mot est accédé à l'intérieur de la ligne de cache. Cela permet de différencier chacun des mots de la ligne de cache pour l'atomicité d'un couple (LdL, StC). Lors de la disparition d'une écriture, il faut aussi propager les étiquettes à l'écriture suivante de la case mémoire.

Structures de données L'algorithme est basé sur la manipulation de références vers des nœuds. Toutes les opérations sur les nœuds (ajout d'un nœud, ajout d'un arc, gestion des marqueurs, ...) sont effectuées à partir de références vers des nœuds. L'utilisation d'un niveau d'indirection permet de gérer la suppression définitive ou temporaire des nœuds dès que cela est possible : le nœud est supprimé mais pas les références vers celui-ci. Pour chaque nœud, la liste des références courantes vers celui-ci est conservée. Cela permet par exemple de mettre à jour ces références lorsque le nœud est supprimé.

Les accès sont composés de deux références. La première permet de conserver le nœud principal de l'accès (nœud R ou W). La seconde est utilisée pour le nœud G dans le cadre d'une écriture. Dans le cas d'une lecture anticipée, cette seconde référence sert à stocker le nœud de la dernière écriture du processeur.

Les différents segments mémoire sont stockés dans un arbre, ce qui permet de retrouver rapidement le segment accédé à partir d'une adresse qu'il contient. Pour chaque segment, un tableau représentant son contenu courant est conservé. Ce tableau contient l'écriture correspondant à la valeur de chaque case mémoire du segment.

Pour pouvoir disposer de la dernière écriture de chaque processeur à chaque case mémoire, il faut conserver une structure similaire pour chaque processeur. Les tableaux contiennent par contre uniquement des références vers le nœud de la dernière écriture à chaque case mémoire.

Ces tableaux sont en pratique quasiment vides du fait de la suppression des références inutiles. À la place, des arbres ou des tables de hachage pourraient être utilisés permettant de réduire l'occupation mémoire. Néanmoins les tailles des différents segments de nos expérimentations ne sont pas problématiques et cela n'a pas été nécessaire dans notre cas.

Historique des opérations Un mécanisme d'historique des dernières opérations (exécutions des traitants, ajouts de nœuds, d'arcs, etc.) est conservé en permanence. Cela permet, lors de la détection d'une violation, d'analyser la séquence des opérations ayant mené à la violation et aider à en trouver la cause.

6.2 Description des expérimentations

6.2.1 Plate-forme matérielle simulée

La Figure 6.5 montre l'architecture générale de la plateforme matérielle simulée pour réaliser nos expérimentations. Celle-ci contient les composants listés ci-dessous.

Système d'interconnexion Le système d'interconnexion utilisé est très abstrait et implante le protocole VCI. Il simule une topologie de type *crossbar* : le trajet pour aller de n'importe quel initiateur d'une transaction vers n'importe quel destinataire a les mêmes caractéristiques. Chaque trajet possible est modélisé par une simple FIFO. L'utilisation de ce système abstrait à l'avantage d'être très rapide à simuler par rapport à un système implantant fidèlement une topologie de réseau en grille par exemple.

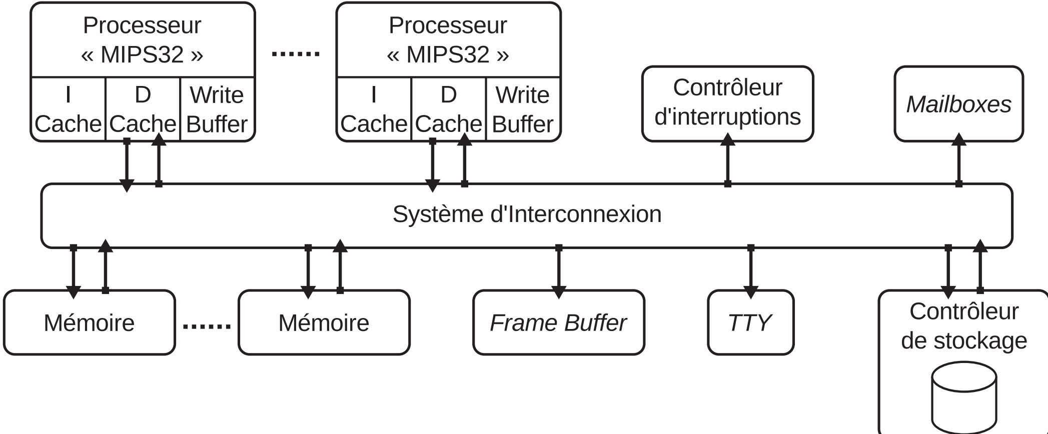


FIGURE 6.5 - Plate-forme utilisée dans les expérimentations

Processeur « MIPS32 » Ce processeur est un ISS du jeu d'instructions MIPS32. Il modélise un pipeline à trois étages (chargement de l'instruction, exécution de l'instruction et écriture des registres) et les problèmes de dépendances qui en découlent. Le pipeline de ce processeur est entièrement *in-order*, les accès à la mémoire ne sont pas réordonnés par le pipeline. Nous avons utilisé des plates-formes avec différents nombres de processeurs.

Caches et tampons d'écriture Chaque processeur est connecté à un groupe formé d'un cache pour les instructions, d'un cache pour les données et d'un tampon d'écriture. Ils partagent une interface vers le système d'interconnexion. La politique de cohérence utilisée est de type *write through invalidate*. Les caches utilisent des lignes de cache de 32 octets, soit 8 mots de 32 bits et sont non associatifs. Les caches d'instructions et de données contiennent 256 lignes et contiennent donc 8kio au total. Différentes tailles pour les cache de données et les tampons d'écriture ont été utilisées. Aucun mécanisme de mémoire virtuelle n'est implanté.

Le réordonnancement des lectures par rapport aux écritures précédentes est implanté dans ce groupe par l'intermédiaire du tampon d'écriture. Le tampon d'écriture permet aussi la lecture anticipée des écritures qu'il contient. Un tampon de regroupement est utilisé afin de regrouper les écritures consécutives dans une même ligne de cache et limiter les communications.

Mémoire La mémoire, entièrement partagée, est répartie dans plusieurs bancs. Chaque banc mémoire contient un répertoire permettant de gérer la cohérence du contenu de son banc. Les invalidations des caches sont envoyées par la mémoire lorsque cela est nécessaire. Ces mémoires sont initialisées dès le démarrage de la simulation avec le contenu d'un programme fourni en paramètre du simulateur.

Afin de ne pas trop limiter les performances lors de l'utilisation de beaucoup de processeurs, la mémoire est distribuée en plusieurs bancs ce qui permet de répartir la charge des accès. Comme les invalidations sont envoyées par les bancs mémoires, l'utilisation d'un banc unique ne passe pas à l'échelle avec l'augmentation du nombre de processeurs. La distribution de la mémoire permet d'augmenter le parallélisme du système mémoire car le nombre d'accès qui pouvant être traités en même temps est plus important. Les chances de provoquer une erreur de consistance en cas de mauvaise implantation sont aussi plus importantes.

Périphériques Divers périphériques sont utilisés : un contrôleur d'interruption, un ensemble de *mailboxes*, un *frame buffer*, une sortie de type TTY. Un périphérique de stockage de masse

contenant un DMA est aussi utilisé. Tous les accès à ces périphériques sont considérés comme des accès I/Os pour la VCM.

6.2.2 Logiciel embarqué sur la plate-forme

La plate-forme est utilisée avec le système d'exploitation SMP *DNA-OS* [GP09]. Deux programmes ont été utilisés avec cette plate-forme pour réaliser les expérimentations.

- Une application de décodage *MJPEG*. Cette application a été utilisée sur des plates-formes ayant de un à quatre processeurs. L'application est composée de plusieurs *threads*. Un premier *thread* lit un fichier d'entrée depuis le périphérique de stockage et distribue les données à plusieurs *threads* de décodage. Ces *threads* transmettent ensuite les images décodées à un dernier *thread* chargé de les afficher. Autant de *threads* de décodage que de processeurs sont utilisés. Cette application effectue beaucoup de transferts de données entre les différents *threads* en utilisant des FIFOs logicielles.
- Afin de faire des expériences avec plus de processeurs, l'application *ocean* du benchmark *SPLASH2* [Splash2] a été utilisée avec 4, 8, 16 et 32 processeurs. Cette application a été conçue pour évaluer la performance d'un système à mémoire partagée. Cette application ne fait pas de transferts de données mais distribue un calcul sur plusieurs processeurs qui travaillent sur les mêmes données. Pour nos expérimentations, les paramètres du calcul *OCEAN* sont une taille de grille de 258×258 et une précision cible de 10^{-4}. Un *thread* par processeur est utilisé.

6.2.3 Résultats des simulations

Les expérimentations ont été effectuées principalement sur un ordinateur équipé d'un processeur *Intel Core i5 2500K* ayant une fréquence bloquée à 3,3 GHz et 6 Mio de cache. Afin d'effectuer des tests sur une autre architecture, certaines expérimentations ont été faites aussi avec un processeur *Intel Core 2 Duo E4500* ayant une fréquence de 2,2 GHz et 2 Mio de cache. Pour une même simulation, il faut environ 2 fois plus de temps avec le *Core 2 Duo* qu'avec le *Core i5*. Les simulation faites avec le *Core 2 Duo* sont indiquées avec le préfixe « c2d ». Quand il n'y a pas de précision, c'est que les résultats ont été obtenus avec le *Core i5*.

La plupart des expérimentations faites sont composées d'un seul *thread* et certaines sont composées de deux *threads*. Le *Core i5* utilisé dispose de quatre cœurs permettant à ces dernières de profiter du parallélisme. Les expérimentations ont toutes été effectuées séparément afin d'éviter un impact dû à l'utilisation partagée du cache du processeur.

Nous donnons dans cette partie un aperçu du comportement des différentes simulations et des applications exécutées dans les plates-formes simulées. Nous nous intéressons en particulier à l'efficacité des applications et à aux événements générés.

6.2.3.1 Comportement des applications

Les simulations ont été effectuées sur une durée de 1 milliard de cycles. Environ 500 000 cycles sont nécessaires pour initialiser l'OS avant que l'application soit démarrée réellement.

La Figure 6.6 montre le comportement du logiciel exécuté par les différentes plates-formes simulées. L'axe des abscisses indique l'application exécutée ainsi que le nombre de processeurs de la plate-forme matérielle. L'efficacité de la parallélisation renseigne sur la quantité moyenne de travail effectuée par rapport au nombre de processeurs. Le taux d'inactivité des processeurs renseigne le temps pendant lequel l'ordonnanceur de l'OS n'a rien à exécuter sur des processeurs disponibles.

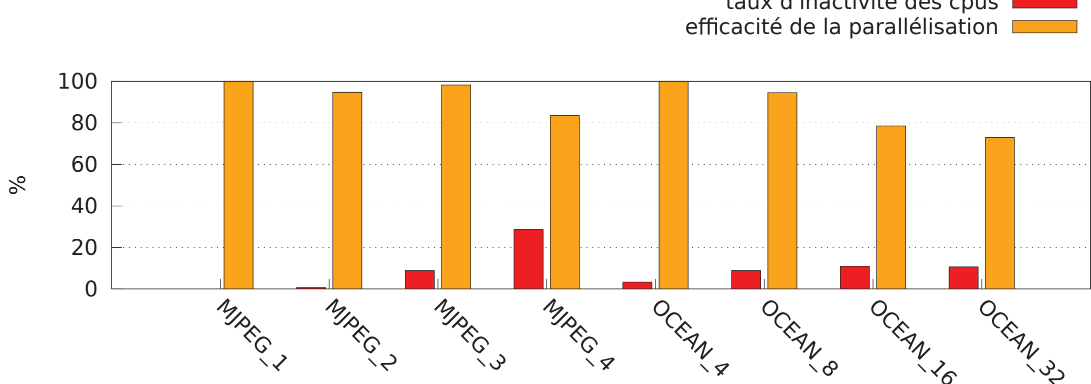


FIGURE 6.6 - Comportement des applications exécutées

Pour l'application *MJPEG*, l'efficacité de la parallélisation correspond au nombre d'images décodées en fonction du nombre de processeurs. On observe que l'application atteint ses limites à partir de 4 processeurs : plus d'images sont décodées qu'avec 3 processeurs, mais le taux d'inactivité est élevé (plus de 25%, soit plus d'un processeur en moyenne). Cela entraîne logiquement une baisse de l'efficacité relative au nombre de processeurs.

Pour l'application *OCEAN*, l'efficacité est évaluée par rapport au temps nécessaire pour terminer le calcul. Pour information, il faut environ 10 milliards de cycles avec 4 processeurs et 1,7 avec 32. L'efficacité est indiquée par rapport à l'exécution sur 4 processeurs dans ce cas. Pour cette application, l'efficacité baisse pour 16 et 32 processeurs alors que le taux d'inactivité reste constant. Pour 16 processeurs, la latence des lectures augmente ce qui provoque ce ralentissement. Pour 32 processeurs, le phénomène est amplifié par le fait que la plupart des écritures sont transmises jusqu'à la mémoire : peu d'entre elles sont regroupées par le tampon d'écriture.

6.2.3.2 Durées des simulations non-instrumentées

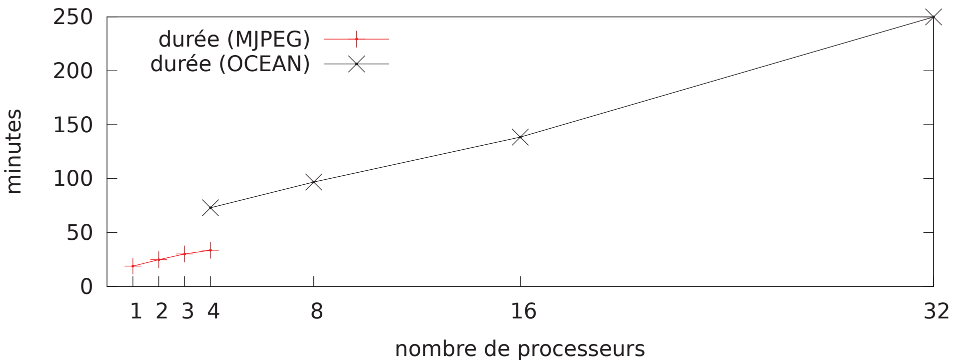


FIGURE 6.7 - Durées initiales des simulations effectuées (pour 1 milliard de cycles)

La FIGURE 6.7 donne les temps de simulation initiaux des plates-formes matérielles simulées pour les deux applications. Les temps indiqués concernent les plates-formes non instrumentées. L'ajout de processeurs à une plate-forme a un impact linéaire sur le temps de simulation. L'évolution du temps de simulation est néanmoins affine car il y une partie des plates-formes matérielles qui est fixe quel que ce soit le nombre de processeurs. Le temps de simulation des deux plates-formes comprenant 4 processeurs est très différent car la plate-forme utilisée pour l'application *OCEAN* comprend beaucoup plus de bancs mémoire que celle pour l'application *MJPEG*.

6.2.3.3 Événements générés

Le TABLEAU 6.1 donne pour chaque expérience la quantité d'événements générés pendant la simulation. Pour chaque expérience, les nombres d'instructions, d'accès, d'accès spéciaux (LDLs et STCs), d'acquittements mémoire et d'opérations de cache sont aussi indiquées. Les différentes quantités sont données en millions d'unités par processeur afin d'obtenir des chiffres comparables. Les opérations des caches regroupent les mises à jour, chargements de lignes et invalidations. Elles sont composées majoritairement de mises à jour dues à des écritures locales du processeur.

Pour l'application *MJPEG*, on observe des nombres similaires avec une baisse pour *MJPEG_4*. Ces nombres sont cohérents avec l'efficacité de la parallélisation donnée précédemment.

Pour l'application *OCEAN*, on observe une évolution similaire. Néanmoins, pour *OCEAN_32*, il y a une grosse augmentation des acquittements mémoire et des opérations de cache qui est due au faible taux de regroupement des écritures. Dans cette application, le nombre d'accès spéciaux servant à implanter les prises de verrous logiciels augmentent de manière non linéaire avec le nombre de processeurs. La majorité de ces accès sont des LDLs, ce qui nous indique que la grande partie des tentatives de prises de verrous nécessite plusieurs essais. L'application se synchronisant régulièrement, ce phénomène est logique.

