Accélération de la simulation par échantillonnage dans les architectures multiprocesseurs embarquées

Melhem Tawk

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre: 09/10

Université de Valenciennes et du Hainaut-Cambrésis

THÈSE

présentée et soutenue publiquement le 23 juin 2009

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Melhem TAWK

Accélération de la Simulation par Échantillonnage dans les Architectures Multiprocesseurs Embarquées

Composition du jury

Rapporteurs :	Bernard GOOSSENS	Professeur	DALI, Université de Perpignan
	El Mostapha ABOULHAMID	Professeur	DIRO, Université de Montréal Canada
Examinateurs :	Thierry COLLETTE	Directeur de Recherche	CEA-LIST Paris
	Omar HAMMAMI	Enseignant-chercheur	ENSTA Paris
	Kaled Z. IBRAHIM	Chercheur	Lawrence Berkeley Lab USA
Directeur :	Smail NIAR	Professeur	LAMIH, Université de Valenciennes

UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT-CAMBRÉSIS
Laboratoire d'Automatique de Mécanique et d'Informatique Industrielles et Humaines — UMR CNRS 8530
59313 VALENCIENNES CEDEX 9
Tél. : +33 (0)3 27 51 13 50 – Télécopie : +33 (0)3 27 51 13 16

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Glossaire

ACMI : Average Cache Misses per Interval ;	Nombre moyen de défauts de cache par intervalle de l'application.
AMRI : Average Memory References per Interval ;	Nombre moyen de références mémoires par intervalle de l'application.
AS : Adaptive Sampling ;	Méthode d'échantillonnage proposée dans cette thèse et basée sur l'utilisation de CS et de barrières de simulation (voir chapitre 3).
BB : Basic Block ;	Blocs de base de l'application.
BBM : Basic Block Matrix ;	Matrice comprenant plusieurs BBVs représentant des intervalles des tailles différentes.
BBV :Basic Block Vector ;	Vecteur représentant un intervalle de l'application. Il est généré dans l'outil de profilage. Ici nous avons utilisé SimPoint.
CS : Cluster of Strings ;	Recouvrement parfait effectué par la méthode AS pour les phases parallèle. Il comprend une séquence de phases par processeur.
CST : Cluster of Strings Table	Table contenant les CS générés durant la simulation.
DSE :Design Space Exploration ;	Exploration de l'espace des configurations architecturales possibles du MPSoC.
EPC :	Énergie consommée par cycle d'horloge.
Hétérogènes :	Applications s'exécutant de façon concurrentes et ayant des besoins en ressources architecturales différents (Mémoire, Calcul, etc.).
Homogènes :	Applications s'exécutant de façon concurrentes et ayant des besoins identiques en ressources architecturales (Mémoire, Calcul, etc.).
Intervalle :	Section de l'exécution d'un programme. La taille d'un intervalle correspond au nombre d'instructions exécutées durant la section. Un intervalle est composé d'un ou plusieurs blocs de base.
IPC :	Nombre moyen d'instructions exécutées par cycle horloge.
Irrégulière :	Application dans laquelle les besoins en ressources dans les différentes phases sont différents. Dans une telle application la variation de la performance (exemple l'IPC) est importante d'une phase à l'autre.
MGS : Multi-Granularity Sampling ;	Méthode d'échantillonnage proposée dans cette thèse, basée sur l'utilisation de MPC (voir chapitre 5).
MPARM :	Platforme de simulation des MPSoC utilisée dans le cadre de cette thèse. Celle ci est basée sur des cores ARM7.

Mphase :	Ensemble des BBMs similaires ou ayant le même comportement durant l'exécution.
MPC : Multi-Phase Cluster ;	Recouvrement parfait effectué par la méthode MGS pour les phases parallèle. Le MPC comprend une phase par processeur.
MPCT : Multi-Phase Cluster Table	Table contenant les MPC générés durant la simulation.
Phase :	Ensemble des intervalles similaires ou ayant le même comportement durant l'exécution.
TWSB : Threshold Waiting at Simulation Barrier ;	Seuil pour le nombre des cycles d'attente dans les barrières de simulation.
Recouvrement de phases :	C'est une sorte de combinaison des phases parallèle. Le recouvrement de phases correspond à une génération du CS dans le cas de la méthode AS ou à une génération du MPC dans le cas de la méthode MGS.
Régulière :	Application dans laquelle les besoins en ressources dans les différentes phases sont identiques. Dans une telle application la variation de la performance (exemple l'IPC) est faible.
SimPoint :	Méthode d'accélération par analyse de phases de l'application. Elle utilise un profilage de l'application et la méthode de k-means.
SMART :	Méthode d'accélération qui consiste à choisir périodiquement les échantillons de l'application.

Chapitre 1

Introduction

1.1 Problématique

Les systèmes embarqués sur puce, ou SoC pour "System on Chip", sont de plus en plus présents dans notre environnement quotidien et professionnel. Ces systèmes s'engouffrent chaque jour un peu plus dans les domaines comme les transports (automobile, aéronautique, etc.), les télécommunications, l'électroménager (télévision, four à micro-ondes, etc.) et les équipements médicaux. Pour l'année 2004 uniquement, environ 260 millions de processeurs ont été vendus pour équiper des PCs de bureau tandis que plus de 14 milliards de processeurs (sous diverses formes : micro-processeur, micro-contrôleur, DSP) ont été vendus pour les systèmes embarqués. De plus, une augmentation annuelle de 16% du chiffre d'affaires est prévue pour le marché des systèmes embarqués pour l'année 2009 (Future of Embedded Systems Technology from BCC Research Group) [28].

Aujourd'hui, les SoC offrent des fonctionnalités de plus en plus complexes, telles que la vidéo, la connexion internet, le commerce électronique, satisfaisant ainsi les besoins des utilisateurs. Durant ces dernières années, les SoC ont connu une croissance accélérée de la complexité architecturale. Cette complexité est notamment observable dans l'intégration sur une même puce, d'une architecture constituée de multiples ressources de calcul hétérogènes (mémoires et processeurs) et un réseau d'interaction (ou network-on-chip - NoC) adapté pour la communication entre les différentes ressources. L'avènement de ces systèmes permet aujourd'hui d'envisager sur une puce des applications innovantes, très gourmandes en ressources de calcul, tel que le traitement multimédia.

Malheureusement, cette forte intégration de ressources risque de violer les contraintes imposées par ces systèmes. En effet, les SoC doivent être conçus de manière efficace afin d'assurer une certaine fiabilité. A cela s'ajoutent les contraintes du marché, qui poussent à réduire le temps de conception et de mise sur le marché "time to market". D'un point de vue commercial, ces deux facteurs favorisent la réussite du produit final. En outre, ces systèmes doivent être également développés et vérifiés de manière sûre dans le but de minimiser le coût de fabrication, le temps d'exécution, la consommation d'énergie et la taille du système. Afin de réaliser ces objectifs et profiter de l'augmentation du nombre de transistors par puce, des chercheurs multiplient les efforts pour faciliter le développement des SoC. La solution "Multiprocesseur sur puce" ou MPSoC pour construire des systèmes embarqués performants représente une alternative intéressante. Ces derniers sont devenus une solution incontournable pour l'exécution des applications nécessitant des calculs importants comme les applications multimédia. Les MPSoC sont généralement des systèmes hétérogènes puisqu'ils peuvent contenir, en outre, de la mémoire (Cache, SRAM, Scrathpad, etc.), différents processeurs (RISC, VLIW, etc.), des réseaux d'interconnexion ou NoC pour "Network on Chip", des périphériques d'entrée et de sortie et éventuellement de la logique reconfigurable (FPGA).