Expérience	MJPEG_1	MJPEG_2	MJPEG_3	MJPEG_4
Événements	1020	933	944	810
Instructions	526	484	487	418
Accès / LDL + STC	181 / 0,062	166 / 0,056	166 / 0,056	143 / 0,048
Acq. mem	47,6	42,8	45,4	38,7
Op. Cache	40,4	36,1	38,8	32,7
Expérience	**OCEAN_4**	**OCEAN_8**	**OCEAN_16**	**OCEAN_32**
Événements	1065	965	813	802
Instructions	663	610	494	449
Accès / LDL + STC	145 / 0,013	128 / 0,047	116 / 0,104	102 / 0,423
Acq. mem	37,4	33,3	29,4	49,8
Op. Cache	37,2	32,9	28,8	49,1

TABLEAU 6.1 - Événements générés par les simulations (en millions / processeur)

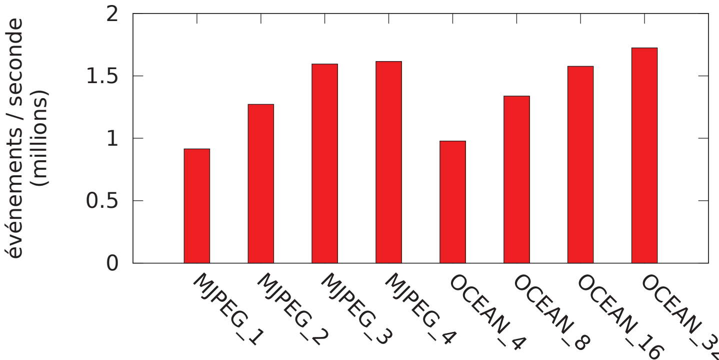


FIGURE 6.8 - Débit moyen d'événements générés pendant les simulations

La FIGURE 6.8 indique le débit virtuel d'événements de la simulation initiale (le nombre d'événements divisé par le temps de simulation). Pour une même application, celui-ci augmente avec le nombre de processeurs car le temps de simulation n'est pas linéaire par rapport au nombre de pro-

cesseurs alors que le nombre d'événements générés l'est quasiment. On peut ainsi s'attendre à un impact croissant (avec le nombre de processeurs) des différents traitements puisque la simulation est plus efficace (du point de vue de la quantité d'événements générés).

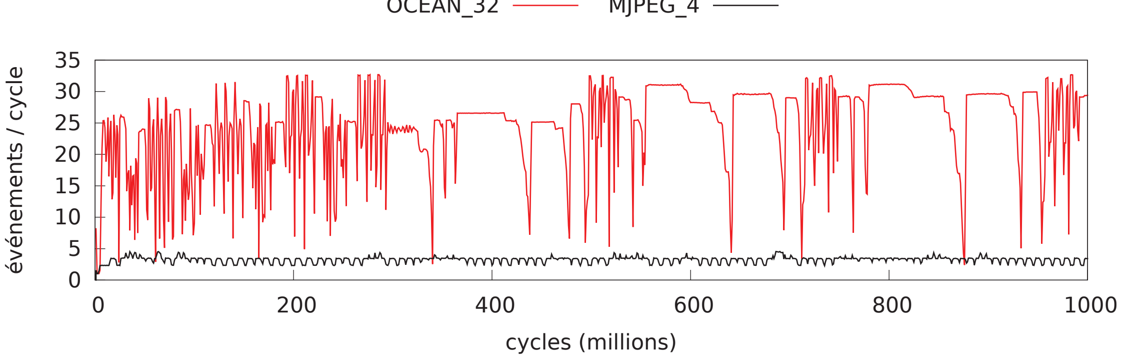


FIGURE 6.9 - Taux de génération d'événements (moyenne sur 1 million de cycles)

La FIGURE 6.9 montre l'évolution du nombre d'événements générés (par tranche de 1 million de cycles) durant la simulation. Ce nombre est stable dans le cas de l'application *MJPEG*, tandis que pour l'application *OCEAN* il y a de très fortes chutes lors des synchronisations des différents *threads*.

6.2.4 Détection des violations du modèle de consistance mémoire

L'algorithme de VCM ne détecte pas d'erreur dans les plates-formes simulées. Cela est normal car le modèle TSO est bien implanté dans celles-ci.

Afin de vérifier que l'algorithme est bien capable de détecter les violations, des simulations ont été faites en modifiant certains composants simulés.

6.2.4.1 Vérification d'un modèle plus contraint

Bien que le modèle TSO soit implanté dans les plates-formes, la VCM du modèle SC (plus contraint) n'échoue pas. Cela est dû au logiciel exécuté : celui-ci effectue les barrières mémoire adéquates afin de garantir une exécution correcte.

Comme ce qui est vérifié est uniquement le comportement de l'exécution du logiciel, il est normal que, dans le cas d'un logiciel prévu pour une exécution sur une plate-forme relâchée, cette vérification se déroule sans détection d'erreur.

6.2.4.2 Erreur de consistance

Par contre, en modifiant la manière dont sont gérés les tampons d'écritures, nous avons autorisé les lectures à doubler des écritures à la même ligne de cache, mais à un mot distinct de la ligne.

Comme nous considérons qu'une case mémoire est une ligne entière de cache, cela n'est alors pas correct. Cela est bien détecté par l'algorithme. En passant à des cases mémoire de la taille d'un mot, il n'y a alors plus d'erreurs détectées.

6.2.4.3 Erreur dans les accès atomiques

La gestion des accès spéciaux à été modifiée pour permettre aux STCs de se faire même si une écriture a eu lieu depuis le dernier LDL. Ces erreurs ont bien été détectées par l'algorithme.

De manière générale, l'historique mis en place dans l'algorithme permet à chaque fois d'étudier en détail le scénario ayant mené à la détection d'une erreur.

6.3 Évaluation du coût de la vérification de la consistance mémoire

Dans cette partie, nous évaluons le coût des différents traitements nécessaires pour effectuer la VCM avec la simulation d'une plate-forme matérielle. Les résultats sont tirés de la simulation des mêmes plates-formes en activant ou non les différentes étapes du traitement des événements générés :

- la simulation sans traitement des événements (uniquement la génération de ceux-ci),
- la simulation en faisant le traitement des événements dans l'environnement d'analyse,
- des simulations en effectuant en plus différentes configurations de l'algorithme de VCM.

Certaines configurations ont aussi été exécutées en faisant le traitement des événements (et la VCM) dans un second *thread*.

L'évaluation du coût des étapes du traitement résulte donc de la différence entre les durées de deux simulations différentes. Les résultats présentés ici sont des moyennes effectuées sur plusieurs simulations. L'ensemble des résultats est dans l'annexe D.

6.3.1 Impact sur les simulations

6.3.1.1 Impact global de la vérification de la consistance mémoire

Le Tableau 6.2 indique l'augmentation du temps de simulation lorsque la VCM est effectuée pendant celle-ci. Pour chaque expérience, deux pourcentages sont donnés. Le premier correspond à l'impact lorsque la simulation et l'analyse sont faites dans le même *thread*. Le deuxième correspond à l'impact lorsque l'analyse est faite dans un second *thread*.

Les pourcentages séquentiels augmentent avec le nombre de processeurs puisque le temps de simulation n'évolue pas de manière linéaire. On peut remarquer que l'utilisation d'un deuxième *thread* permet d'obtenir un impact beaucoup plus limité (de quelques pourcents). Nous analysons le détails de ces coûts dans la suite de cette partie.

MJPEG_1	MJPEG_2	MJPEG_3	MJPEG_4
20,5 % / 1,9 %	28,5 % / 4,5 %	35,7 % / 6,2 %	37,3 % / 6,8 %

OCEAN_4	OCEAN_8	OCEAN_16	OCEAN_32
21,8 % / 4,1 %	27,2 % / 2,1 %	39,4 % / 11,9 %	43,2 % / 5,8 %

Tableau 6.2 - Impact de la VCM sur le temps de simulation

6.3.1.2 Génération et traitement des événements

La Figure 6.10 indique le coût moyen par événement de la génération et du traitement des événements. Le coût de la génération des événements est calculé en mesurant l'écart entre le temps de simulation initial et celui dans lequel les événements sont générés par les modèles de simulation mais sans que ceux-ci soient ensuite traités. Le coût de la génération parallèle correspond au surcoût par événement entre le temps de simulation initial et celui où la VCM est faite dans un second *thread*. Le coût parallèle, qui correspond au pourcentage donné dans le Tableau 6.2, est à chaque fois légèrement supérieur au coût de génération seul. Pour les simulations *MJPEG*, les temps obtenus sur un ordinateur équipé d'un *Core 2 Duo* sont aussi donnés.

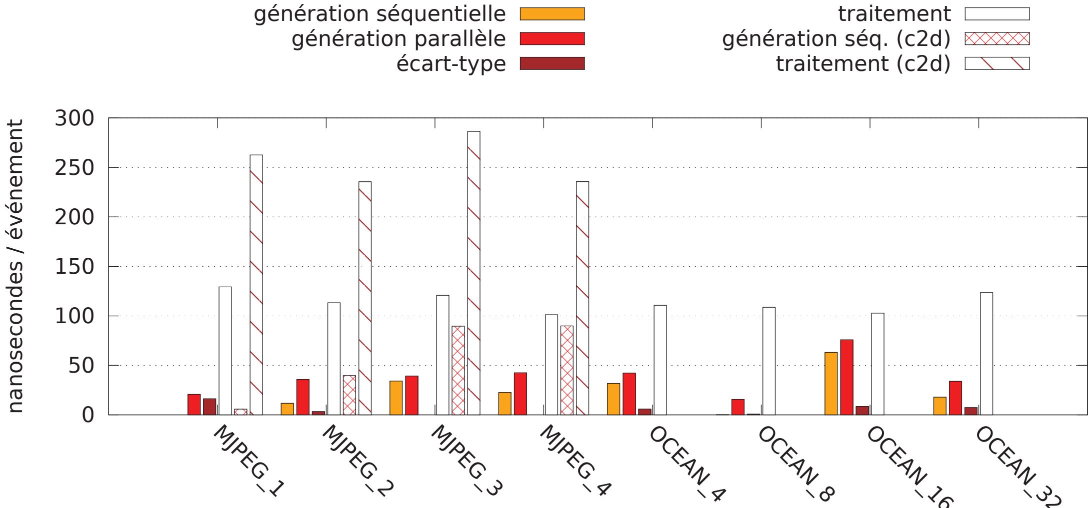


FIGURE 6.10 - Coût de traitement par événement

Même si l'ordre de grandeur du coût par événement est le même, de grandes variations existent entre les différentes expériences. En particulier, il est plus important dans l'expérience *OCEAN_16*, ce qui explique l'impact de 11,9 % indiqué dans le tableau. Il est difficile d'expliquer pourquoi il y a de telles variations. Elles sont probablement dues à des effets de placement de données en cache ou de prédiction de branchement.

Pour l'expérience *MJPEG_1*, le temps de génération séquentielle n'est pas donné car sur le processeur *Core i5* la durée de la simulation en générant les événements est paradoxalement plus rapide que sans les générer. Ce n'est pas le cas sur le *Core 2 Duo*.

L'écart-type (divisé par le nombre d'événements) observé sur les différentes répétitions de la simulation initiale est indiqué sur la figure. Celui-ci est dans certains cas du même ordre de grandeur que le temps de génération, en particulier pour l'expérience *MJPEG_1*.

Le coût de traitement par événement est aussi indiqué. Celui-ci est mesuré en comparant le temps de simulation générant les événements et celui où les événements sont de plus traités par l'environnement d'analyse. Il correspond à l'exécution des différents traitants (trois par événement) dans l'environnement d'analyse sans effectuer l'algorithme de VCM. Ce coût est similaire pour toutes les expériences, malgré des variations d'environ 10 %. Ces variations étant du même ordre de grandeur que les écarts-types observés, il n'est pas possible d'en tirer des conclusions. On observe néanmoins qu'il n'y a pas de relation entre le nombre de composants générant des événements (en particulier les processeurs) et le coût de traitement d'un événement. Ainsi l'environnement d'analyse des événements semble bien passer à l'échelle.

6.3.2 Coût de la vérification de la consistance mémoire

La FIGURE 6.11 montre l'évolution du nombre d'événements générés par tranche de 1 million de cycles pour l'exemple *OCEAN* sur 32 processeurs. Malgré de grosse chutes dues aux synchronisations de l'application, on distingue deux zones. Une première zone (jusqu'à environ 350 millions de cycles) où 2,5 millions d'accès sont générés tous les millions de cycles simulés. Ensuite le taux est plus près de 3,5 millions d'accès tous les millions de cycles simulés. La figure donne aussi le nombre cumulé d'accès. On observe une légère augmentation de la pente de ce nombre correspondant à l'augmentation du débit aux alentours de 350 millions de cycles.

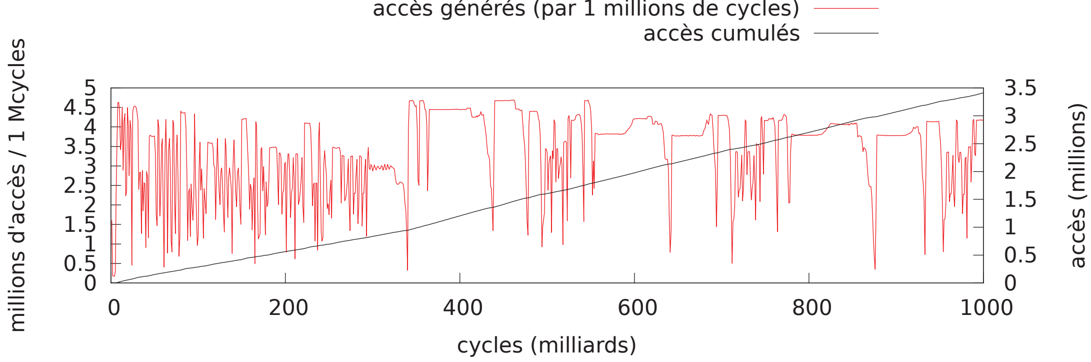


FIGURE 6.11 - Évolution de la génération d'accès pour *OCEAN_32*

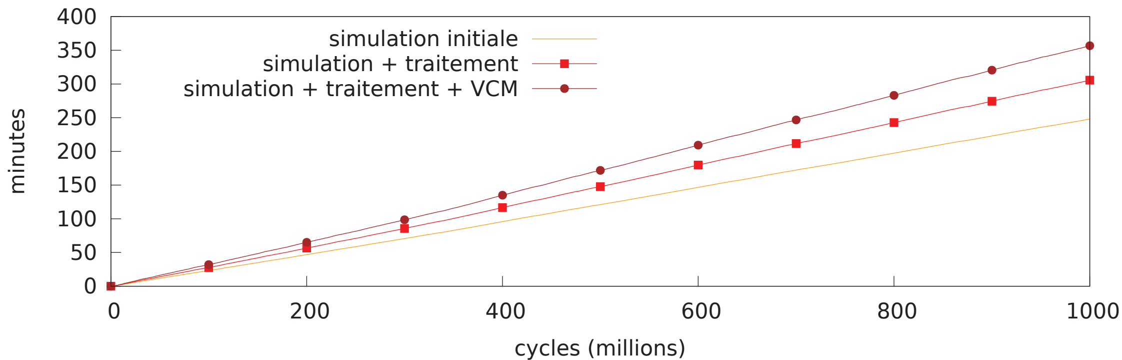


FIGURE 6.12 - Évolution du temps des simulations pour *OCEAN_32*

6.3.2.1 Évolution linéaire de la vérification de la consistance mémoire

La FIGURE 6.12 donne l'évolution du temps de la simulation initiale, de la simulation avec traitement des événements et de la simulation avec la VCM. Les évolutions sont linéaires mais on peut observer le léger accroissement de la pente de la courbe de la VCM vers 350 millions de cycles. Cet accroissement est une conséquence logique de l'augmentation du débit d'accès que l'on observe dans la la FIGURE 6.11.

En observant les écarts entre les courbes, on s'aperçoit que le coût de la VCM est légèrement inférieur au coût de la génération et du traitement des événements qu'il faut faire au préalable.

La FIGURE 6.13 résume le coût de la VCM pour chacune des simulations. Dans cette figure, les temps obtenus sur un processeur *Core 2 Duo E4500* sont indiqués pour les expériences *MJPEG* (avec le préfixe « c2d »). Les coûts indiqués sont les coûts moyens par accès bien que les entrées de l'algorithme ne soient pas limitées aux accès (les instructions et les opérations des caches sont aussi traitées).