L'un des défis majeurs dans la conception des MPSoC, est la réduction de la phase d'évaluation des différentes alternatives de conception. En effet, ces alternatives sont représentées par un espace de solutions très vaste. Comme le montre la figure 1.1, la conception d'un SOC passe par l'utilisation de plusieurs niveaux de modélisation et d'évaluation de performances. A chaque niveau, un ensemble d'alternatives architecturales est évalué afin de garder la ou les alternatives les plus performantes. Afin de réduire le temps de conception, plus précis est le niveau, plus restreint sera le nombre de solutions d'alternatives architecturales évalués.

Dans cette thèse, nous nous intéressons au niveau de modélisation cycle précis / bit

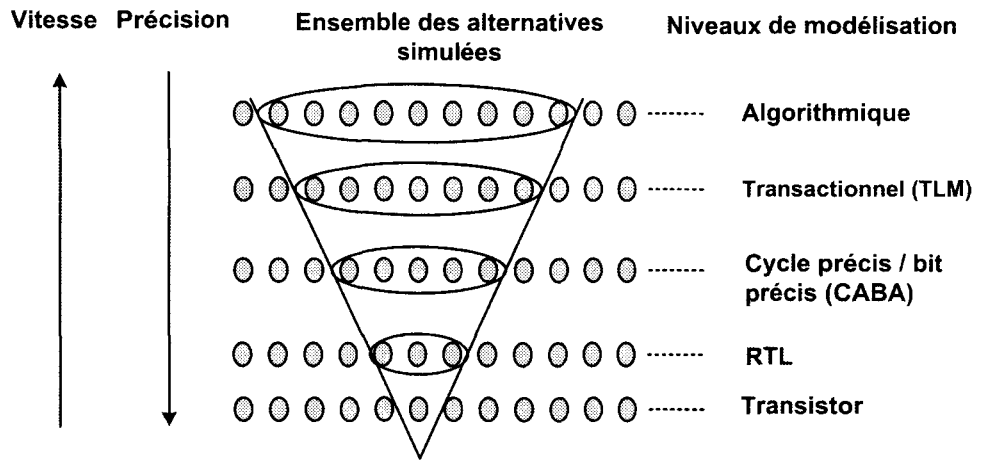


FIG. 1.1 – Différents niveaux de modélisation.

précis (ou CABA pour Cycle Accurate / Bit Accurate). Dans ce niveau, la description de l'architecture est réalisée sous forme de blocs architecturaux. Les détails tels que la structure du pipeline des processeurs, ou ceux de la structure micro-architecturale des caches sont exposés. Il s'agit par conséquent d'évaluer plusieurs centaines de solutions micro-architecturales pour déterminer le temps de simulation et la consommation de puissance. Habituellement, à ce niveau l'exploration, de l'espace est réalisée en modifiant le nombre et la structure du processeur (le nombre d'étages du pipeline, la technique de prédiction de branchements, etc.), la structure du cache (taille des blocs, degrés d'associativité), débit et structure du réseau, etc.

C'est dans ce contexte que se situe cette thèse. Plus précisément, le but de ce travail est de trouver une méthodologie pour accélérer l'évaluation des différentes alternatives.

Diminution du temps de simulation

Pour concevoir des MPSoC plus rapidement et plus efficacement, nous devons offrir aux concepteurs de nouveaux outils. Avec l'augmentation de la complexité des systèmes MPSoC la simulation au niveau CABA devient problématique pour évaluer les MPSoC. Le temps de simulation augmente avec l'augmentation de la complexité. En utilisant notre plateforme de simulation MPSoC qui sera détaillée par la suite, la vitesse de simulation est divisée par un facteur de 6 quand le nombre de processeurs augmente de 1 à 12. La figure 1.2 montre la vitesse de simulation en nombre d'instructions simulées par seconde et la fréquence correspondante en nombre de cycles simulés par seconde pour différentes configurations architecturales. La baisse en vitesse de simulation est due à l'overhead dans l'ordonnancement des tâches qui augmente quand plus de processeurs sont ajoutés à l'architecture MPSoC.

Pour résoudre ce problème, différentes méthodes visant à accélérer la simulation ont été proposées. Parmi ces méthodes, nous nous intéressons ici à la méthode par échantillonnage de l'application, appelée "Sampling". Cette dernière consiste à choisir des phases représentatives (échantillons ou samples) pour réaliser la simulation de l'application. Ces phases correspondent à des petites portions de l'application et constituent un sous ensemble de l'ensemble des instructions de l'application. Les échantillons choisis représentent le com-

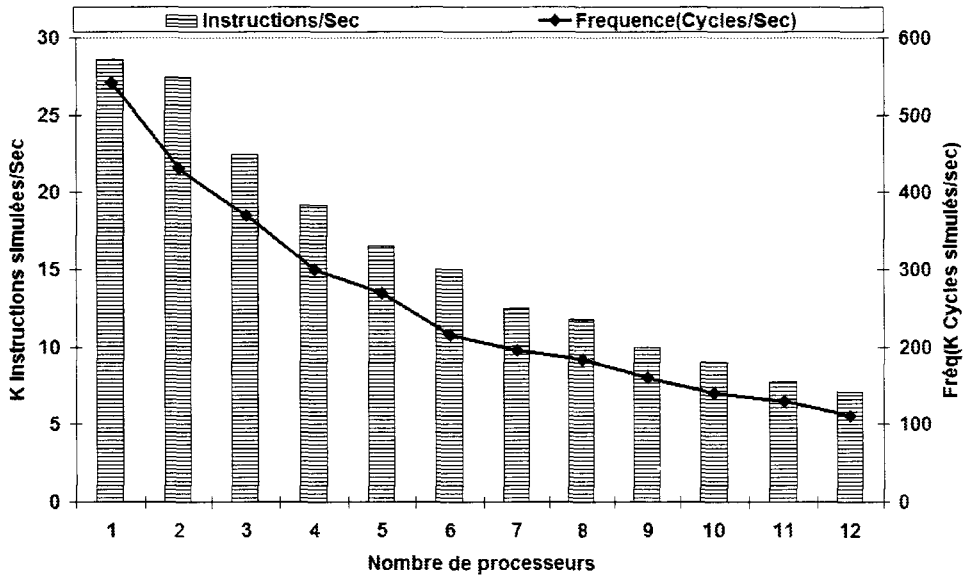


FIG. 1.2 – Vitesse de simulation (en instructions/seconde) et la fréquence correspondante en cycles/-sec pour Rijndael exécutée avec différents nombres de processeurs ARM.

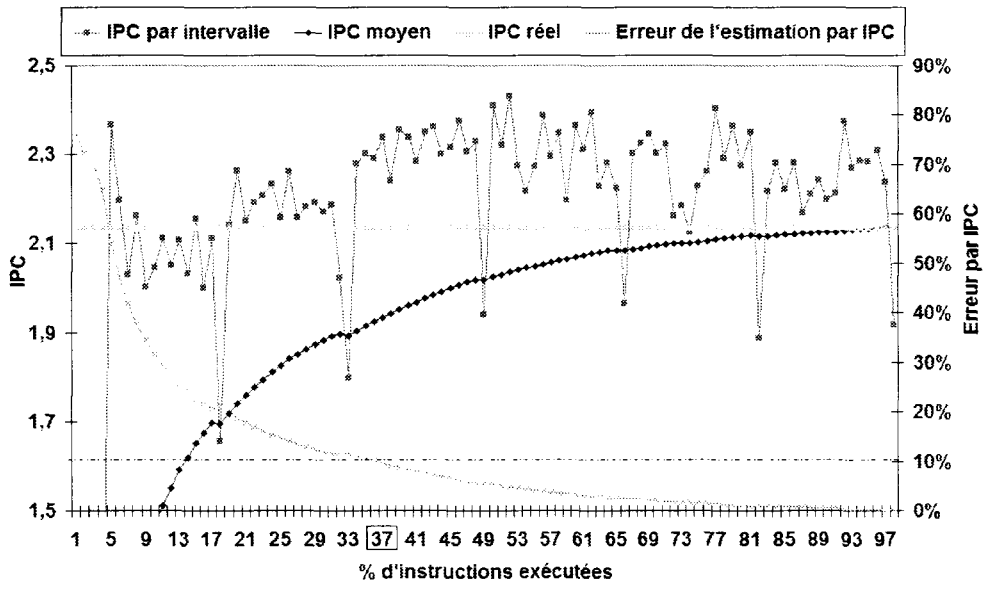


FIG. 1.3 – L'IPC par intervalle, l'IPC moyen, l'IPC total réel et l'erreur de l'estimation pour FFT sur 4 processeurs.

portement de l'application entière sur une plateforme architecturale donnée.

La figure 1.3 qui montre les résultats expérimentaux de l'application FFT , expose l'avantage de l'utilisation de la méthode par échantillonnage pour les systèmes embarqués. Dans cette figure, l'axe des abscisses correspond au nombre d'instructions exécutées. L'axe des ordonnées donnent quatre valeurs :

1. Le nombre moyen d'instructions exécutées par cycles ou (IPC) par intervalle de 50K instructions.
2. L'IPC depuis le début de l'application jusqu'au point x.
3. L'IPC de l'application entière qui est indépendant de x.
4. L'erreur en utilisant l'IPC moyen pour calculer l'IPC total (sur l'axe y à droite).

Comme on peut le voir, la simulation d'un seul intervalle de l'application ne peut pas produire une estimation précise de l'IPC total.

A titre d'exemple, pour obtenir une erreur d'estimation inférieure à 10%, au moins 37% de l'application doit être simulé, donnant ainsi un facteur d'accélération limité de 2.7. Ceci démontre que pour obtenir une accélération plus grande, il est nécessaire de choisir plusieurs échantillons.

Échantillonnage des applications dans les architectures multiprocesseurs embarquées

Si la méthode par échantillonnage a été appliquée de manière extensive dans le cas des systèmes monoprocesseurs, son application dans le cas des systèmes multiprocesseurs a été moins étudiée. En effet dans ce cas, son utilisation pose un certain nombre de problèmes. Parmi les problèmes rencontrés, nous citons la difficulté à déterminer à l'avance les phases parallèles exécutées simultanément par les différents processeurs.

Alors que, les phases à simuler peuvent être déterminées à priori pour les systèmes monoprocesseurs, ceci n'est pas possible pour les systèmes multiprocesseurs. En effet à cause des conflits sur les ressources architecturales partagées, la disposition des phases entre les différents processeurs change d'une configuration architecturale à l'autre. La figure 1.4 montre le problème pour deux processeurs. Chaque application est décomposée en phases similaires et répétitives. Ces phases ont une taille identique de point du vue du nombre d'instructions exécutées. L'application 1 du processeur "Proc1" alterne entre deux phases, "a" et "b". L'application 2 commence par la séquence de phases suivante : "x x y y y x z t". Ici les phases avec le même identificateur correspondent à l'exécution de la même partie du code. En exécutant chaque application individuellement (cas de monoprocesseur), les intervalles appartenant à la même phase produisent le même nombre de cycles (voir figure 1.4.a). A l'opposé, quand ces deux applications s'exécutent simultanément sur 2 processeurs, les phases influent les unes sur les autres, ce qui provoque un décalage des phases (voir figure 1.4.b). A cause des ressources partagées (bus, mémoire, etc.), les performances ainsi que le comportement des applications parallèles deviennent inter-dépendants. Dans cette situation, la détermination à priori des phases s'exécutant simultanément est impossible.

1.2 Contributions

Notre contribution dans le domaine de la simulation des MPSoC, à travers cette thèse, est de proposer des méthodes permettant de résoudre les problèmes présentés dans la section précédente. En résumé, nos contributions sont :

1. Une méthode a été proposée dans le but d'accélérer la simulation pour les MPSoCs en appliquant l'échantillonnage de l'application à exécuter. Elle consiste à combiner d'une façon intelligente les phases parallèles. Ainsi les combinaisons des phases qui

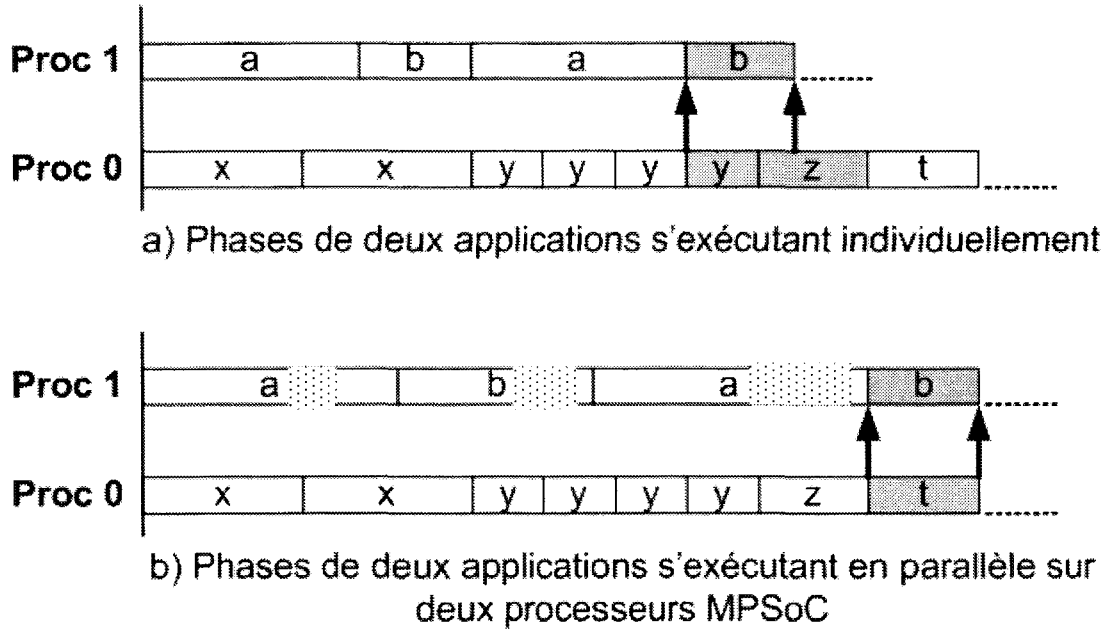


FIG. 1.4 – Simulation des phases sur 2 processeurs MPSoC. a) Exécution individuelle : b s'exécute en même temps que y et z. b) Exécution en parallèle où Proc1 est retardé à cause de la contention : b s'exécute maintenant en parallèle avec t et non pas avec y et z.

se répètent ne seront pas simulées (on dira aussi sautées). Par conséquent, elles ne subiront pas la simulation détaillée. Pour certaines applications, l'accélération a atteint un facteur de 800.

2. Toujours en se basant sur l'échantillonnage, une deuxième méthode a été conçue. Cette méthode combine les phases pour produire une seule phase par processeur. La taille de la phase est adaptée dynamiquement selon le comportement de l'application. Ceci facilite la détection des phases des applications. Ainsi pour les applications ayant des besoins différents pour les ressources, l'accélération est nettement améliorée.
3. L'utilisation de l'échantillonnage d'application nécessite de résoudre deux problèmes avant de démarrer la simulation d'un intervalle. Il s'agit de trouver d'une part un moyen pour restaurer l'image exacte du contexte de l'application et d'autre part un autre moyen pour restaurer l'état micro-architectural (mémoire cache, prédicteur de branchements). Ces deux tâches doivent être faites au démarrage de chaque intervalle simulé. Nous avons étudié l'utilisation de la méthode par "points de contrôle" ou "checkpoints" pour réaliser ces deux tâches. Ainsi la proposition d'une technique qui utilise d'une manière efficace les checkpoints avec notre méthode d'accélération a été réalisée.
4. La classification des phases proposées dans SimPoint[1] a été étendue dans le cas des MPSoC. Le but ici est de supporter l'analyse et la classification des intervalles de différentes tailles. Ceci permet une utilisation efficace des checkpoints pour construire l'état exact du système. Comme nous le verrons par la suite, la nouvelle classification

[1]http ://www.cse.ucsd.edu/ calder/simpoint/simpoint_overview.htm.

des phases est capable d'augmenter l'accélération tout en diminuant l'erreur sur l'estimation.