On observe que le coût de la VCM est du même ordre de grandeur pour toutes les simulations réalisées sur la même machine (*Core i5* ou *Core 2 Duo*), ce qui confirme que la complexité de l'algorithme ne croît pas avec le nombre de processeurs. Néanmoins le coût de la vérification pour *OCEAN_32* est presque deux fois supérieur à celui pour *OCEAN_16*. Nous suspectons que cela est dû au taux beaucoup plus élevé d'opérations de cache et d'acquittements d'écriture en mémoire. Chaque acquittement mémoire implique en particulier de manipuler la case mémoire accédée : il est nécessaire de restaurer l'accès en écriture contenu dans la case mémoire si celle-ci à été supprimée temporairement.

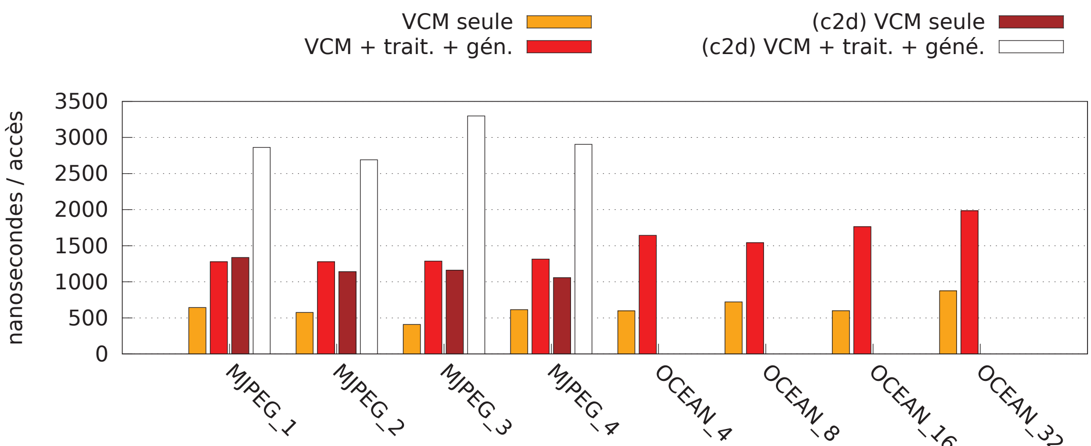


FIGURE 6.13 - Coût moyen de l'algorithme de VCM par accès

6.3.2.2 Opérations effectuées par l'algorithme

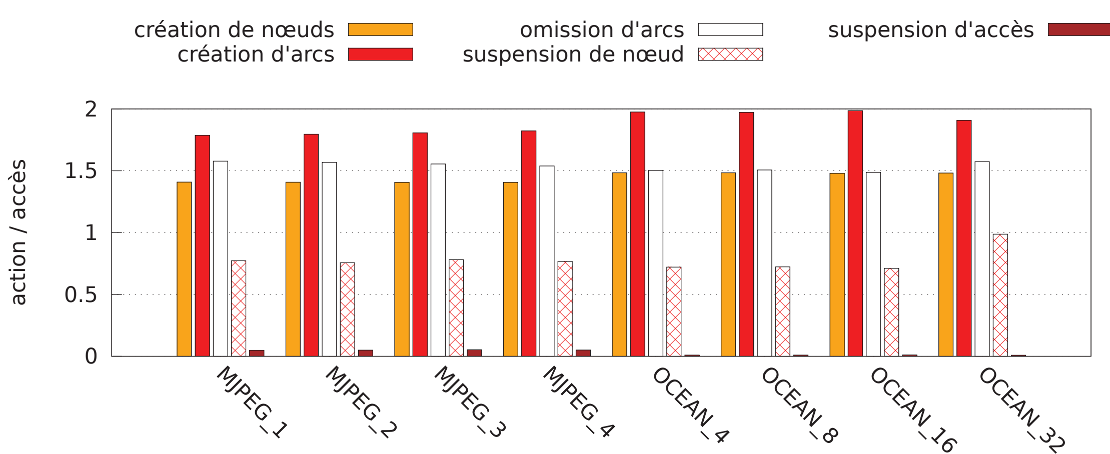


FIGURE 6.14 - Statistiques par accès

La FIGURE 6.14 indique quelques statistiques concernant les opérations effectuées durant la VCM. La figure donne des moyennes par accès traité. Les différentes statistiques sont très similaires d'une expérience à l'autre. Néanmoins on observe que le nombre d'arcs et de nœuds créés est légèrement supérieur dans le cas de l'application *OCEAN*.

L'omission d'un arc signifie qu'un arc n'a pas été créé car son nœud source était déjà supprimé. Près de la moitié des arcs n'a ainsi pas été créée.

Les suspensions de nœuds correspondent aux nœuds isolés supprimés temporairement. La suspension d'accès correspond à la suppression temporaire des références formant un accès quand leurs nœuds sont eux-mêmes supprimés (temporairement ou définitivement). On observe aussi que le nombre de suspensions d'accès est très inférieur au nombre de suspensions de nœud.

Bien que nous n'ayons pas implanté cette solution, il serait intéressant de voir si la suspension d'un nœud n'est pas pénalisante lorsque sa référence (c'est à dire son accès) ne peut pas être supprimée en même temps. Cela aurait un coût très limité au niveau de l'occupation mémoire car la place occupée par une référence est de toute façon comparable à celle du nœud. Cela éviterait la suspension (et la restauration associée) de nombreux nœuds sans provoquer d'explosion de la mémoire.

6.3.2.3 Impact de la méthode de détection des cycles

Les deux politiques de détection des cycles ont été implantées (détection « moindre effort » et « régulière ») (cf. 3.1.4). Quelques centaines de nœuds seulement sont parcourus lors de la détection minimale. Le surcoût de la détection régulière a été mesuré comme négligeable malgré le fait que chaque nœud créé soit parcouru en moyenne 1.5 fois.

6.3.2.4 Impact du modèle de consistance

Les expériences faites en vérifiant un modèle très relâché (RMO) prenant en compte les dépendances de données à travers chaque registre de chaque processeur ont mené à un coût supplémentaire très léger. Celui-ci est de l'ordre de 0,5 % par rapport à la vérification du modèle TSO. Lors de la vérification de ce modèle, 2 fois plus de nœuds et 1,5 fois plus d'arcs sont créés que pour le modèle TSO. La complexité de l'algorithme est en effet majoritairement comprise dans le traitement des accès.

6.3.3 Occupation mémoire

6.3.3.1 Évolution de l'occupation mémoire

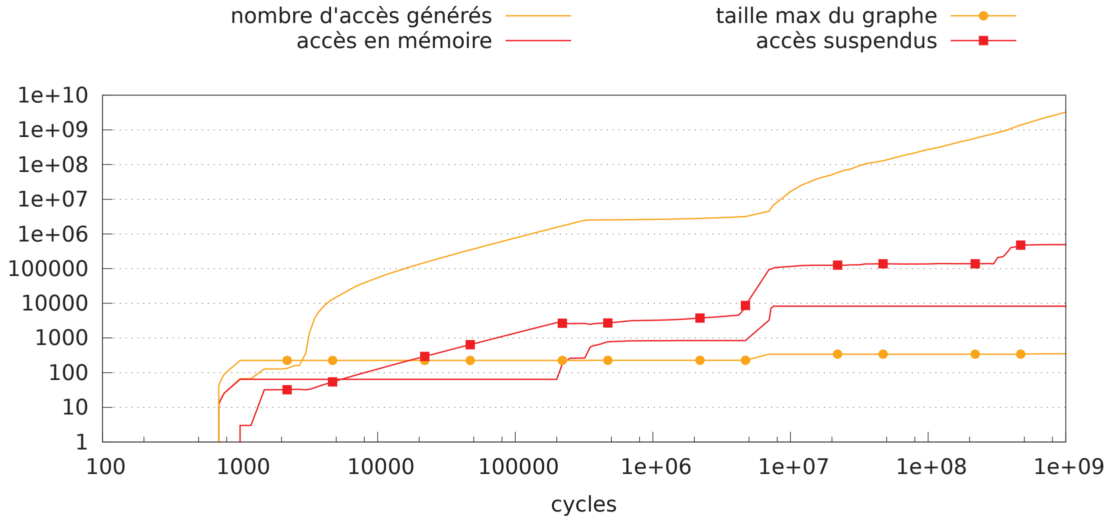


FIGURE 6.15 - Évolution de la mémoire utilisée lors de la simulation

La FIGURE 6.15 montre l'évolution de la partie dynamique de l'occupation mémoire de la VCM lors de la simulation *OCEAN_32*. Dans cette figure, les deux échelles sont logarithmiques. La figure indique l'évolution du nombre d'accès générés pendant la simulation ainsi que l'évolution de la taille du graphe, du nombre d'accès conservés en mémoire pendant la VCM. La taille du graphe et le nombre d'accès indiqués sont en fait les maxima atteints pour éviter les effets de l'échantillonnage sur la figure. En particulier, la taille du graphe varie beaucoup.

Le nombre d'accès se stabilise au dessus de 8000 (maximum de 8271) ce qui correspond à la taille cumulée des caches dans le système ($32 \times 256 = 8192$ lignes). La taille du graphe atteint au maximum 350 éléments (nœuds et arcs). Dans un premier temps la taille du graphe dépasse le nombre d'accès car durant la phase d'initialisation de l'OS, les caches sont très peu utilisés. L'application démarre vers 250 000 cycles, ce qui provoque une augmentation du nombre d'accès en mémoire au fur et à mesure que les différents processeurs utilisent leur cache. Le calcul commence réellement vers 5 millions de cycles.

La figure montre aussi l'évolution du nombre d'accès suspendus : c'est à dire le nombre d'écritures (et quelques accès I/Os) dont le nœud G a été supprimé temporairement ainsi que les références qui forment l'accès correspondant. Sur la figure, on distingue plusieurs paliers correspondant aux différentes étapes du calcul. Le nombre d'accès suspendus correspond environ aux nombres de cases mémoire utilisées depuis le début de la simulation et dépend entièrement de l'application. Sans suppression des nœuds isolés (et de leurs références), l'occupation mémoire nécessaire serait dans ce cas deux ordres de grandeur plus élevée.

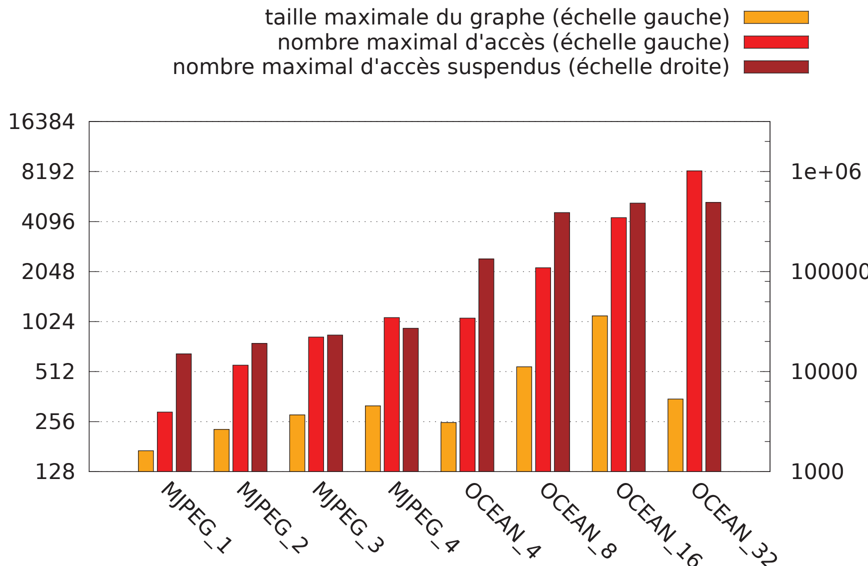


FIGURE 6.16 - Maxima atteints lors des simulations

La FIGURE 6.16 donne les maxima atteints pour chaque simulation ainsi que la taille des caches. Le nombre d'accès suspendus utilise l'échelle de droite, car celui-ci est beaucoup plus grand que les autres. Pour une même application, celui-ci augmente car des *threads* applicatifs supplémentaires sont utilisés ainsi que des structures de données associées (dans l'OS et dans l'application).

Le nombre maximal d'accès en mémoire est à chaque fois très légèrement au dessus de la taille des caches (256 lignes par processeur). Ceci était attendu, car l'algorithme conserve l'ensemble des accès en cache, ainsi que les accès en cours de réalisation : ces derniers peuvent être dans les tampons d'écriture ou dans les différents canaux de communication. Pour une même application, la taille du graphe semble augmenter linéairement avec le nombre de processeurs. Néanmoins, pour la simulation *OCEAN_32*, celle-ci est très faible. Cela est dû à une saturation de certains bancs mémoires très accédés et qui limitent l'application. En divisant par deux les bancs mémoire en question (permettant un accès concurrent à ceux-ci), la taille du graphe maximale obtenue remonte ainsi à plus de 1700.

6.3.3.2 Impact des paramètres architecturaux

Afin d'illustrer l'impact des différents paramètres architecturaux sur l'occupation mémoire de l'algorithme, nous avons effectué des simulations en faisant varier ceux-ci. Un seul processeur a été simulé pour mettre en évidence les variations.

la FIGURE 6.17 montre l'occupation mémoire obtenue en faisant varier la taille du cache. Dans ces simulations, les différents tampons utilisés pour masquer la latence des écritures sont désactivés. La figure montre l'évolution du nombre d'accès en mémoire au fur et à mesure de l'émission des accès par le processeur. La figure commence à 22000 accès car, auparavant le logiciel exécuté

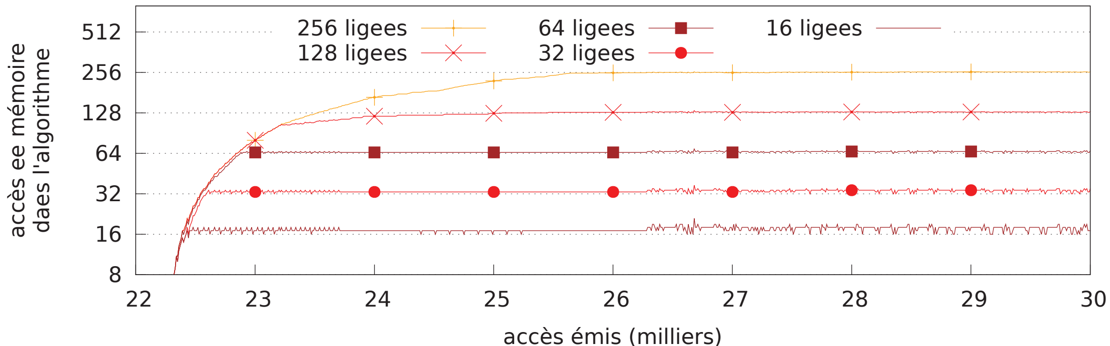
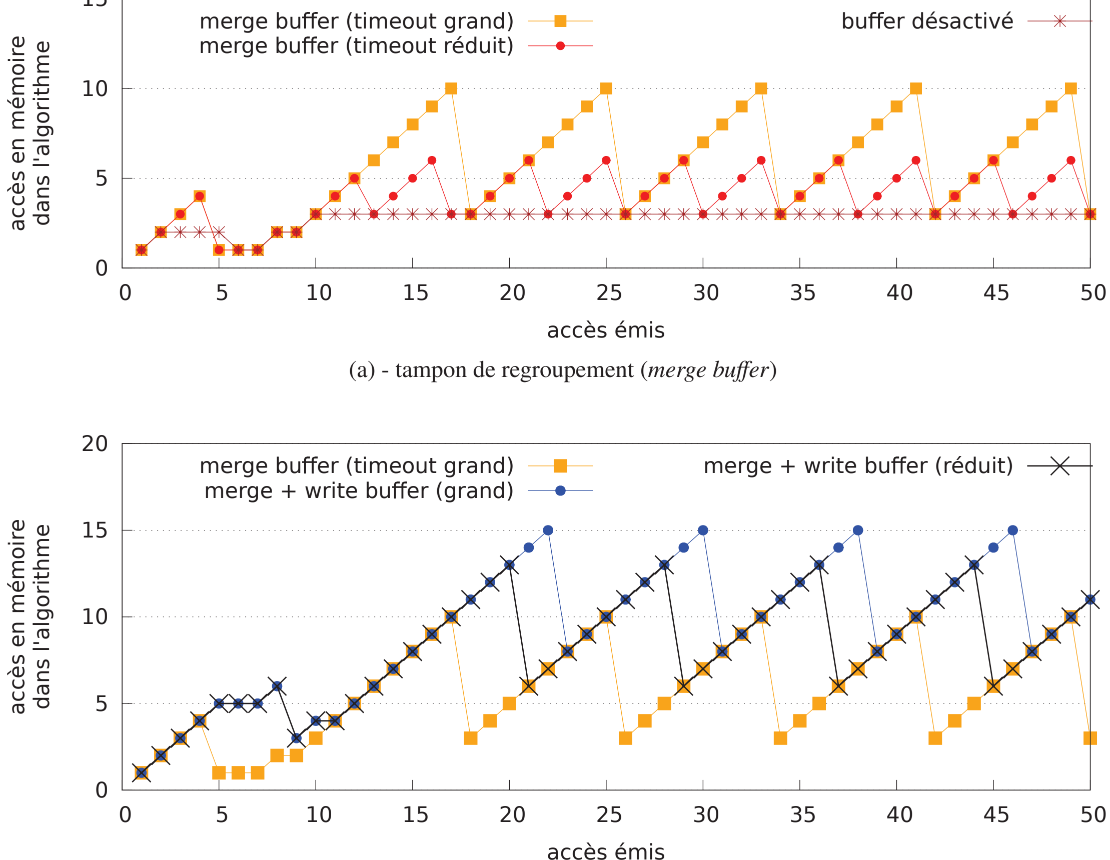


FIGURE 6.17 - Impact de la taille des caches sur l'occupation mémoire

par le processeur ne fait quasiment que des écritures et n'utilise donc quasiment pas son cache. On peut voir que le nombre d'accès en mémoire se stabilise autour de la taille du cache.