1.3 Plan

Notre manuscrit est organisé selon le plan suivant :

Chapitre 2) État de l'art : Dans ce chapitre nous présentons les différentes méthodes visant à accélérer la simulation pour les systèmes monoprocesseurs et multiprocesseurs. Nous présentons aussi les techniques utilisées avec la méthode par échantillonnage pour restaurer respectivement l'image exacte du contexte de l'application et l'état micro-architectural avant de démarrer la simulation d'un intervalle. A partir de cette étude, nous positionnerons nos travaux et nous donnerons les grandes lignes de nos contributions.

Chapitre 3) Accélération de la simulation par "Échantillonnage Adaptatif" : Dans ce chapitre nous présenterons notre méthode d'échantillonnage pour accélérer la simulation dans les systèmes MPSoC. Nous détaillerons le fonctionnement de cette méthode en particulier pour les points suivants : Le mécanisme utilisé pour combiner les phases parallèles, la détection et la réalisation des sauts des combinaisons qui se répètent. Les résultats expérimentaux obtenus ont montré que l'accélération peut être très importante si les applications ont un comportement similaire (besoins identiques en ressources matérielles).

Chapitre 4) Synthèse de phases et adaptation des tailles des intervalles : Deux techniques complémentaires à la méthode proposée dans le chapitre 3 sont présentées dans ce chapitre pour améliorer l'accélération. La première technique consiste à synthétiser des combinaisons de phases en utilisant celles déjà simulées. La deuxième technique consiste à adapter les tailles des intervalles en se basant sur les références mémoires des applications. Là aussi une étude des résultats expérimentaux est montrée pour ces deux techniques.

Chapitre 5) Échantillonnage par Multi-Granularité : Pour obtenir une accélération importante pour les applications ayant des besoins différents en ressources matérielles, une nouvelle méthode par échantillonnage sera proposée dans le chapitre 5. Nous présenterons le principe de fonctionnement de cette méthode qui analyse différentes tailles des intervalles (appelées granularités dans notre cas). Ainsi cette méthode combine les phases pour produire une seule phase par processeur. Cette solution est capable de détecter toutes les phases des applications, ce qui permet une accélération importante.

Chapitre 6) Utilisation du "checkpointing" pour la simulation des MPSoC : Une étude concernant l'utilisation des checkpoints pour restaurer l'image exacte du contexte de l'application et l'état micro-architectural sera présentée dans le cadre de la méthode proposée dans le chapitre 5. De plus, une extension de la classification des phases de SimPoint sera proposée. Cette extension consiste à analyser et classifier en même temps des intervalles ayant des tailles différentes dans le but, d'une part, d'utiliser efficacement les checkpoints et d'autre part d'améliorer l'accélération et la précision de simulation.

Chapitre 7) Conclusion Une conclusion de cette thèse ainsi qu'un bilan des travaux effectués seront présentés. Nous détaillerons les contributions apportées avant d'aborder

quelques perspectives à nos travaux.

A la fin de ce mémoire, nous avons ajouté un glossaire qui rappelle les principaux termes et notions utilisés dans la thèse.

Chapitre 2

État de l'art

2.1 Introduction

La conception des systèmes MPSoC s'appuie fortement sur les simulateurs pour évaluer et valider des nouvelles plateformes avant la fabrication. L'idée, permettant de trouver rapidement la meilleure configuration, repose sur la nécessité d'une exploration rapide et précise de l'espace des alternatives architecturales accompagnée d'une estimation rapide des performances de chaque alternative à explorer. Ceci accélère la phase de conception des MPSoC.

La motivation principale de notre travail est la mise au point de méthodes permettant de réduire le temps de simulation afin d'obtenir les performances des différentes alternatives dans un temps court. Nous nous intéressons en particulier à réduire le temps de simulation en considérant les différentes phases de l'application. Cette méthode dite méthode par *"échantillonnage de l'application"* consiste à choisir un nombre réduit d'intervalles non consécutifs pour représenter toute l'application.

L'application de cette méthode nécessite des outils pour construire le contexte exact du programme et l'état exact des structures architecturales au début de chaque intervalle qui suit une interruption de la simulation.

Cet ensemble d'objectifs nous amène à présenter dans ce chapitre, centré sur l'état de l'art, les différentes méthodes d'accélération de la simulation en mettant particulièrement l'accent sur la méthode par échantillonnage. Nous survolons ensuite les techniques existantes dans le cadre des méthodes par échantillonnage pour la construction du contexte du programme et de l'état des structures architecturales. Ces deux outils sont nécessaires pour toute méthode d'accélération performante.

2.2 Nécessité des méthodes de simulation des systèmes mono-puce

La conception des MPSoC devient de plus en plus complexe. Cela est dû à plusieurs raisons :

1. L'augmentation du nombre de transistors sur une même puce avec plus d'un milliard de transistors en 2010.
2. L'hétérogénéité des architectures proposées qui devient nécessaire pour respecter les contraintes imposées par l'application et le marché.
3. La taille croissante des nouvelles applications que ces systèmes doivent supporter.
4. Le nombre important de contraintes à respecter : Assurer un niveau de la performance en puissance de calcul et en consommation d'énergie, réduire le *"time to market"* et du coût final du système, etc.

Pour respecter ces contraintes, une simulation conjointe des deux parties logicielle et matérielle, appelée aussi "co-simulation", est nécessaire. La co-simulation consiste alors à simuler l'ensemble du système et à fournir au concepteur des informations sur le déroulement de l'application sur la plateforme matérielle : le temps d'exécution, la consommation d'énergie, etc. Ces informations permettent de localiser les paramètres à optimiser/modifier dans le logiciel, dans le matériel ou dans l'association qui a été réalisée entre ces deux derniers. Cela est fait grâce à une modélisation du système complet qui prend en compte le logiciel et le matériel en utilisant un langage de description des deux parties simultanément.

Les langages de description des systèmes embarqués les plus utilisés sont :

SystemC [52] . Ce langage a pour objectif de modéliser le matériel et le logiciel d'un SoC en se basant sur des librairies C++. Il introduit de nouveaux concepts, par rapport au langage C++, afin de supporter la description du matériel et ses caractéristiques inhérentes comme la concurrence et l'aspect temporel. Ces nouveaux concepts sont implémentés par des classes C++, tels que les modules, les ports, les signaux, les FIFO, les processus (threads et methods). Depuis décembre 2005, SystemC est standardisé auprès de l'IEEE sous le nom de IEEE 1666-2005. Au cours de ces dernières années, SystemC a suscité un intérêt de plus en plus grand auprès des chercheurs et des industriels relativement à d'autres langages. Cependant, ce langage n'est, pour le moment, pas synthètisable. Il n'est donc pas possible de l'utiliser au-delà du niveau RTL. Pour cette raison, SystemC est utilisé en général pour les tâches de modélisation et de vérification rapide du système. De plus, comme il n'a été standardisé que récemment, son adoption par les grands consortiums de systèmes sur puce risque de demander un certain temps. Le simulateur du système MPSoC sur lequel les expériences ont été réalisées a été conçu avec SystemC.