La FIGURE 6.18 montre l'occupation mémoire obtenue en faisant varier la taille des tampons. Dans ces simulations, le cache est désactivé. La partie de la simulation qui est montrée correspond à un moment où le logiciel effectue une initialisation d'une partie de la mémoire : celui-ci écrit donc dans des cases mémoire contigües.



(a) - tampon de regroupement (*merge buffer*)

(b) - tampon de regroupement (*merge buffer*) et tampon d'écriture (*write buffer*)

FIGURE 6.18 - Impact des tampons sur l'occupation mémoire

La FIGURE 6.18a montre l'occupation durant trois simulations utilisant un tampon de regroupement configuré différemment. Le tampon de regroupement consiste à regrouper les écritures contigües afin de réduire les communications. Afin de garantir qu'une écriture ne restera pas indéfiniment dans ce tampon, un *timeout* permet de forcer le tampon à se vider. La figure illustre trois configurations : tampon de regroupement désactivé, activé avec un *timeout* réduit et activé normalement. Ces réglages font que le tampon est vidé quand il contient plus ou moins d'écritures.

Quand le *timeout* est suffisamment grand, le tampon est vidé quand la prochaine écriture ne peut pas être fusionnée avec le contenu actuel. Dans nos architecture, sa taille maximale est une ligne de cache, soit 8 mots, ce qui explique pourquoi le nombre d'accès en mémoire retombe tous les 8 accès.

Dans la FIGURE 6.18b, un tampon d'écriture classique est activé. Le tampon de regroupement bloque en effet l'écriture suivante lorsqu'il est en train de se vider. C'est pour cette raison que dans la figure précédente le nombre d'accès retombe toujours au même niveau.

En activant un tampon d'écriture, les écritures peuvent être mises de côté en attendant que le tampon de regroupement se vide. On observe dans la figure, que le nombre plancher d'accès en mémoire augmente bien lorsqu'on active le tampon d'écriture. Deux capacités du tampon d'écriture ont été configurées. L'une étant trop faible, elle permet une certaine amélioration mais le tampon d'écriture devient plein avant que le tampon de regroupement ne soit à nouveau prêt.

6.4 Conclusion

Dans ce chapitre, nous avons présenté les implantations faites pour vérifier nos propositions. Nous avons illustré les propositions de cette thèse par des expérimentations sur un simulateur de MPSoCs. L'environnement d'analyse a été implanté ainsi que la vérification de deux modèles de consistance.

Les expériences effectuées montrent que l'algorithme de VCM proposé passe bien à l'échelle. Celui-ci a une occupation mémoire qui permet d'envisager l'analyse d'exécution non limitée puisque cette occupation est stable avec l'accroissement du nombre d'accès. La durée de la VCM ne dépend en particulier pas du nombre de processeurs dans la plate-forme matérielle. L'environnement de traitement des événements mis en place passe aussi à l'échelle.

Conclusion

Conclusion générale

Dans cette thèse, nous avons proposé une méthode de vérification de la consistance mémoire adaptée aux prototypes virtuels simulés. Pour réaliser cette vérification, nous nous appuyons sur un environnement générique permettant de réaliser des analyses à partir d'événements extraits d'une simulation d'un prototype virtuel.

La vérification de la consistance mémoire d'une exécution est généralement réalisée sur de courtes exécutions à cause de manque d'informations disponibles. Des informations sur les accès à la mémoire sont en effet cruciales pour réaliser la vérification en temps linéaire. Les méthodes classiques consistent à construire un graphe représentant l'exécution complète et à vérifier ensuite l'absence de cycles, synonymes de violations de la consistance mémoire. Ces méthodes atteignent leurs limites sur de longues exécutions à cause de l'occupation mémoire résultante du graphe.

Méthode efficace de vérification de la consistance mémoire passant à l'échelle

Afin de pouvoir passer à l'échelle, le défi consiste à limiter l'occupation mémoire de la vérification tout en conservant un algorithme ayant un coût linéaire en temps de calcul. Dans le chapitre 3, nous avons proposé une méthode basée sur l'utilisation d'un graphe dynamique permettant de vérifier en temps linéaire la consistance mémoire qui a été implantée et vérifiée expérimentalement. Ce graphe est formé de nœuds représentant les accès à la mémoire et d'arcs représentant les contraintes d'ordonnancement entre les différents accès pour en former une séquence légale.

En disposant des informations adéquates, nous avons montré que les éléments de ce graphe peuvent être supprimés au fur et à mesure que l'exécution du programme se déroule. Dans le chapitre 4, nous avons de plus défini comment réduire la taille du graphe en supprimant temporairement certains éléments. Nous avons pressenti que l'occupation mémoire résiduelle dépendrait principalement des paramètres architecturaux du système matériel exécutant le programme tels que la taille des caches. Cela a été confirmé par les différentes expériences réalisées à cet effet.

Prise en compte des spécificités des modèles de consistance mémoire

Les modèles de consistance mémoire ne se limitent pas à spécifier les contraintes d'ordonnancement des écritures et des lectures à la mémoire partagée. Les propriétés des différentes zones adressables ne sont pas toutes identiques. Les accès aux périphériques sont en particulier traités de manière spécifique.

Dans le chapitre 4, nous avons décrit comment distinguer les propriétés de chaque segment mémoire. Nous avons par ailleurs aussi montré comment vérifier les propriétés d'accès spéciaux tels que les LdLs et StCs. La vérification de ces différentes caractéristiques a été mise en place expérimentalement lors de la simulation d'une plate-forme matérielle utilisant différents périphériques, de la mémoire cohérente suivant un modèle de consistance de type TSO.

Informations nécessaires à la vérification efficace de la consistance mémoire

L'algorithme proposé de vérification de la consistance mémoire nécessite la connaissance des informations utilisées classiquement : l'ordre dans lequel est exécuté le programme par les processeurs, l'ordre des écritures à chaque case mémoire et l'association de chaque lecture avec l'écriture dont elle a lu la donnée. Ces informations sont nécessaires pour avoir une complexité linéaire, mais sont en pratique impossibles à obtenir pour un programme quelconque exécuté sur un système réel. Dans le cadre du prototypage virtuel, ces informations sont néanmoins accessibles.

En plus de ces informations, l'algorithme proposé a aussi besoin de l'information, pour chaque écriture, du moment où celle-ci disparaît : c'est à dire du moment où elle ne peut plus être lue. Cette dernière information est obtenue en traçant le contenu des différents tampons intermédiaires, caches et mémoires de la plate-forme matérielle. Cette information n'est raisonnablement pas accessible dans un système réel mais ne pose pas de problème à obtenir dans un prototype virtuel.

Infrastructure générique d'analyse de simulations de systèmes multi-processeurs

La mise en place de l'analyse du déroulement d'une simulation d'une plate-forme multi-processeurs peut être difficile à cause de la quantité d'informations à rassembler. Dans le chapitre 5, nous avons proposé un environnement de trace et d'analyse permettant de mettre en place une analyse centrée sur le traitement d'événements.

Cet environnement permet d'abstraire efficacement la plate-forme matérielle simulée. Grâce à l'inclusion de relations de cause à effet dans les événements, l'analyse ne nécessite pas de corréler les informations provenant de différents composants du système simulés. Cet environnement a été implanté et utilisé pour effectuer la vérification de la consistance.

Perspectives

Le travail effectué dans cette thèse permet d'envisager la vérification de la consistance mémoire en même temps qu'un prototype virtuel est simulé. La méthode proposée passe à l'échelle car l'occupation mémoire ainsi que le coût algorithmique sont limités. Néanmoins ce coût n'est pas négligeable : il représente de 30 à 40 pourcents du temps des simulations effectuées. Ce temps peut être masqué en grande partie par une exécution de l'analyse en parallèle. À cause de la grande quantité d'informations nécessaires, le simple coût de leur extraction représente toutefois plusieurs pourcents du temps de simulation.

L'utilisation de la méthode proposée lors d'une simulation plus rapide, par exemple en utilisant des modèles de simulation plus abstraits, implique une augmentation de son coût relatif. Dans cette optique, il est donc intéressant d'étudier la faisabilité de la parallélisation de l'environnement d'analyse et de l'algorithme afin que l'analyse ne soit pas un facteur limitant.

La méthode de vérification de la consistance mémoire proposée permet de prendre en compte un certain nombre de caractéristiques d'un modèle de consistance. Elle n'est en particulier applicable que dans le cas où les propriétés des différents segments mémoire sont statiques. Dans les architectures utilisées aujourd'hui, ces propriétés sont définies dans les tables de page de paramétrage des mécanismes de mémoire virtuelle. Elles sont donc en réalité dynamiques. Il faut donc étudier l'impact des modifications des propriétés des cases mémoire sur l'algorithme ainsi que définir les informations supplémentaires à obtenir pour pouvoir les prendre en compte.

De manière plus générale, la méthode d'analyse proposée peut s'appliquer à tout problème pouvant se traduire sous forme similaire : la vérification qu'il n'y a pas de cycles dans un graphe. Ainsi il semble intéressant de voir s'il n'est pas possible d'utiliser une méthode semblable pour

réaliser d'autres analyses. Les problèmes concernant les accès à la mémoire ne sont en particulier pas limités à la consistance. La consistance ne garantit en effet en aucun cas le bon fonctionnement du logiciel. Les *data races* sont en particulier un problème très important de la programmation concurrente qui n'est en rien réglé par le respect d'un modèle de consistance (même la consistance séquentielle). Il serait ainsi intéressant d'étudier les possibilités d'adapter la méthode utilisée pour la VCM pour s'attaquer à la détection des *data races*.

Bibliographie

[AH90] Sarita V. Adve et Mark D. Hill. « Weak ordering - a new definition ». Dans : *Proceedings of the 17th annual international symposium on Computer Architecture*. ISCA '90. Seattle, Washington, United States : ACM, 1990, p. 2–14.

[Aha+93] Mustaque Ahamad *et al.* « The power of processor consistency ». Dans : *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. SPAA '93. Velen, Germany : ACM, 1993, p. 251–260.

[Alg+09] Jade Alglave *et al.* « The semantics of power and ARM multiprocessor machine code ». Dans : *Proceedings of the 4th workshop on Declarative aspects of multicore programming*. DAMP '09. Savannah, GA, USA : ACM, 2009, p. 13–24.

[AM06] Arvind Arvind et Jan-Willem Maessen. « Memory Model = Instruction Reordering + Store Atomicity ». Dans : *Proceedings of the 33rd annual international symposium on Computer Architecture*. ISCA '06. Washington, DC, USA : IEEE Computer Society, 2006, p. 29–40.

[AMD64] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices. Mar. 2012.

[AMP96] R. Alur, K. McMillan et D. Peled. « Model-checking of correctness conditions for concurrent objects ». Dans : *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*. Juil. 1996, p. 219 –228.

[ARMv7] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. ARM Limited. Avr. 2007.

[Ati+10] Mohamed Faouzi Atig *et al.* « On the verification problem for weak memory models ». Dans : *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '10. Madrid, Spain : ACM, 2010, p. 7–18.

[CD06] V. Chennareddy et J.K. Deka. « Formally Verifying the Distributed Shared Memory Weak Consistency Models ». Dans : *Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on*. Déc. 2006, p. 455 –460.

[CH01] Anne E. Condon et Alan J. Hu. « Automatable verification of sequential consistency ». Dans : *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '01. Crete Island, Greece : ACM, 2001, p. 113–121.

[Che+09] Yunji Chen *et al.* « Fast complete memory consistency verification ». Dans : *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. Fév. 2009, p. 381 –392.

[CI08] Nathan Chong et Samin Ishtiaq. « Reasoning about the ARM weakly consistent memory model ». Dans : *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*. MSPC '08. Seattle, Washington : ACM, 2008, p. 16–19.

[CL02] Harold W. Cain et Mikko H. Lipasti. « Verifying sequential consistency using vector clocks ». Dans : *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '02. Winnipeg, Manitoba, Canada : ACM, 2002, p. 153–154.

[CLS03] Jason F. Cantin, Mikko H. Lipasti et James E. Smith. « The complexity of verifying memory coherence ». Dans : *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '03. San Diego, California, USA : ACM, 2003, p. 254–255.

[CLS05] J.F. Cantin, M.H. Lipasti et J.E. Smith. « The complexity of verifying memory coherence and consistency ». Dans : *Parallel and Distributed Systems, IEEE Transactions on* 16.7 (juil. 2005), p. 663 –671.

[Con+99] A.E. Condon *et al.* « Using Lamport clocks to reason about relaxed memory models ». Dans : *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. Jan. 1999, p. 270 –278.

[CSB93] F. Corella, J. M. Stone et C. M. Barton. *A formal specification of the PowerPC shared memory architecture*. Rap. tech. IBM Research Division, jan. 1993.

[DSB86] M. Dubois, C. Scheurich et F. Briggs. « Memory access buffering in multiprocessors ». Dans : *Proceedings of the 13th annual international symposium on Computer architecture*. ISCA '86. Tokyo, Japan : IEEE Computer Society Press, 1986, p. 434–442.

[DSB88] M. Dubois, C. Scheurich et F.A. Briggs. « Synchronization, coherence, and event ordering in multiprocessors ». Dans : *Computer* 21.2 (fév. 1988), p. 9 –21.

[Gao+12] Jianliang Gao *et al.* « A clustering-based scheme for concurrent trace in debugging NoC-based multicore systems ». Dans : *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. Mar. 2012, p. 27 –32.

[Gha+90] K. Gharachorloo *et al.* « Memory consistency and event ordering in scalable shared-memory multiprocessors ». Dans : *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. Mai 1990, p. 15 –26.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. Rap. tech. 1995.

[GK92] P.B. Gibbons et E. Korach. « The complexity of sequential consistency ». Dans : *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*. Déc. 1992, p. 317 –325.

[GK94] Phillip B. Gibbons et Ephraim Korach. « On testing cache-coherent shared memories ». Dans : *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. SPAA '94. Cape May, New Jersey, United States : ACM, 1994, p. 177–188.

[GK97] Phillip B. Gibbons et Ephraim Korach. « Testing shared memories ». Dans : *SIAM Journal on Computing* 26 (1997), p. 1208–1244.

[GP09] Xavier GUÉRIN et Frédéric PÉTROT. « A system framework for the design of embedded software targeting heterogeneous multi-core SoCs ». Dans : *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE. 2009, p. 153–160.