VHDL [12] est destiné à décrire le comportement et/ou l'architecture d'un module matériel. L'utilisation de ce langage dans le monde industriel ou académique est fortement liée à une phase avancée de la conception du système. VHDL permet une description du système au niveau RTL. L'intérêt d'une telle description réside dans son caractère exécutable permettant de vérifier le comportement réel du circuit. En comparaison avec SystemC, le langage VHDL ne permet pas la vérification de la conception à un haut niveau d'abstraction notamment au niveau transactionnel ou TLM [26]. Ainsi, dans un même environnement de co-simulation, ces deux langages peuvent être utilisés dans des phases de conception différentes suivant le niveau d'abstraction. Ils peuvent donc être considérés comme deux langages complémentaires.

Verilog [58] est conçu pour modéliser des circuits logiques, jusqu'au niveau du transistor. Il hérite du langage C. Il reste que VHDL et Verilog sont deux approches à la fois partiellement recouvrantes et complémentaires. Il existe d'ailleurs sur le marché des outils capables de simuler des modèles mixtes VHDL et Verilog. Récemment, une hybridation du Verilog et du SustemC a été propsosée. Ce qui a amené la définition du langage SystemVerilog [13].

Aujourd'hui la co-simulation, au "niveau cycle précis bit précis" ou Cycle-Acurate-Bit-Accurate (CABA), des systèmes multiprocesseurs mono-puce, nommés MPSoC, est devenue un outil nécessaire pour la conception de ces systèmes [17, 55]. Nous avons montré dans les motivations de cette étude (cf. chapitre précédent) que les temps de simulation à ce niveau d'abstraction pour ces systèmes ne sont pas satisfaisants. Pour cela, dans le cadre de notre travail, nous mettons l'accent sur la réduction du temps de simulation des systèmes MPSoC. De plus, nous utilisons un simulateur [17] qui adopte le langage SystemC comme langage de description pour les MPSoC. Notons que notre travail est indépendant du langage de description ainsi que de la plateforme architecturale adoptée. Dans la section suivante, nous étudierons les méthodes existantes pour accélérer la simulation des systèmes MPSoC.

2.3 Méthodes d'accélération de la simulation

Afin de trouver la configuration architecturale la plus convenable d'un système embarqué pour une application spécifique ou un ensemble d'applications spécifiques, il est nécessaire d'explorer un grand nombre de configurations architecturales. Le défi est de réduire la phase d'exploration dans le DSE *"Design Space Exploration"*, afin de réduire le temps de mise sur le marché. La réduction de la DSE peut être faite à deux niveaux :

A) En réduisant le nombre d'alternatives architecturales à évaluer, en utilisant une heuristique intelligente pour guider la recherche vers les configurations les plus prometteuses. Des métaheuristiques comme les algorithmes génétiques ou la recherche tabou ont été largement utilisées [29, 50, 14, 48].

B) En accélérant le processus d'évaluation des performances associé à chaque alternative. Ceci vise la réduction du temps de la fonction coût de chaque alternative.

Dans cette thèse, nous nous intéressons à cette deuxième solution.

Toute méthode visant l'accélération de la simulation pour une évaluation rapide des performances des MPSoC doit tenir compte des facteurs suivants :

1. Le temps de simulation et la précision souhaités pour l'évaluation de chaque alternative.
2. Le niveau d'abstraction dans lequel est décrit le système.

Les méthodes existantes et visant la réduction du temps d'évaluation associé à une alternative dans le DSE peuvent être divisées en trois grandes familles. Ces trois familles sont détaillées ici.

2.3.1 Méthode d'accélération par abstraction du niveau de description des systèmes

L'accélération de la simulation est obtenue ici par l'utilisation, dans la descritpion du système, de niveaux d'abstraction plus élevés que le niveau CABA. La méthode TLM (*Transaction Level Modeling*) [26] fait partie de cette approche. Dans TLM, les temps de simulation sont réduits en augmentant la granularité des opérations de communication. Par conséquent, les surcharges (ou overheads) associées à la synchronisation et la communication entre composants de l'architecture sont réduites.

L'un des projets qui ont implémenté le TLM dans le cas des MPSoC est [61]. Pour palier au manque de précision ici lorsque le processeur désire envoyer une requête vers la mémoire, la requête comprend en plus des informations habituelles, la valeur de l'horloge locale. Au niveau du réseau d'interconnexion et du module mémoire cible, un paquet contenant l'horloge locale sur chaque canal d'entrée doit être reçu avant de faire avancer l'horloge locale du réseau ou de la mémoire. Cette approche nécessite néanmoins l'utilisation de message "nuls" afin de faire avancer les horloges, ce qui peut entraîner une augmentation de la surcharge sur le simulateur. Les valeurs des accélérations obtenues sont assez importantes, mais celles-ci n'ont pas été mesurées sur des applications réelles. L'inconvénient majeur de cette approche est le manque de précision dans le cas d'architectures MPSoC complexes.

2.3.2 Méthode d'accélération de simulation au niveau CABA

Dans ce deuxième groupe, la simulation est réalisée en cycle précis bit précis.

Au niveau Cycle Précis Bit Précis (CABA), le système est décrit de façon précise du point de vue temps d'exécution des instructions. Ce niveau permet de simuler le comportement des composants au cycle près. En effet, une description de la micro-architecture interne du processeur (pipeline, prédiction de branchement, cache, etc.) des mémoires et du réseau d'interconnexion est aussi réalisée. Au niveau communication entre modules d'architectures, un protocole de communication précis au bit près est adopté. Cette description détaillée et fine du système permet d'améliorer la précision de l'estimation des performances. Les principales techniques de cette famille sont :

2.3.2.1 Simulation statistique

La simulation statistique(SS) [51] est une méthode qui vise l'accélération de la simulation CABA. Elle se déploie en trois étapes :

1. Acquérir les statistiques du programme à exécuter pour générer le programme synthétique (SP). Certaines de ces statistiques ne dépendent que du programme d'origine à exécuter et de ses entrées, comme les types d'instructions. A l'opposé, d'autre statistiques dépendent non seulement du programme mais aussi de l'architecture matérielle de la plateforme, tels que taux de défauts cache, ou les défauts de prédiction de branchement.
2. Générer le programme synthétique (SP). Un programme synthétique ayant les statistiques mesurées dans l'étape 1 est généré. Pour que ce programme soit utile, il doit être court et il doit posséder les mêmes statistiques que le programme d'origine. Plus la longueur, en nombre d'instructions, du programme synthétique est réduite, meilleure sera l'accélération de la simulation mais l'erreur peut être importante.
3. Exécuter le programme SP. Dans cette dernière étape de la SS, le programme SP est exécuté sur le simulateur au niveau CABA afin de mesurer la performance en termes d'IPC (instructions par cycles) et d'EPC (énergie par cycle).

La méthode SS est intéressante mais exige plusieurs tests et différents simulateurs pour extraire les statistiques nécessaires au SP.

2.3.2.2 Réduction du jeu de données en entrée

L'objectif de ces méthodes est d'accélérer la simulation en réduisant le jeu de données en entrée sans modifier le simulateur décrit au niveau CABA. Les données réduites sont construites à partir des données de référence et sont utilisées à la place de ces dernières. Kleinosowski et al. [43] proposent une méthode fondée sur cette idée. Cette méthode collecte le profil d'exécution au niveau des fonctions précisant ainsi la fraction du temps total d'exécution consacrée pour chaque fonction. La réduction du jeu de données varie d'un benchmark à l'autre. Pour certains benchmarks, les fichiers d'entrée nommés aussi "input files" sont tronqués. Pour les applications sans fichiers d'entrée, le code source est analysé afin de réduire le nombre d'itérations des boucles.