[GPS04] Alex GONTMAKHER, Sergey POLYAKOV et Assaf SCHUSTER. « Complexity Of Verifying Java Shared Memory Execution. » Dans : *Parallel Processing Letters* 13.4 (10 sept. 2004), p. 721–733.

[Hae+12] Bernhard HAEUPLER *et al.* « Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance ». Dans : *ACM Trans. Algorithms* 8.1 (jan. 2012), 3:1–3:33.

[Han+04] S. HANGAL *et al.* « TSOtool: a program for verifying memory systems using the memory consistency model ». Dans : *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. Juin 2004, p. 114 –123.

[Hil98] M.D. HILL. « Multiprocessors should support simple memory consistency models ». Dans : *Computer* 31.8 (août 1998), p. 28 –34.

[HQR99] Thomas A. HENZINGER, Shaz QADEER et Sriram K. RAJAMANI. « Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems ». Dans : *Proceedings of the 11th International Conference on Computer Aided Verification*. CAV '99. London, UK, UK : Springer-Verlag, 1999, p. 301–315.

[IA-32] *Intel64 and IA-32 Architectures Software Developers's Manual Volume 3: System Programming Guide*. Intel Corporation. Mai 2012.

[Itanium] *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. Application note 251429-001. Intel Corporation, oct. 2002.

[ITR11] ITRS. « International Technology Roadmap for Semiconductor ». Dans : 2011. Chap. System Drivers.

[Lam79] L. LAMPORT. « How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs ». Dans : *Computers, IEEE Transactions on* C-28.9 (sept. 1979), p. 690 –691.

[LH94] D.H. LINDER et J.C. HARDEN. « Access graphs: a model for investigating memory consistency ». Dans : *Parallel and Distributed Systems, IEEE Transactions on* 5.1 (jan. 1994), p. 39 –52.

[LHH91] Anders LANDIN, Erik HAGERSTEN et Seif HARIDI. « Race-free interconnection networks and multiprocessor consistency ». Dans : *Proceedings of the 18th annual international symposium on Computer architecture*. ISCA '91. Toronto, Ontario, Canada : ACM, 1991, p. 106–115.

[Lud+02] J. M. LUDDEN *et al.* « Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems ». Dans : *IBM Journal of Research and Development* 46.1 (jan. 2002), p. 53 –76.

[MH05] Chaiyasit MANOVIT et Sudheendra HANGAL. « Efficient algorithms for verifying memory consistency ». Dans : *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA '05. Las Vegas, Nevada, USA : ACM, 2005, p. 245–252.

[MH06] C. MANOVIT et S. HANGAL. « Completely verifying memory consistency of test program executions ». Dans : *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. Fév. 2006, p. 166 –175.

[MS05] A. Meixner et D.J. Sorin. « Dynamic verification of sequential consistency ». Dans : *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*. Juin 2005, p. 482 –493.

[MS06] A. Meixner et D.J. Sorin. « Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures ». Dans : *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. Juin 2006, p. 73 –82.

[PD00] Fong Pong et M. Dubois. « Formal automatic verification of cache coherence in multiprocessors with relaxed memory models ». Dans : *Parallel and Distributed Systems, IEEE Transactions on* 11.9 (sept. 2000), p. 989 –1006.

[PD95] Fong Pong et M. Dubois. « A new approach for the verification of cache coherence protocols ». Dans : *Parallel and Distributed Systems, IEEE Transactions on* 6.8 (août 1995), p. 773 –787.

[Pla+98] Manoj Plakal *et al.* « Lamport clocks: verifying a directory cache-coherence protocol ». Dans : *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '98. Puerto Vallarta, Mexico : ACM, 1998, p. 67–76.

[Pou+09] Nicolas Pouillon *et al.* « A Generic Instruction Set Simulator API for Timed and Untimed Simulation and Debug of MP2-SoCs ». Dans : *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*. Juin 2009, p. 116–122.

[POWER] *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. version 3.0. IBM Corporation. 2005.

[PZT02] M. Prvulovic, Zheng Zhang et J. Torrellas. « ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors ». Dans : *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. 2002, p. 111 –122.

[Qad03] S. Qadeer. « Verifying sequential consistency on shared-memory multiprocessors by model checking ». Dans : *Parallel and Distributed Systems, IEEE Transactions on* 14.8 (août 2003), p. 730 –741.

[RHS12] E.A. Rambo, O.P. Henschel et L.C.V. dos Santos. « On ESL verification of memory consistency for system-on-chip multiprocessing ». Dans : *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. Mar. 2012, p. 9 –14.

[Roy+06] Amitabha Roy *et al.* « Fast and generalized polynomial time memory consistency verification ». Dans : *Proceedings of the 18th international conference on Computer Aided Verification*. CAV'06. Seattle, WA : Springer-Verlag, 2006, p. 503–516.

[SFC91] Pradeep S. Sindhu, Jean-Marc Frailong et Michel Cekleov. *Formal specification of memory models*. Rap. tech. Xerox Corporation, déc. 1991.

[Sha+08] Ofer Shacham *et al.* « Verification of chip multiprocessor memory systems using a relaxed scoreboard ». Dans : *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. Washington, DC, USA : IEEE Computer Society, 2008, p. 294–305.

[SN04] Robert C. Steinke et Gary J. Nutt. « A unified theory of shared memory consistency ». Dans : *J. ACM* 51.5 (sept. 2004), p. 800–849.

[SoCLib] *SoCLib*. `http://www.soclib.fr`.

[Sor+98] Daniel J. Sorin *et al.* *Lamport Clocks: Reasoning About Shared Memory Correctness*. Rap. tech. Computer Sciences Departement of University of Wisconsin, 1998.

[SPARC] *The SPARC Architecture Manual*. SPARC International. 1994.

[Splash2] *The Modified SPLASH-2*. `http://www.capsl.udel.edu/splash/`.

[SystemC] Accellera Systems Initiative. *SystemC*.

[Tar71] Robert Tarjan. « Depth-first search and linear graph algorithms ». Dans : *Switching and Automata Theory, 1971., 12th Annual Symposium on*. Oct. 1971, p. 114 –121.

[Tile64] *Tile Processor User Architecture Manual*. Tilera Corporation. Mai 2011.

[Uba+12] R. Ubal *et al.* « A Sequentially Consistent Multiprocessor Architecture for Out-of-Order Retirement of Instructions ». Dans : *Parallel and Distributed Systems, IEEE Transactions on* 23.8 (août 2012), p. 1361 –1368.

[VCI] *Protocole VCI/OCP*. `http://www.ocpip.org/home.php`.

[Z] *z/Architecture Principles of Operation*. International Business Machines Corporation. Mai 2004.

A Coût de la détection des cycles dans le graphe dynamique

Cette annexe donne une étude de la complexité de la méthode de détection des cycles dans le graphe dynamique. Cette méthode est décrite dans la section 3.1.4 du chapitre 3.

A.1 Méthodes de détection

A.1.1 Récapitulatif

Pour un graphe donné, la détection de la présence de cycles en son sein coute $O(T)$, où T représente sa taille en nombre d'éléments. Les arcs et les nœuds sont comptés comme étant un élément.

La méthode proposée consiste à exécuter la détection lorsque la taille du graphe atteint un seuil. Le seuil est ensuite multiplié par un facteur constant. Deux variantes sont proposées dans la chapitre 3 :

- la première se contente de doubler le seuil après chaque détection ;
- la seconde diminue de plus le seuil à chaque fois que des éléments sont supprimés.

A.1.2 Notations

Dans la suite, nous étudions le coût de cette détection lors des exécutions de celle-ci. Nous utilisons les notations suivantes.

Lors de la k-ième détection des cycles ($k \geq 1$) :

- t_k est la taille du graphe, c'est donc le coût de la k-ième détection ;
- a_k est le nombre d'éléments ajoutés depuis la détection précédente ;
- b_k est le nombre d'éléments supprimés depuis la détection précédente.

Pour la première détection, a_1 et b_1 représentent les nombres d'éléments ajoutés et supprimés depuis le début. t_k et a_k sont des entiers strictement positifs et b_k est un entier positif ou nul.

On note n_k le nombre d'éléments ajoutés depuis le début lors de la k-ième détection et $n_0 = 0$:

$$n_k = \sum_{i=1}^{k} a_k = n_{k-1} + a_k, \forall k \geq 1 \tag{A.1}$$

On note S le seuil de départ pour la détection ($S >= 1$) et F le facteur par lequel il est multiplié à chaque détection ($F > 1$).

On note $\delta \in \{0, 1\}$: δ vaut 0 si le seuil n'est pas décrémenté lors de la suppression d'un élément du graphe ; δ vaut 1 dans le cas contraire. Ce paramètre permet de modéliser les deux variantes étudiées.

On note c_k le coût total de la détection depuis le début lors de la k-ième détection.

A.2 Étude de la complexité

A.2.1 Taille du graphe

Lors des détections, deux équations régissent la taille du graphe t_k :

$$t_k = \begin{cases} a_1 - b_1 & \text{si } k = 1; \\ t_{k-1} + a_k - b_k & \forall k \geq 2. \end{cases} \tag{A.2}$$

$$t_k = \begin{cases} S - \delta.b_1 & \text{si } k = 1; \\ F.t_{k-1} - \delta.b_k & \forall k \geq 2. \end{cases} \tag{A.3}$$

L'équation A.2 est déduite de l'évolution de la taille du graphe entre deux détections : la taille du graphe est la taille précédente en prenant en compte les ajouts et les suppressions d'éléments. L'équation A.3 est déduite de l'évolution du seuil de détection. Celui-ci est en effet égal à la taille du graphe lors des détections et se calcule en fonction du seuil précédent et des suppressions.

En exprimant $F.t_{k-1}$ avec ces deux équations, on obtient pour tout $k \geq 2$:

$$F.t_{k-1} = F.(t_k - a_k + b_k)$$
$$F.t_{k-1} = t_k + \delta.b_k$$

Ainsi,

$$F.(t_k - a_k + b_k) = t_k + \delta.b_k \qquad \forall k \geq 2$$
$$(F-1).t_k = F.a_k - (F - \delta).b_k \qquad \forall k \geq 2$$

On obtient donc l'expression suivante de la taille du graphe en fonction du nombre d'ajouts et de suppressions :

$$t_k = \frac{F}{F-1}.a_k - \frac{(F-\delta)}{F-1}.b_k, \forall k \geq 2$$

Comme $b_k \geq 0$ et $(F - \delta)/(F - 1) > 0$, on a donc :

$$t_k \leq \frac{F}{F-1}.a_k, \forall k \geq 2 \tag{A.4}$$

A.2.2 Coût de la détection

Dans cette section, nous montrons par récurrence que $c_k \leq \frac{F}{F-1} n_k$ pour tout k supérieur à 1.

1ère détection Comme t_1 est la taille du graphe, on a

$$c_1 = t_1$$
$$c_1 = a_1 - b_1$$
$$c_1 \leq a_1 \text{ car } b_1 \geq 0$$
$$c_1 \leq \frac{F}{F-1} a_1 \text{ car } F > 1$$

***k*-ième détection** Soit $k >= 2$, supposons $c_{k-1} \leq \frac{F}{F-1} n_{k-1}$.

Comme t_k est la taille du graphe, le coût de la k-ième détection est t_k. Ainsi

$$c_k = c_{k-1} + t_k$$

Et donc, en utilisant l'équation A.4 et l'hypothèse de récurrence :

$$c_k \leq \frac{F}{F-1}(n_{k-1} + a_k)$$
$$c_k \leq \frac{F}{F-1} n_k$$

Par récurrence, nous avons donc

$$c_k \leq \frac{F}{F-1} n_k \ \forall k \geq 1 \tag{A.5}$$

A.2.3 Complexité

Ainsi le coût de la détection depuis le début lors de la k-ième détection est bien inférieur à $\frac{F}{F-1} n_k$, où n_k est le nombre d'éléments ajoutés dans le graphe depuis le début. En conclusion, pour les deux variantes, si n est le nombre total d'éléments ajoutés dans le graphe, la complexité totale de cette détection est donc

$$O(\frac{F}{F-1} \times n).$$

Si $F = 2$ (la valeur du paramètre cité dans le chapitre 3) la complexité est $O(2n)$.

B Fonctions de traitement des accès

Cette annexe est consacrée aux fonctions de traitement des accès qui ne sont pas présentes dans le chapitre 4. Elle contient les fonctions pour les segments de catégorie « I/O », « atomique » et « non atomique »

B.1 Segment I/O

FIGURE B.1 - Graphe partiel d'une case mémoire d'un segment « I/O »

Les ALGORITHMES B.1 et B.2 donnent les fonctions d'émission et de complétion pour les accès à un segment de catégorie « I/O ». L'ordre partiel construit est rappelé dans la FIGURE B.1. Comme les accès I/Os forment un ordre total, chaque accès est traité de manière identique. Chaque accès ne peut pas disparaître avant d'avoir servi à compléter l'accès suivant. Cela permet d'assurer que l'ordre est bien total. Néanmoins comme il n'y a rien à faire lors de la disparition de ces accès, nous ne donnons pas de fonction de traitement associée.

1: **fonction** ÉMISSIONIO(accès a)
2: **si** a est une lecture **alors**
3: crée le nœud R_a
4: AJOUTEMARQUEURS($R_a, \#m_a - 1$) ▹ −1 car il y a déjà 1 marqueur initial
5: **sinon**
6: crée le nœud W_a
7: AJOUTEMARQUEURS($W_a, \#m_a - 1$) ▹ −1 car il y a déjà 1 marqueur initial
8: **fin si**
9: **fin fonction**

ALGORITHME B.1 - Émission d'un accès à un segment « I/O »

1: **fonction** COMPLÉTIONIO(accès a_m, accès référent r_m)
Prérequis : a_m n'a pas déjà été complété
Prérequis : r_m est un accès I/O qui n'a pas encore *disparu*
Prérequis : r_m n'a pas encore été utilisé comme référent pour compléter un accès
2: ajoute $R_r/W_r \rightarrow R_a/W_a$
3: ENLÈVEMARQUEUR(R_a/W_a)
4: **fin fonction**

ALGORITHME B.2 - Complétion d'un accès « I/O »

B.2 Segment atomique

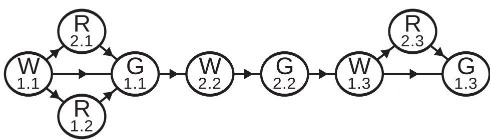


FIGURE B.2 - Graphe partiel d'une case mémoire d'un segment « atomique »

La FIGURE B.2 rappelle le graphe partiel généré pour une case mémoire d'un segment atomique. Les ALGORITHMES B.3 et B.4 donnent les fonctions de traitement de l'émission et de la complétion des lectures et écritures atomiques. L'ALGORITHME B.5 donne la fonction de disparition d'une écriture atomique. Le traitement des écritures est identique à celui pour la catégorie anticipée.

```
1:  fonction ÉMISSIONATOMIQUE(accès a)
2:      si a est une lecture alors
3:          crée le nœud R_a
4:          AJOUTEMARQUEURS(R_a, #m_a − 1)        ▷ −1 car il y a déjà 1 marqueur initial
5:      sinon
6:          crée le nœud W_a
7:          AJOUTEMARQUEURS(W_a, #m_a − 1)        ▷ −1 car il y a déjà 1 marqueur initial
8:          pour chaque case m accédée par a faire
9:              crée le nœud G_{a_m}              ▷ le nœud a un unique marqueur initial
10:             ajoute l'arc W_a → G_{a_m}
11:         fin pour
12:     fin si
13: fin fonction
```

ALGORITHME B.3 - Émission d'un accès à un segment « atomique »

```
1:  fonction COMPLÉTIONATOMIQUE(accès a_m, accès référent r_m)
Prérequis :  a_m n'a pas déjà été complété
Prérequis :  r_m est une écriture qui n'a pas encore disparue
2:      si a est une lecture alors
3:          ajoute W_r → R_a → G_{r_m}
4:          ENLÈVEMARQUEUR(R_a)
5:      sinon
Prérequis :  r_m n'a pas encore été utilisé comme référent pour compléter une écriture
6:          ajoute G_{r_m} → W_a
7:          ENLÈVEMARQUEUR(W_a)
8:      fin si
9:  fin fonction
```

ALGORITHME B.4 - Complétion d'un accès « atomique »

```
1:  fonction DISPARITIONATOMIQUE(écriture a_m)
2:      ENLÈVEMARQUEUR(G_{a_m})
3:  fin fonction
```

ALGORITHME B.5 - Disparition d'un accès « atomique »

B.3 Segment non atomique

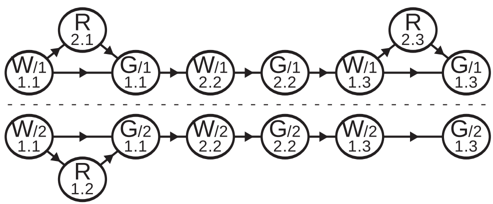


FIGURE B.3 - Graphe partiel d'une case mémoire d'un segment « non atomique »

La FIGURE B.3 rappelle le graphe partiel généré pour une case mémoire d'un segment atomique. Les ALGORITHMES B.6 et B.7 donnent les fonctions de traitement de l'émission et de la complétion des lectures et écritures atomiques. L'ALGORITHME B.8 donne la fonction de disparition d'une écriture atomique. Le traitement des lectures est identique à celui de la catégorie atomique. Pour les écritures, cela est très similaire à part que chaque écriture est séparée en autant d'écritures qu'il n'y a de processeurs. Un ordre partiel différent est créé pour chaque SI qui regroupe les sous-écritures correspondantes et les lectures de cette SI.