Après chaque tentative de réduction de jeu de données, les profils au niveau des fonctions sont comparés à ceux obtenus par les données de référence. Autrement dit, pour chaque fonction, la fraction du temps d'exécution au temps total dans le cas des données de référence est comparée à celle obtenue avec les données réduites. Les données réduites

pour lesquelles le profil d'exécution est le plus proche de celui de données de référence sont utilisées.

Il est à noter que cette méthode a un inconvénient majeur du fait qu'il est nécessaire de construire le jeu de données réduit pour chaque application, ou version de l'application. Ce problème se pose par exemple quand on désire explorer l'espace des optimisations du code afin de trouver celui qui donne les meilleures performances. Ce processus est généralement appelé COSE pour Compilation Optimization Espace Exploration. En effet dans un COSE, chaque alternative exige un jeu de données réduit qui lui est spécifique. Ceci nécessite plusieurs heures de simulation.

2.3.2.3 Modélisation analytique

Ces méthodes consistent à réaliser une évaluation des performances basée sur la modélisation analytique de l'architecture. Il s'agit ici de produire un ensemble d'équations permettant de spécifier les performances de l'architecture. Ces équations utilisent comme paramètres les caractéristiques de l'architecture telles que : les tailles des mémoires caches, les tailles des queues des instructions, le nombre d'unités fonctionnelles, etc.

A titre d'exemple, Karkhanis et al. [38] proposent une méthode en deux phases. La première phase consiste à définir la valeur de CPI en considérant pour une taille donnée de la queue des instructions nommées aussi "instruction issue window".

Dans la seconde phase, des pénalités dues aux mauvaises prédictions de branchement et aux défauts dans les caches d'instructions et de données sont ajoutées. Ces pénalités sont calculées grâce à l'établissement d'une relation entre le nombre d'instructions dans la queue des instructions et le CPI.

Pour des architectures élémentaires et monoprocesseurs, cette approche peut être intéressante. Néanmoins quand l'architecture est complexe et multiprocesseur, le nombre de paramètres à prendre en compte pour la modélisation des performances peut être très important et difficile à cerner. Dans ce cas, la précision du modèle devient très faible.

2.3.2.4 Échantillonnage de l'application

La méthode par échantillonnage de l'application, ou *"sampling"*, réduit le temps de simulation en réalisant la simulation détaillée que sur un nombre limité d'instructions de l'application. Ainsi, au lieu d'exécuter l'application dans sa totalité, seulement un nombre réduit d'intervalles d'instructions, considérés représentatifs de l'application, est exécuté. Ces intervalles représentatifs sont appelés échantillons ou *"samples"*.

Les échantillons de l'application sont choisis soit d'une façon périodique, exemple SMARTS [64, 63], soit grâce à une analyse des intervalles de l'application, comme dans le projet SimPoint [57, 32, 54].

Dans la suite, nous allons présenter brièvement ces deux projets.

SMARTS

Dans le SMARTS, on choisit périodiquement les échantillons qui vont subir la simulation détaillée. La taille de chaque échantillon est de l'ordre d'un millier d'instructions et le nombre d'instructions qui sépare deux échantillons consécutifs est de l'ordre d'un million d'instructions. La figure 2.1 montre l'échantillonnage de l'application dans le cadre de

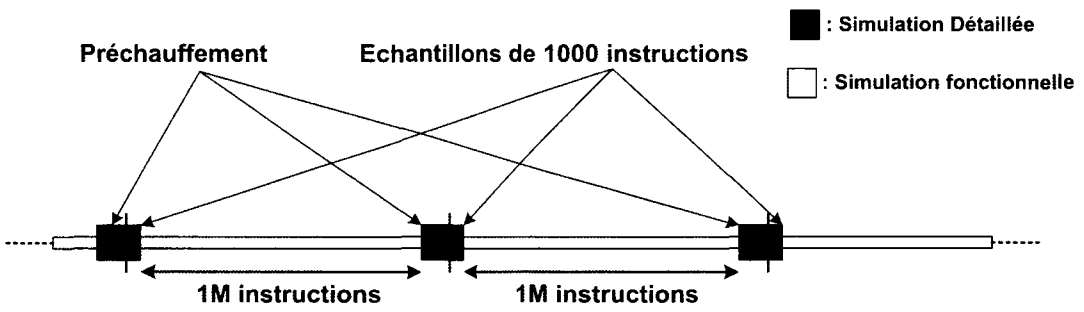


FIG. 2.1 – Échantillonnage de l'application réalisé dans le cadre de SMARTS. Le préchauffement utilisé pour préparer l'état micro-architectural est expliqué dans la suite de ce chapitre.

SMARTS. Ce dernier considère que chaque échantillon de mille instructions représente le million d'instructions qui lui succède. Ainsi, on considère que la performance de chaque échantillon est représentative de la partie suivante des instructions non simulées en détail.

La faiblesse fondamentale de SMARTS est qu'il ne tient pas compte des phases réelles du programme. Si la simulation détaillée se produit immédiatement avant un changement de phase, l'échantillon ne sera pas réellement représentatif car il reflète une période de transition. SMARTS résout ce problème en prenant des échantillons fréquemment, ce qui augmente le nombre d'échantillons nécessaires. Une analyse statistique est utilisée pour indiquer à l'utilisateur le nombre d'échantillons à considérer pour aboutir à un certain niveau de confiance.

SimPoint

SimPoint est un outil très puissant qui permet l'analyse de l'exécution d'un programme. Il détermine les parties de l'application où le même code est (approximativement) ré-exécuté.

L'exécution du programme est d'abord divisée en intervalles consécutifs dont la taille est fixe et correspond à celle des échantillons choisis. Un profil est collecté pour chaque intervalle. Ce profil est indépendant de la configuration architecturale du système. Il est représenté par des vecteurs de blocs de base nommés BBVs pour *"Basic Block Vectors"*.

Un BBV est un tableau de blocs de base (BB) statiques du programme. Un BB est une séquence d'instructions ayant un seul point d'entrée qui correspond à une étiquette de branchement et un seul point de sortie qui peut correspondre à un branchement. Un BB contient au plus un branchement. Les BB n'ont pas tous la même taille.

Les BBVs ne dépendent pas de l'architecture du système mais des entrées du programme. La figure 2.2 montre la génération des BBVs pour trois intervalles I_1, I_2 et I_3. L'élément i de chaque BBV correspond au nombre de fois que le bloc de base i, noté BB_i, est exécuté pendant l'intervalle.

La distance entre deux BBVs détermine de combien les deux intervalles correspondants sont différents. Le grand nombre de blocs de base dans le programme, ou dans un intervalle, augmente de manière importante le temps de comparaison des BBVs. Pour cela, une projection aléatoire à partir de l'espace de dimension original des BB vers un espace de plus petite dimension est proposée pour diminuer le coût de la comparaison de deux vecteurs. Si deux vecteurs sont similaires avant la projection, ils seront toujours similaires après celle-ci.

SimPoint permet de choisir un échantillon unique de l'application. Dans ce cas les BBVs,

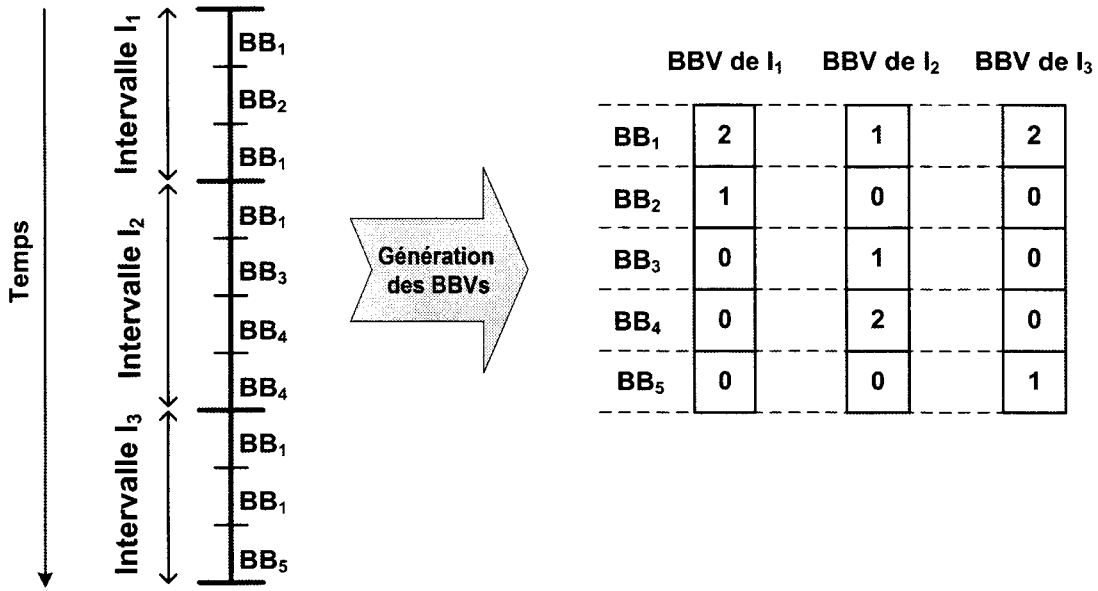


FIG. 2.2 – Génération des BBVs pour trois intervalles de l'application.

pour chaque intervalle, sont donc comparés à un BBV qui représente l'exécution total du programme dans le but de trouver un intervalle unique qui serait le plus proche de l'exécution complète du programme. Cette technique est utilisée dans [56].

Plus généralement, SimPoint peut être utilisé pour classer les intervalles du programme en phases. Ainsi les intervalles ayant le même comportement constituent une phase de l'application. SimPoint détecte la similarité des intervalles en utilisant l'algorithme de classification k-means [47], qui classifie les BBVs en k groupes. Les groupes sont choisis pour être centrés autour de k points. Chaque intervalle appartient à un seul groupe. Celui avec qui la distance est la plus petite. La distance correspond à la somme des carrés des différences entre le BBV en question et le centre de groupe correspondant. Notons ici qu'il est important de bien choisir la valeur de k. Une valeur trop petite améliore l'accélération mais réduit la précision et vice versa. Pour cela, SimPoint essaie plusieurs valeurs de k et garde celle qui modélise précisément la distribution des BBVs. Le *Bayesian Information Criterion* (BIC) [39] est utilisé pour comparer la qualité de classification avec différents nombres de groupes.

Enfin, SimPoint sélectionne l'intervalle le plus représentatif de chaque phase. En effet, le centre de la phase est la moyenne des BBVs appartenant à la phase correspondante, appelé aussi le centroide. Ainsi l'intervalle ayant le BBV le plus proche au centroide est appelé "intervalle représentatif" et il est donc choisi comme étant le représentatif de cette phase. La simulation de cet intervalle , appelée point de simulation, va donner une exécution représentative de la phase. La figure 2.3 montre la classification des intervalles dans trois phases. Dans chaque phase, l'intervalle représentatif est l'intervalle le plus proche au centroide. La collection des statistiques de tous les points de simulation pondérés par le nombre d'intervalles de la phase permet d'estimer la performance du programme tout entier. La figure 2.4 montre la relation entre les phases et les performances de l'application. Ici il s'agit de l'IPC moyen et du taux de défauts dans le Dcache. Comme on peut le voir, chaque phase est représentée par un identificateur et chaque intervalle d'exécution est associé à une phase.

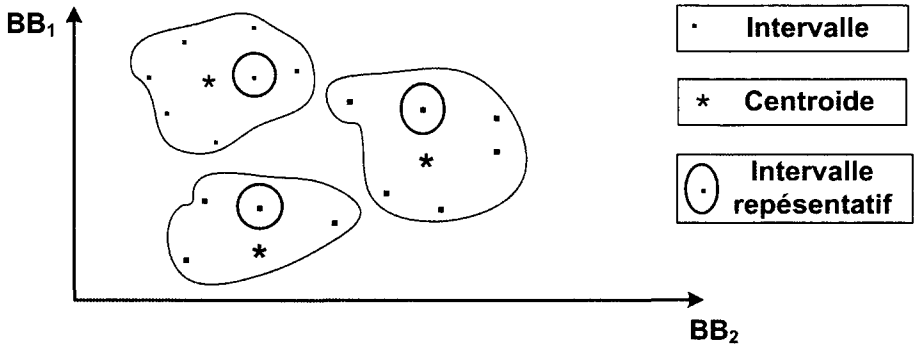


FIG. 2.3 – Classification des intervalles dans trois phases. Chaque point correspond à un BBV contenant les fréquences des deux blocs de base, BB_1 et BB_2.

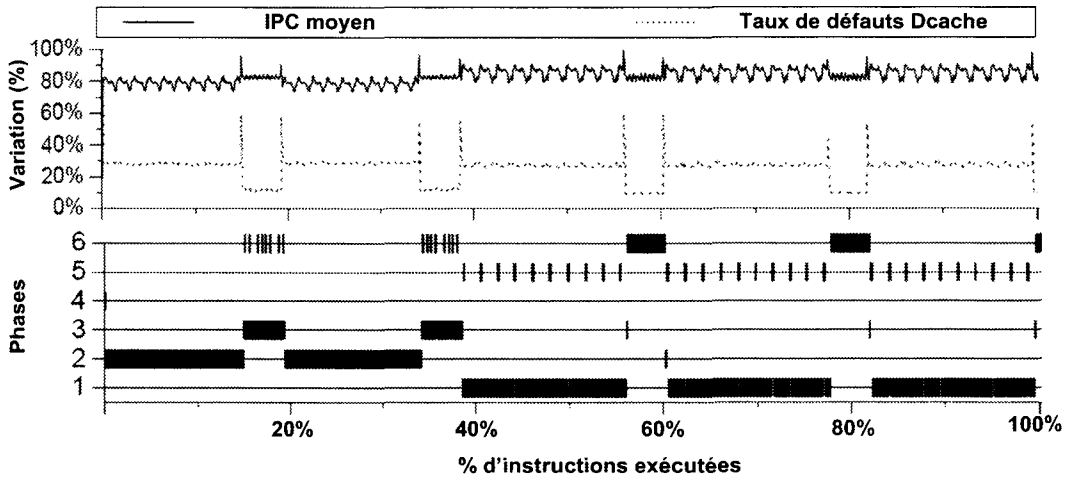


FIG. 2.4 – Le graphique du haut montre l'IPC moyen et le taux de défauts Dcache pour chaque intervalle de 100 millions instructions. Le graphique du bas montre la répartition des phases durant l'exécution. Ces graphiques sont donnés par [57].

En général, deux facteurs permettent d'améliorer la précision [32, 54] :

1. L'augmentation du nombre de phases. Ce nombre est noté k.
2. L'augmentation de la taille des intervalles.

En effet, un nombre important de phases représentées par des courts intervalles peut donner un bon compromis entre accélération et précision [32, 54].

Dans [44], une méthode a été présentée pour utiliser des intervalles de tailles variables. Ces intervalles sont alignés avec les appels, les retours de procédures et les bornes de transition de boucles du code source. Cela est réalisé en profilant le code afin d'identifier les instructions d'appels et de retour des applications ainsi que les branchements. Une taille d'intervalle variable, nommée VLI pour *"Variable Length Interval"* est aussi présentée par un BBV (voir figure 2.2). En effet, une fois que les tailles des intervalles sont déterminées, la version SimPoint 3.0 [45] supportant le VLI est utilisée afin de classifier ces intervalles.