1: **fonction** ÉMISSIONNONATOMIQUE(accès a)
2: **si** a est une lecture **alors**
3: crée le nœud R_a
4: AJOUTEMARQUEURS(R_a, $\#m_a - 1$) ▹ −1 car il y a déjà 1 marqueur initial
5: **sinon**
6: **pour chaque** $i \in [1; P]$ **faire**
7: crée le nœud W_a^i
8: AJOUTEMARQUEURS(W_a^i, $\#m_a - 1$) ▹ −1 car il y a déjà 1 marqueur initial
9: **pour chaque** case m accédée par a **faire**
10: crée le nœud $G_{a_m}^i$ ▹ le nœud a un unique marqueur initial
11: ajoute l'arc $W_a^i \rightarrow G_{a_m}^i$
12: **fin pour**
13: **fin pour**
14: **fin si**
15: **fin fonction**

ALGORITHME B.6 - Émission d'un accès à un segment « non atomique »

1: **fonction** COMPLÉTIONNONATOMIQUE(accès a_m, accès référent r_m)
Prérequis : a_m n'a pas déjà été complété
Prérequis : r_m est une écriture qui n'a pas encore *disparue*
2: **si** a est une lecture **alors**
3: ajoute $W_r^p \rightarrow R_a \rightarrow G_{r_m}^p$ ▷ p désigne le processeur ayant émis a
4: ENLÈVEMARQUEUR(R_a)
5: **sinon**
Prérequis : r_m n'a pas encore été utilisé comme référent pour compléter une écriture
6: **pour chaque** $i \in [1; P]$ **faire**
7: ajoute $G_{r_m}^i \rightarrow W_a^i \rightarrow G_{a_m}^i$
8: ENLÈVEMARQUEUR(W_a^i)
9: **fin pour**
10: **fin si**
11: **fin fonction**

ALGORITHME B.7 - Complétion d'un accès « non atomique »

1: **fonction** DISPARITIONNONATOMIQUE(écriture a_m)
2: **pour chaque** $i \in [1; P]$ **faire**
3: ENLÈVEMARQUEUR($G_{a_m}^i$)
4: **fin pour**
5: **fin fonction**

ALGORITHME B.8 - Disparition d'un accès « non atomique »

C | Description des événements tracés

Les événements utilisés dans notre implantation sont décrits dans cette annexe. Pour chaque événement, nous décrivons son contenu ainsi que les relations qu'il est susceptible d'avoir. Certaines parties du contenu sont marquées comme « optionnelles » : cela signifie qu'elles ne sont pas nécessaires pour la VCM mais qu'elles peuvent être utiles pour d'autres analyses (par exemple sur le logiciel). Dans notre implantation, il est possible d'activer ou non la trace de ces informations supplémentaires.

Dans un premier temps, nous décrivons le contenu commun à tous les événements. Nous nous attardons ensuite sur chacun d'entre eux. La liste des événements tracés par chaque composant est décrite dans la section 6.1.1.1 du chapitre 6.

C.1 Contenu commun

Comme nous l'avons dit dans le chapitre 5, chaque événement contient au moins :

- le nombre de ses conséquences ; et
- la liste de ses causes.

Les causes et les conséquences ont été définies dans le chapitre 5 : ce sont des événements.

À cela nous pouvons ajouter de 0 à 2 dates. Ces dates sont optionnelles et permettent de renseigner une date lorsque l'événement est ponctuel (un acquittement par exemple). Si l'événement n'est pas ponctuel (un accès par exemple), deux dates indiquent le début et la fin de celui-ci.

Tous les événements décrits dans la suite sont munis de ce contenu commun.

C.2 *spread*

L'événement *spread* est un événement spécial. Il n'a pas de signification particulière pour le comportement d'un composant. C'est un événement intermédiaire qui permet d'« augmenter » le nombre de conséquences de l'événement dont il est la conséquence. Comme nous l'avons souligné dans la section 5.2.3.3, cet événement est nécessaire pour gérer les cas où un événement à un nombre de conséquences inconnu lors de sa génération.

Cet événement n'a aucun contenu spécifique.

C.3 exception

L'exception sert uniquement pour l'analyse du logiciel. Son contenu est le type de l'exception (interruption, *syscall*, *bus error*, etc.). Quand cela a du sens, elle peut avoir une cause : l'événement qui a provoqué l'exception.

C.4 instruction

L'instruction contient les informations suivantes :
- l'adresse de l'instruction ;
- (optionnel) le résultat (succès ou échec) ;
- (optionnel) la liste des registres lus ;
- (optionnel) la liste des registres écrits ; et
- (optionnel) les données écrites dans les registres.

Les listes des registres sont nécessaires pour la vérification d'un MCM relâché tel que RMO.

Une instruction peut éventuellement avoir une cause : l'exception causant l'exécution de l'instruction. En dehors de ce cas, les instructions sont émises par les processeurs dans l'ordre d'exécution rendant inutile toute autre relation.

C.5 accès

L'accès contient les informations suivantes :
- le type (lecture, écriture, LdL, StC, invalidation, barrière) ;
- l'adresse ;
- le résultat (échec ou succès) ;
- (optionnel) les données ;

Les invalidations et les barrières mémoire sont gérés comme des accès. Pour les barrières mémoire, cela a été nécessaire car les barrières mémoire ont un impact au niveau des caches et des tampons d'écriture. Lorsque l'accès est global (par exemple une barrière mémoire), l'adresse n'a pas d'importance.

Certains accès peuvent échouer (par exemple les StCs), le résultat indique si c'est le cas. Nous utilisons aussi cela pour la lecture qui sert aux caches pour indiquer les chargements de ligne : à cause de la durée d'un chargement, il est possible que la ligne soit invalidée avant qu'il ne soit terminé. Dans ce cas, nous disons que le chargement a échoué : l'accès a eu lieu (et a pu être utilisé pour répondre à la lecture du processeur qui a causé le chargement) mais n'a pas provoqué de mise à jour du cache.

Un accès a généralement une cause qui peut être une instruction (par exemple pour une écriture d'une instruction de type *load*) ou un autre accès (par exemple pour un chargement de ligne causé par un accès de type lecture). Certains accès n'ont pas forcément de cause (par exemple l'invalidation d'une ligne d'un cache) : nous ne traçons pas les détails du protocole de cohérence car ils ne sont pas nécessaires à la VCM, seul le contenu des caches nous importe.

C.6 acquittement

Un acquittement ne contient aucune information spécifique. La présence de l'événement suffit.

Un acquittement a généralement une cause : l'accès acquitté. Quand l'acquittement se fait par rapport au composant générant l'acquittement, il n'a qu'une seule cause : l'accès acquitté. Dans ce cas la signification de l'acquittement dépend du composant (par exemple, un cache acquittant une lecture signifie que la lecture lit la copie contenue dans le cache).

Dans le cas où une lecture lit directement la valeur contenue dans une requête d'écriture, nous utilisons un acquittement avec deux causes. La première est la lecture, la deuxième est l'écriture dont la valeur est utilisée. Ce procédé est utilisé pour modéliser les tampons d'écriture.

D | Résultats des simulations

Cette annexe présente le comportement des simulations effectuées pour les huit plates-formes (*MJPEG_1*, *MJPEG_2*, *MJPEG_3*, *OCEAN_4*, *OCEAN_8*, *OCEAN_16* et OCEAN_32). Ces plates-formes ont été décrites dans le chapitre 6 qui détaille principalement les résultats d'*OCEAN_32*. Les résultats des autres expérimentations sont similaires et sont donnés ici.

D.1 Expérimentations MJPEG_1

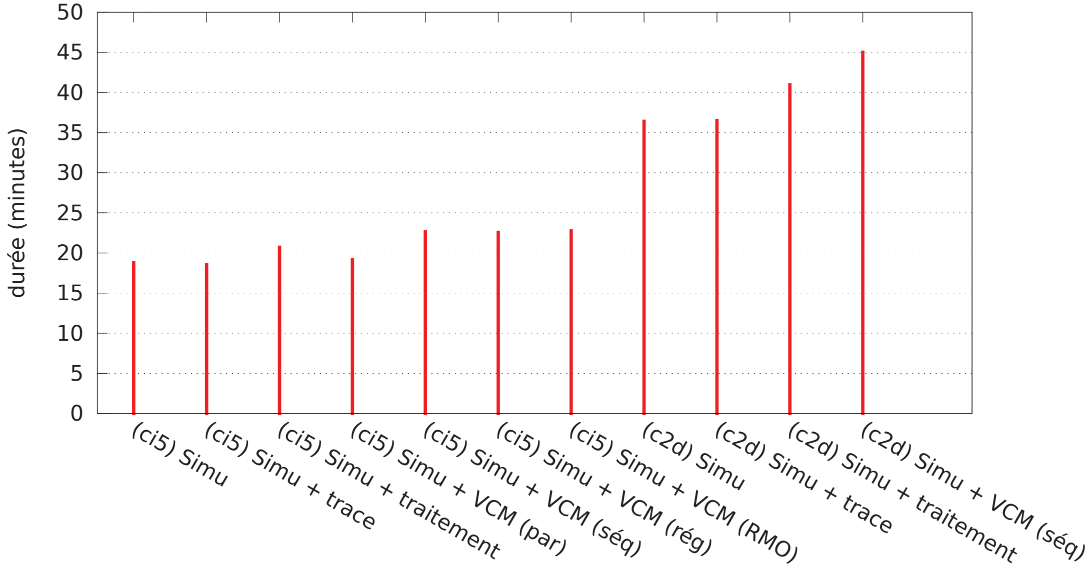


Figure D.1 - Durée des différentes simulations pour MJPEG_1 (1 milliard de cycles)

La Figure D.1 donne les durées moyennes des simulations (pour 1 milliard de cycles) effectuées pour *MJPEG_1*. Celles correspondant aux simulations sur un processeur *Core i5 2500K* (fréquence de 3,3 GHz, cache de 6 Mio) sont préfixées de « (ci5) ». Celles correspondant aux simulations sur un processeur *Core 2 Duo E4500* (fréquence de 2,2 GHz, cache de 2 Mio) sont préfixées de « (c2d) ». Plusieurs configurations de l'analyse de la simulation sont données dans la figure. Celles-ci sont décrites ci-dessous. Hormis quand cela est précisé, un seul *threqd* est utilisé pour la simulation et l'ensemble des traitements. De même, sauf si cela est précisé le MCM vérifié est TSO.

Simu Simulation seule (sans trace ni analyse).

Simu + trace Simulation avec génération des événements mais sans analyse.

Simu + traitement Simulation avec génération des événements et traitement de ceux-ci dans la bibliothèque d'analyse. L'algorithme de VCM n'est pas exécuté.

Simu + VMC (par) Simulation avec la VCM quand le traitement des événements et la VCM sont faits dans un second *thread*.

Simu + VMC Simulation avec la VCM quand tout est est fait dans le même *thread* que la simulation. La détection des cycles est faite avec la méthode minimale.

Simu + VMC (rég) Simulation avec la VCM quand la détection des cycles est faite avec la méthode régulière.

Simu + VMC (RMO) Simulation avec la VCM quand le MCM vérifié est de type RMO.

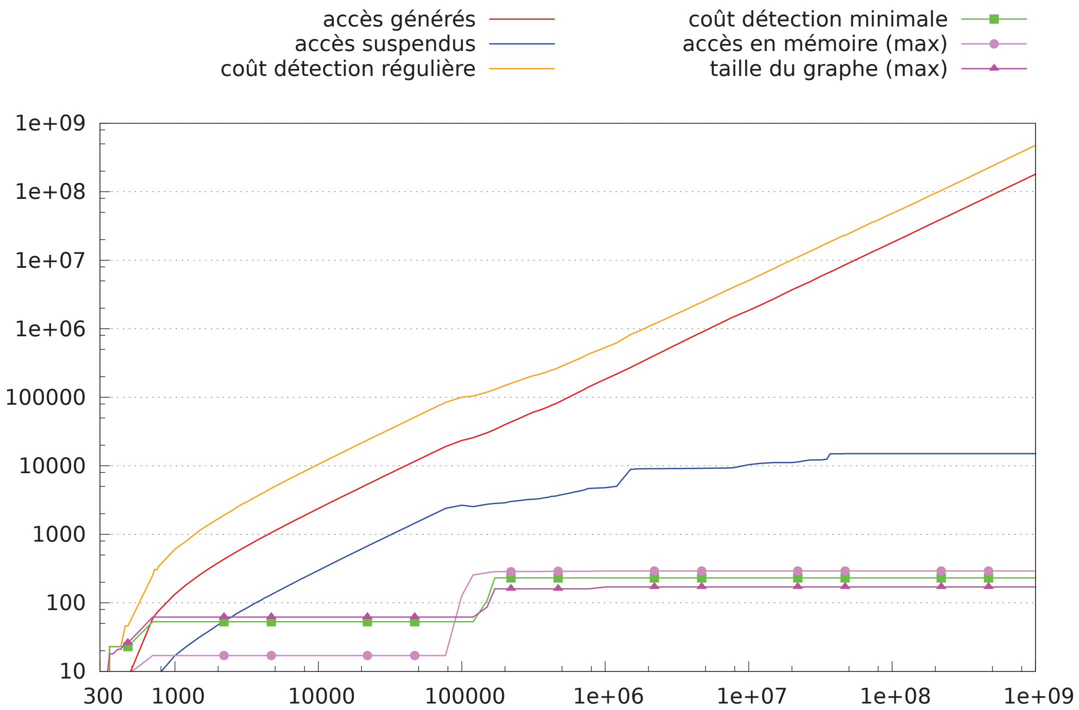


FIGURE D.2 - Évolution de l'algorithme de VCM pour MJPEG_1

La FIGURE D.2 montre l'évolution de certaines métriques de la VCM. On notera que les deux échelles sont logarithmiques, ce qui permet de voir l'évolution au début. Les métriques indiquées sont :

- le nombre total d'accès générés depuis le début,
- le nombre maximal d'accès conservé en mémoire dans l'algorithme,
- la taille maximale atteinte du graphe (nœuds + arcs),
- le nombre d'accès suspendus (supprimés temporairement),
- le coût cumulé (depuis le début) de la détection des cycles pour les deux méthodes.

Les figures correspondant aux autres expérimentations sont données dans les sections suivantes. Pour la durée des simulations, certaines configurations n'ont pas été simulées (en particulier pour les simulations OCEAN) et ne sont donc pas incluses dans les figures.

D.2 Expérimentation MJPEG_2

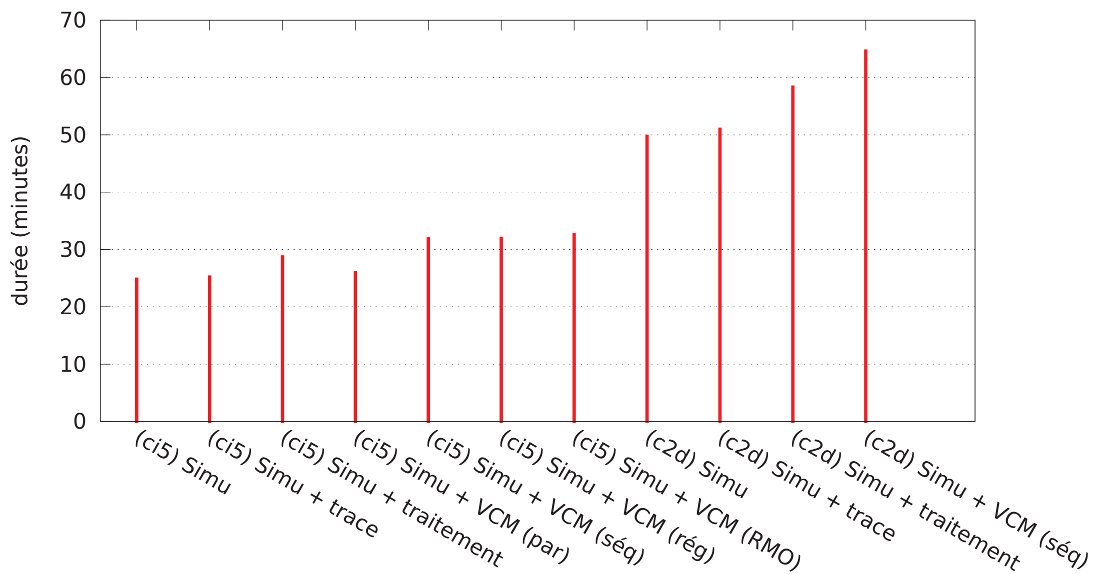


FIGURE D.3 - Durée des différentes simulations pour MJPEG_2 (1 milliard de cycles)

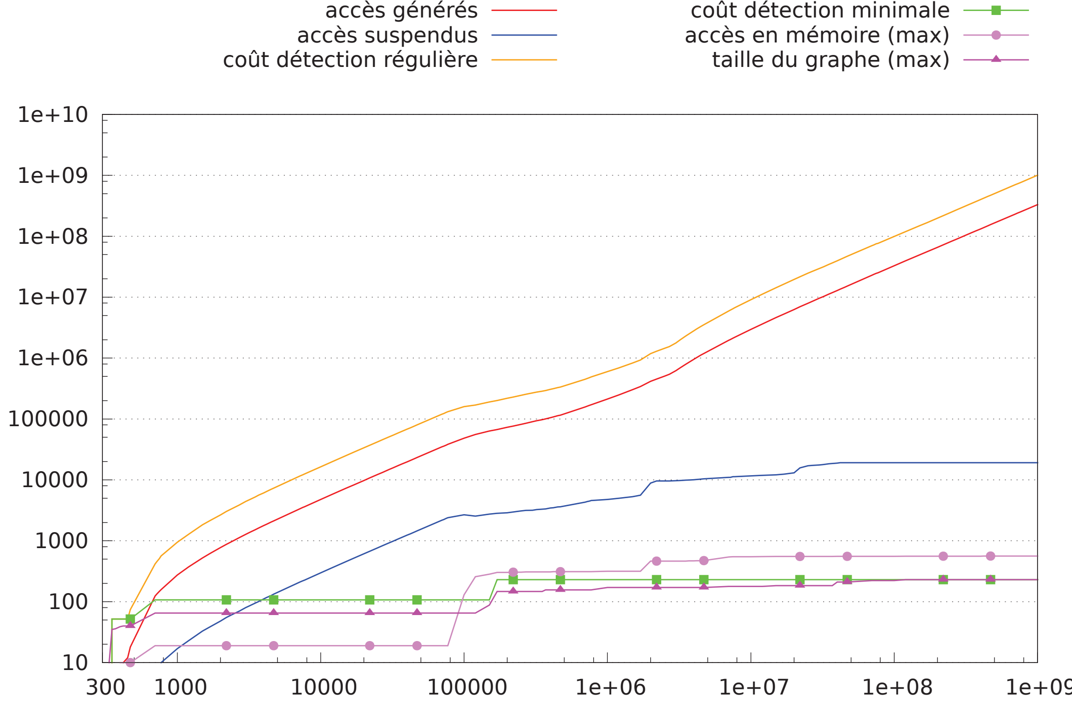


FIGURE D.4 - Évolution de l'algorithme de VCM pour MJPEG_2

D.3 Expérimentation MJPEG_3

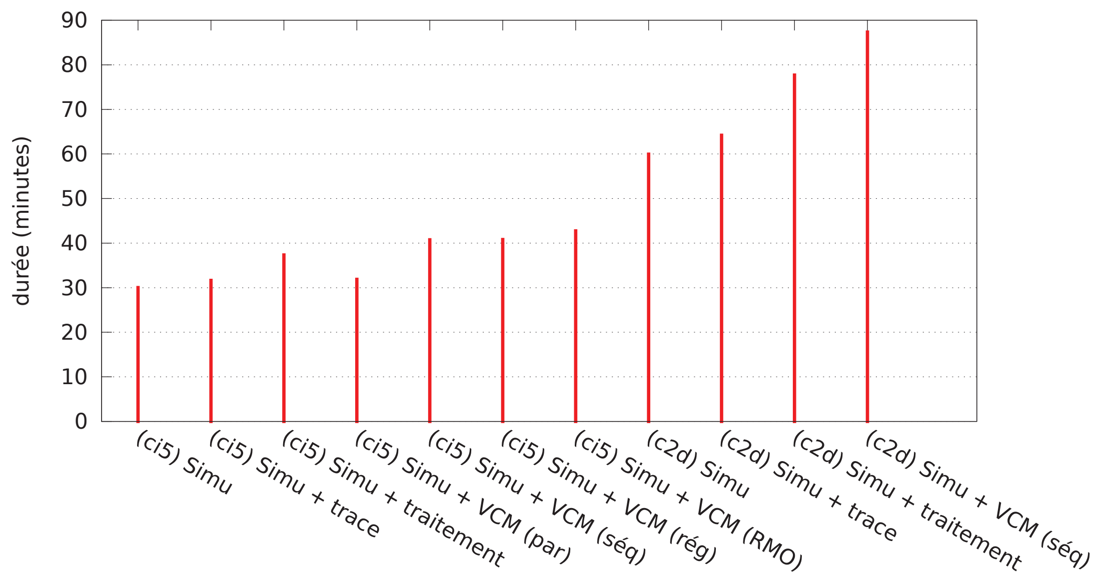


FIGURE D.5 - Durée des différentes simulations pour MJPEG_3 (1 milliard de cycles)

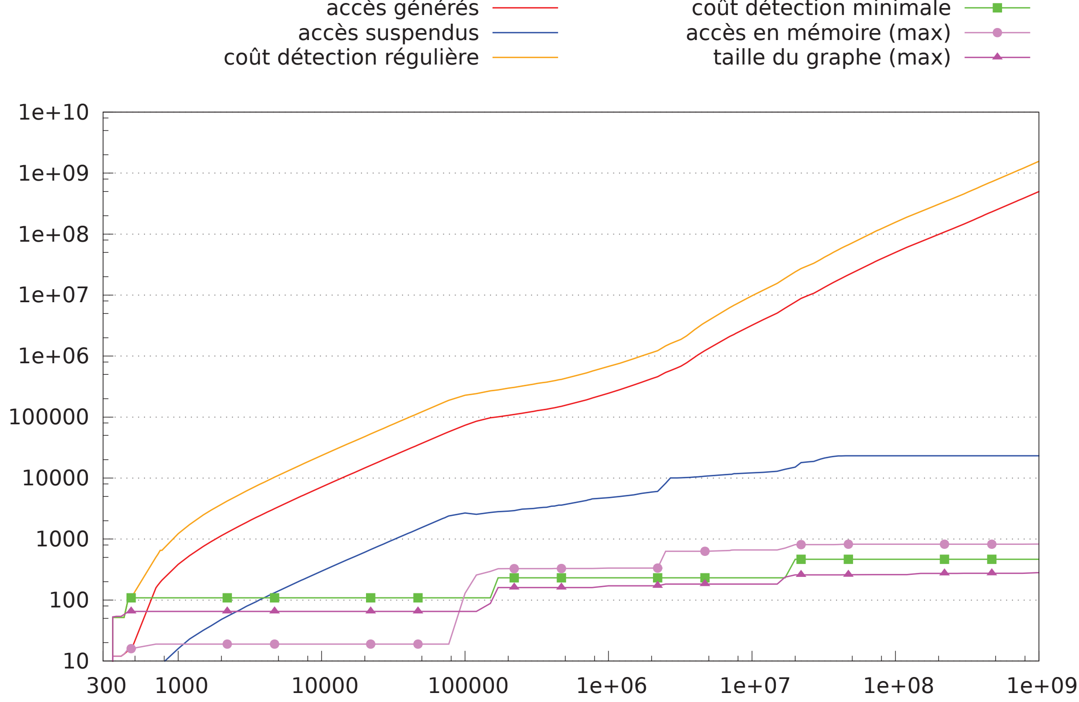


FIGURE D.6 - Évolution de l'algorithme de VCM pour MJPEG_3

D.4 Expérimentation MJPEG_4

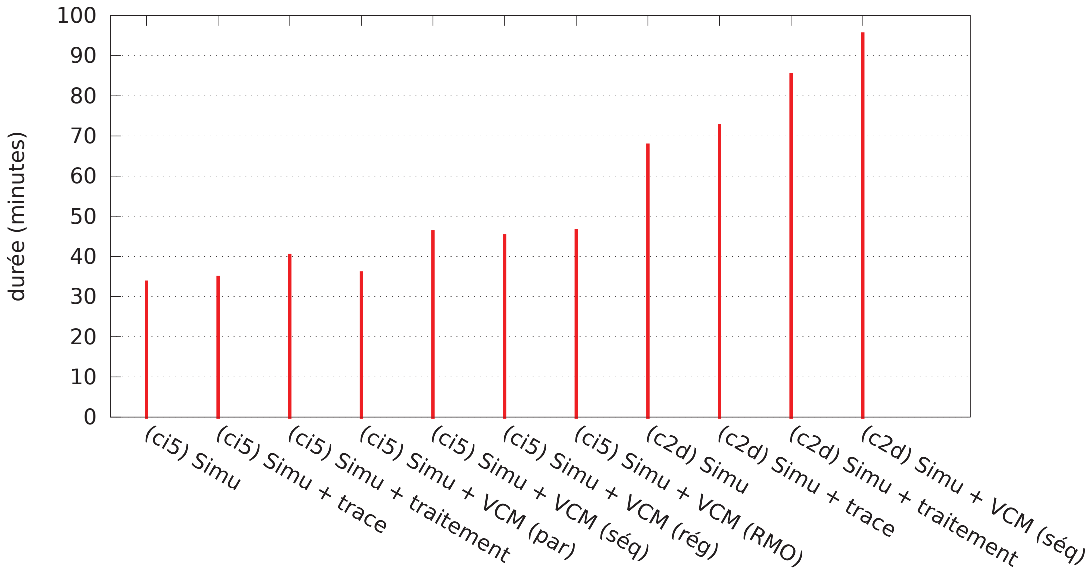


FIGURE D.7 - Durée des différentes simulations pour MJPEG_4 (1 milliard de cycles)

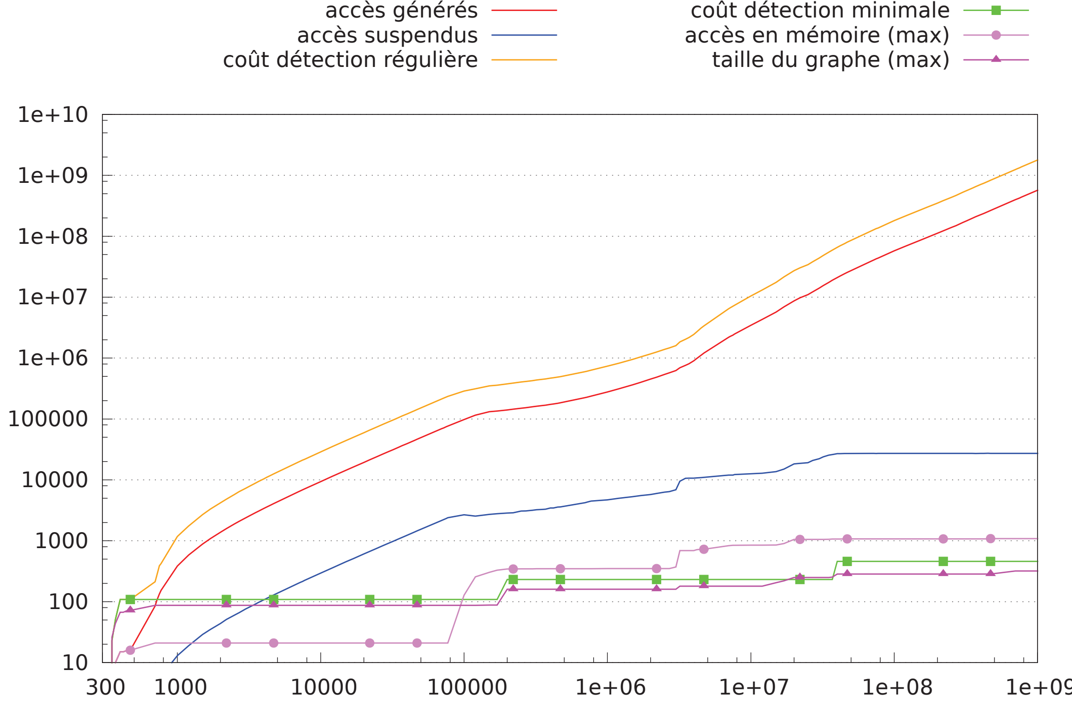


FIGURE D.8 - Évolution de l'algorithme de VCM pour MJPEG_4

D.5 Expérimentation OCEAN_4

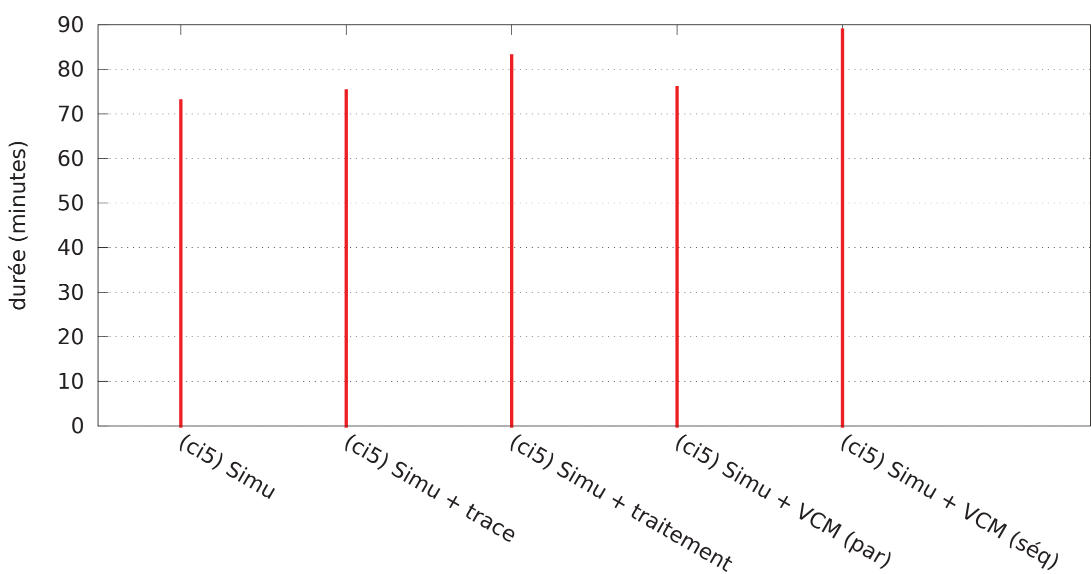


FIGURE D.9 - Durée des différentes simulations pour OCEAN_4 (1 milliard de cycles)

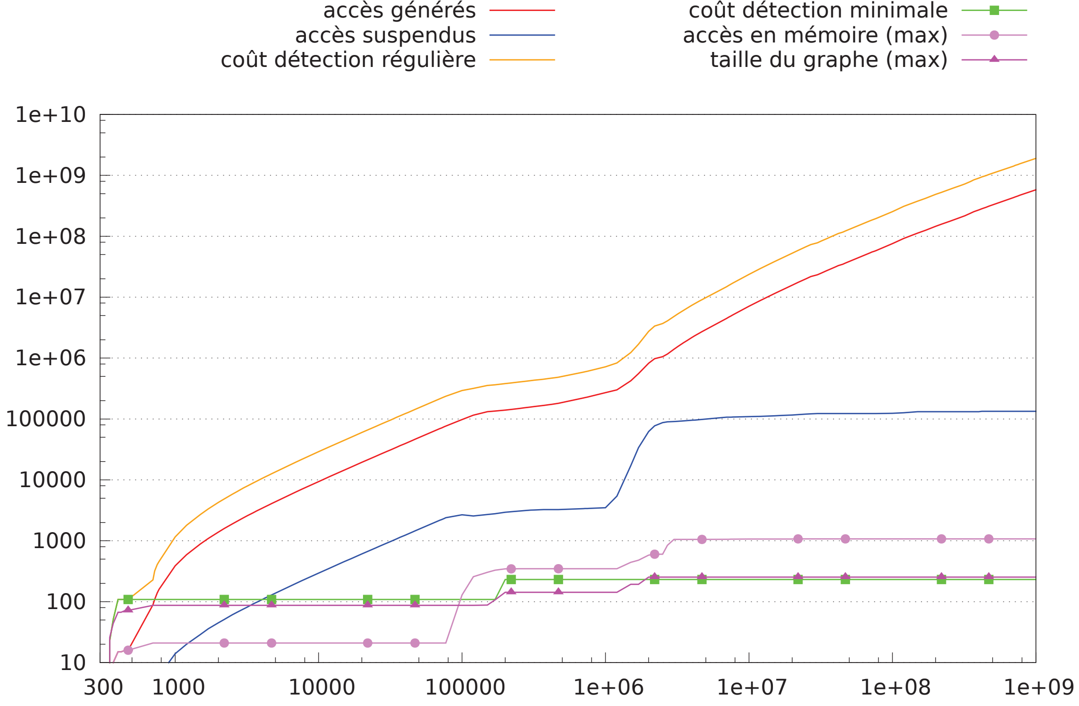


FIGURE D.10 - Évolution de l'algorithme de VCM pour OCEAN_4

D.6 Expérimentation OCEAN_8

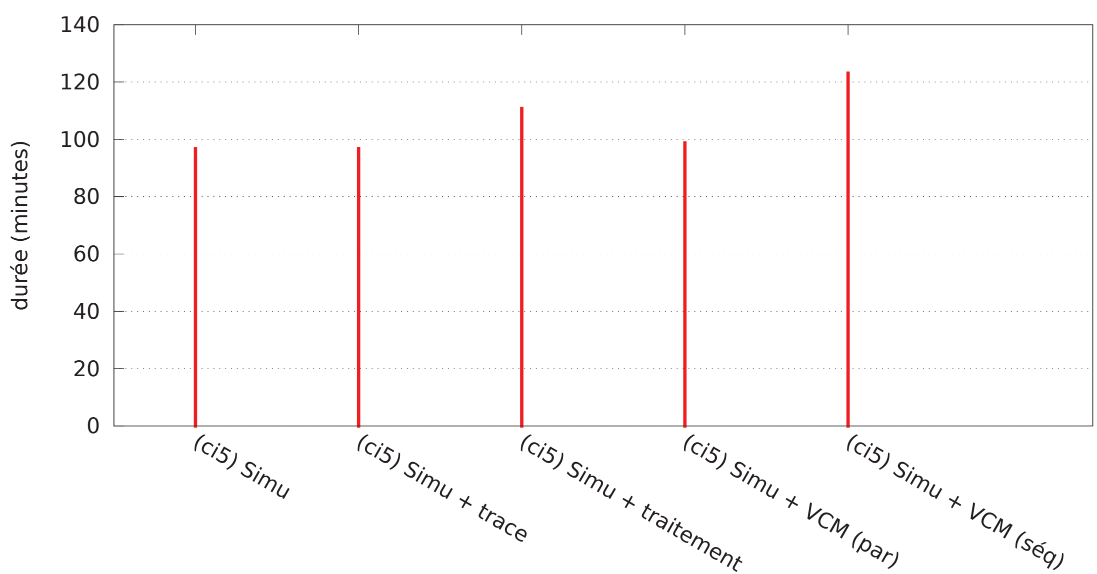


FIGURE D.11 - Durée des différentes simulations pour OCEAN_8 (1 milliard de cycles)

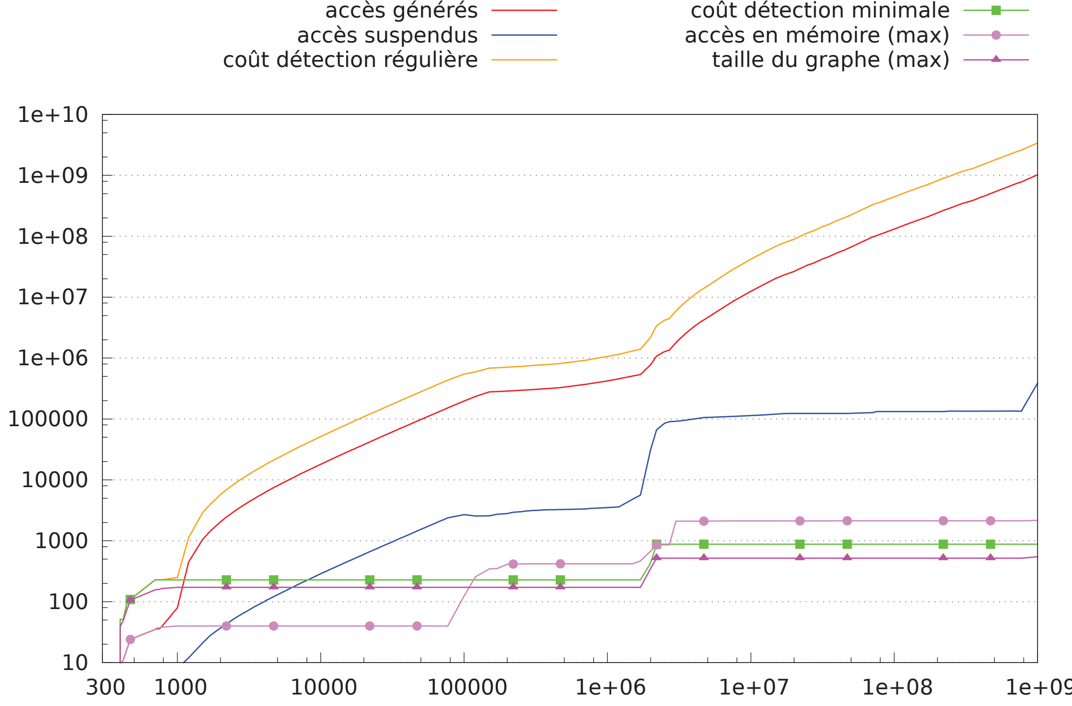


FIGURE D.12 - Évolution de l'algorithme de VCM pour OCEAN_8

D.7 Expérimentation OCEAN_16

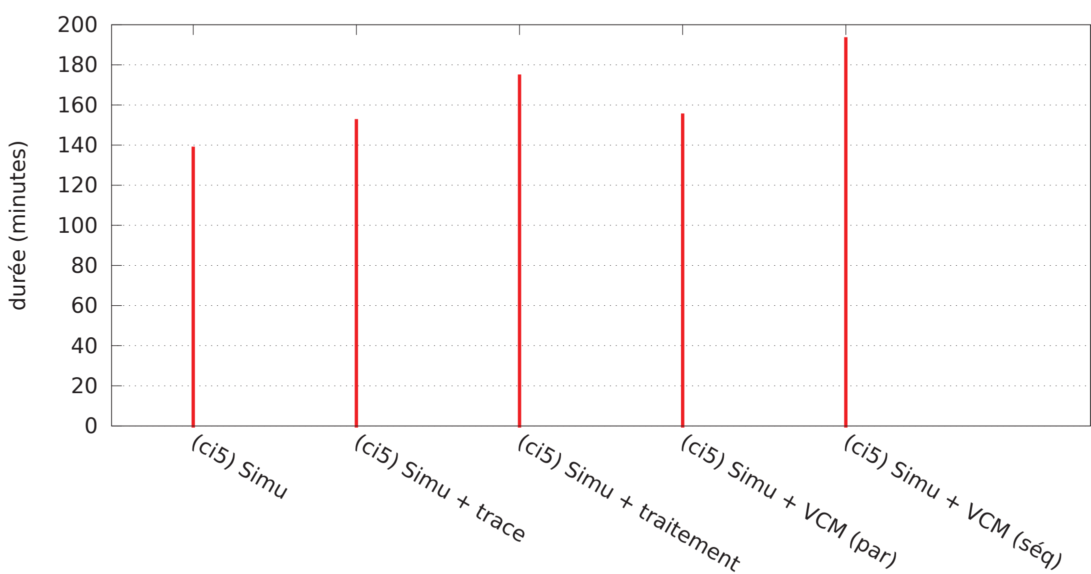


FIGURE D.13 - Durée des différentes simulations pour OCEAN_16 (1 milliard de cycles)

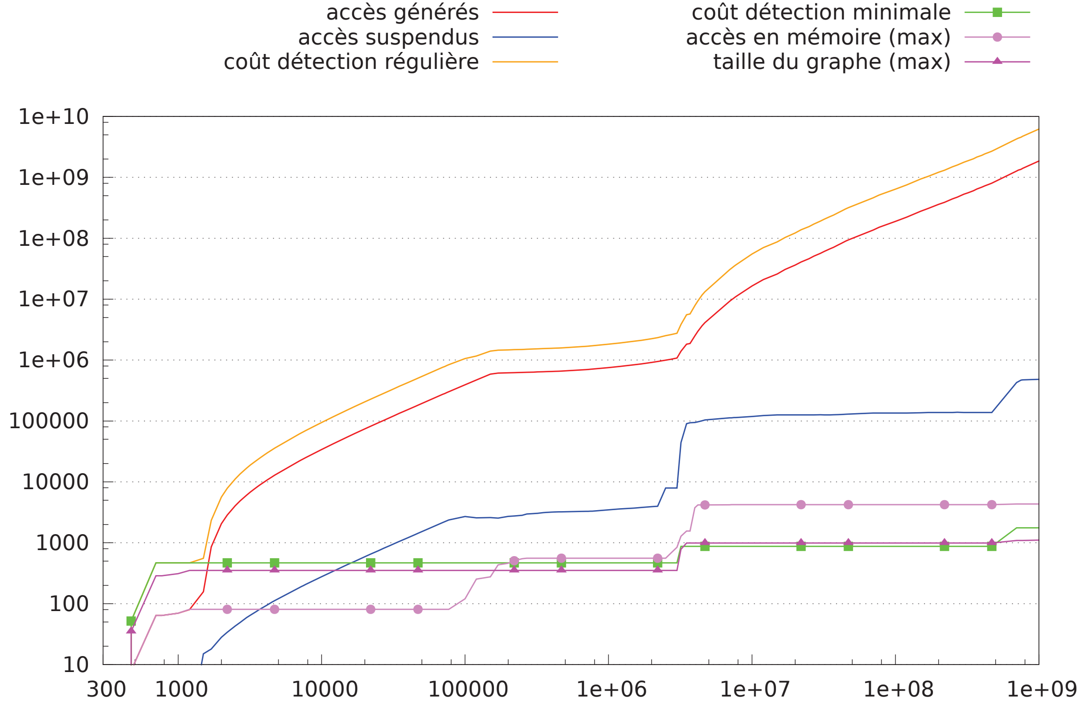


FIGURE D.14 - Évolution de l'algorithme de VCM pour OCEAN_16

D.8 Expérimentation OCEAN_32

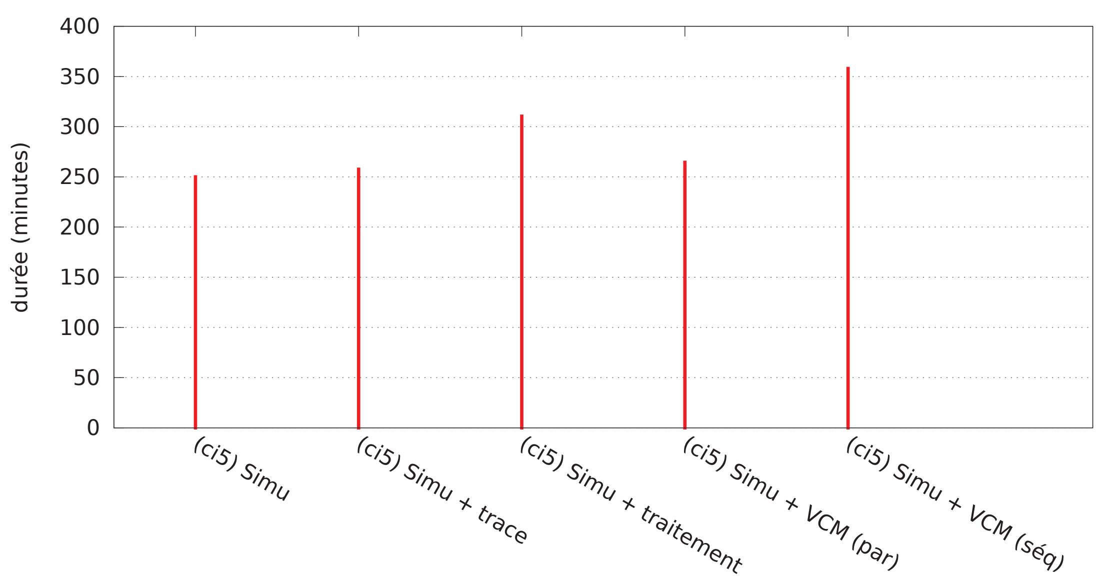


FIGURE D.15 - Durée des différentes simulations pour OCEAN_32 (1 milliard de cycles)

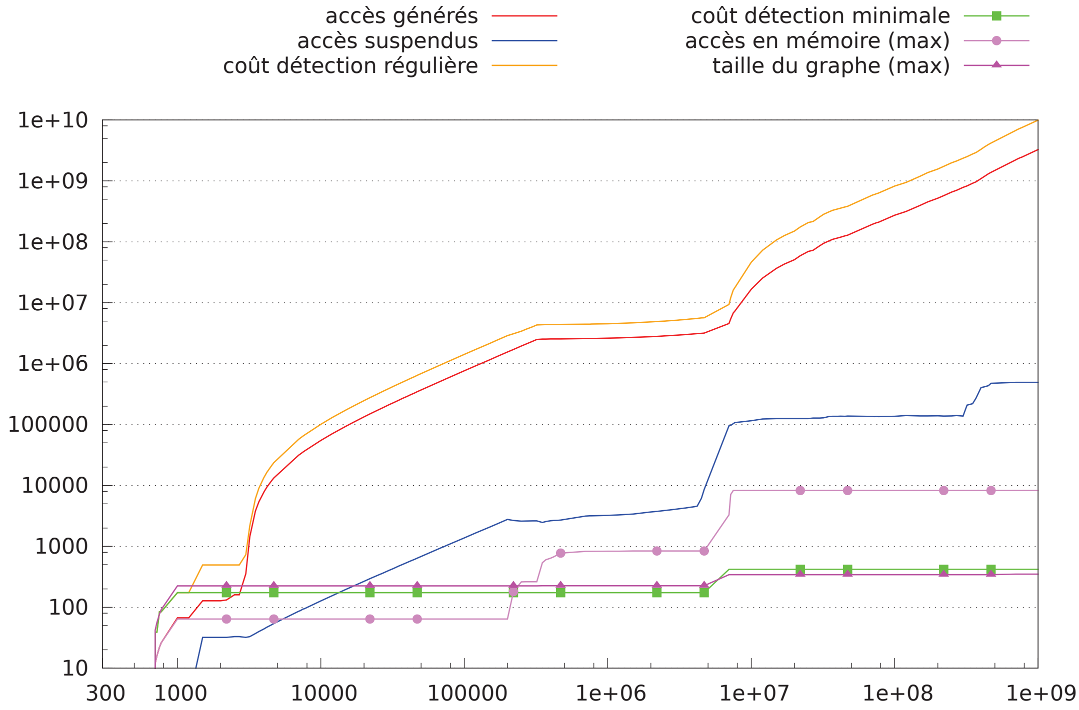


FIGURE D.16 - Évolution de l'algorithme de VCM pour OCEAN_32