Par ailleurs, des travaux [37] ont été menés dans le cadre du SimPoint mettant en oeuvre

un algorithme de classification des BBVs différent de k-means. Cet algorithme appelé EDCM pour "Exponential Dirichleet Compound Multinomial" ne suppose pas de préalable de la projection des BBVs contrairement à l'algorithme k-means. Cette modification améliore la rapidité de SimPoint tout en gardant le même niveau de précision lorsque le nombre de phases détectées est important.

PGSS-Sim

SMARTS ne prend pas en compte les phases de l'application; il surestime le nombre d'échantillons pour aboutir à un niveau élevé de précision, tandis que SimPoint génère des échantillons de grosses granularité et néglige donc les phases de fine granularité dans l'application. Pour cela, Kihm et al. [42] présentent une méthode nommée PGSS-Sim pour *"Phase-Guided Small-Sample Simulation"* qui est une combinaison de SMARTS et de SimPoint. Ceci permet de bénéficier des avantages de ces deux méthodologies. SMARTS travaille avec des échantillons de fine granularité (chaque échantillon correspond à un millier d'instructions) alors que SimPoint permet une accélération basée sur la détection des phases de l'application. Dans le cadre de PGSS-Sim, les BBVs sont utilisés pour guider SMARTS à sélectionner les échantillons. La taille de chaque échantillon, comme dans SMARTS, est d'un millier d'instructions. La seule différence est que le nombre d'instructions qui subissent la simulation fonctionnelle est plus réduit (100K instructions au lieu d' un million instructions). La simulation fonctionnelle consiste à avancer rapidement la simulation vers un point donné tout en négligeant les détails architecturaux du système. En effet, le court intervalle de simulation fonctionnelle permet de détecter les information de phases de fine granularité.

La figure 2.5 montre le fonctionnement de PGSS-Sim. Après chaque échantillon de mille instructions, un BBV est généré dynamiquement durant la simulation fonctionnelle de 100K instructions et il est ensuite associé à une phase unique. Ainsi, chaque BBV généré est comparée avec les BBVs précédents afin de l'affecter à une phase. S'il n'a pas une similarité avec les autres BBVs, une nouvelle phase est construite contenant ce nouveau BBV.
De plus comme on peut le voir dans la figure 2.5, PGSS-Sim consiste à simuler plusieurs échantillons pour une même phase. Quand la phase du BBV est détectée, les échantillons, collectés précédemment pour cette phase, sont testés pour vérifier si ils aboutissent à une limite donnée de confiance[2]. Si tel est le cas, la collection d'échantillons pour la phase en question n'a plus lieu d'être. Ainsi un nouvel intervalle de 100K instructions est prêt à subir la simulation fonctionnelle. Si la limite de confiance n'est pas atteinte ou si aucun échantillon n'est déjà collecté alors un nouvel échantillon doit être réalisé pour cette phase. Dans ces conditions, la distance entre l'échantillon suivant, qui doit être réalisé, et le dernier échantillon collecté pour la même phase est testée.

Dans le cas où cette distance est supérieure à 1 million d'instructions, un nouvel échantillon pour la phase en question prend effet suivi de l'intervalle de 100K instructions de simulation fonctionnelle. Dans le cas contraire, la limite de confiance des échantillons est atteinte, un intervalle de la simulation fonctionnelle est mis en oeuvre à la place de la simulation détaillée d'un échantillon de mille instructions. La vérification de la distance entre les échantillons de la même phase permet de distribuer les échantillons à travers les occurrences de cette phase captant ainsi les variations des performances. En comparant PGSS-Sim à SMARTS et à SimPoint, Kihm et al. ont montré que PGSS-Sim réduit légèrement le nombre

[2]**Intervalle de confiance basé sur le nombre d'échantillons et leur écart type.**

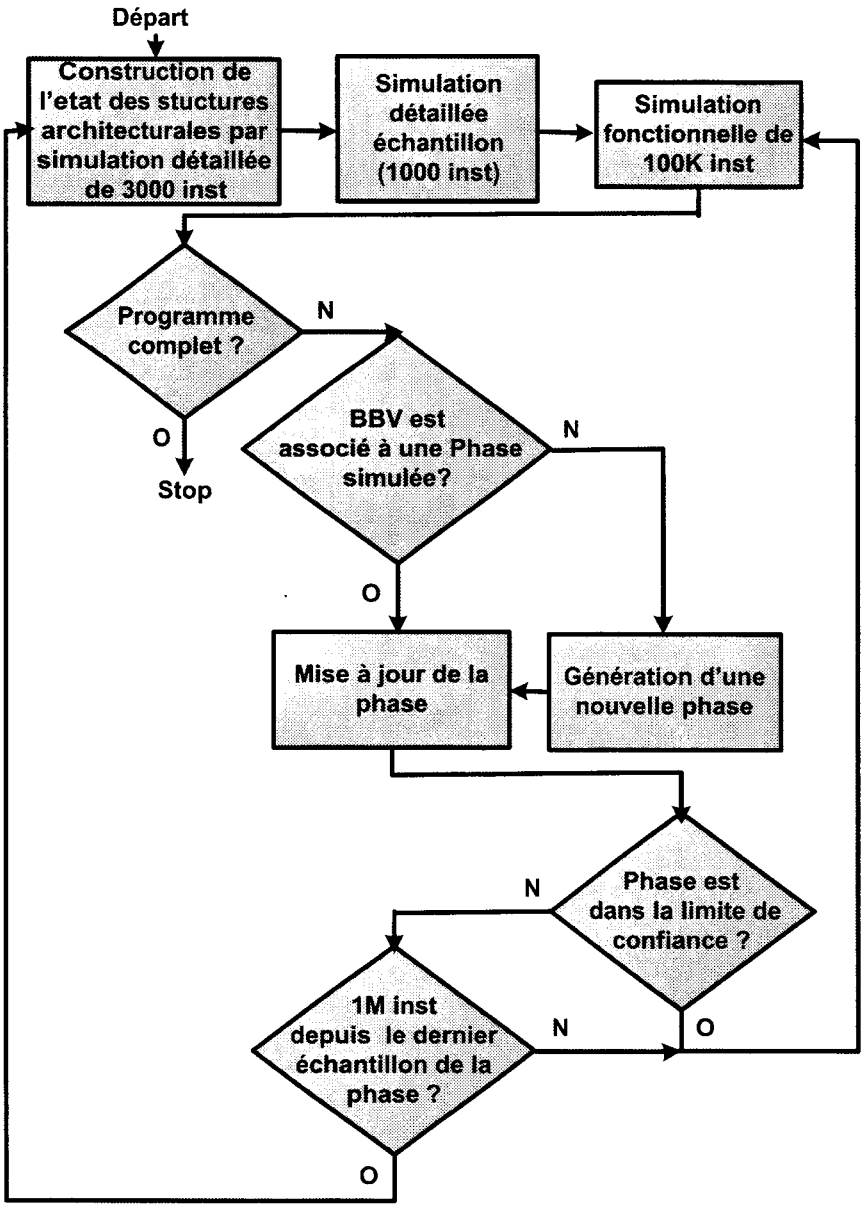


FIG. 2.5 – Le flot du fonctionnement de PGSS-Sim. La construction de l'état exacte des structures architecturales est la même que celle de SMARTS et celle-ci est expliquée dans la suite de ce chapitre. Les BBVs sont générés durant la simulation fonctionnelle. Ce flot est présenté dans [42].

d'instructions qui subit la simulation détaillée tandis que le niveau de précision obtenu par SMARTS et SimPoint est plus élevé. Du point de vue temps de simulation globale PGSS-Sim n'a aucun avantage sur les deux autres.

2.3.3 Accélération de l'évaluation des performances par l'utilisation des circuits reconfigurables

Enfin dans cette troisième famille, l'accélération de la simulation est réalisée en effectuant l'émulation de l'architecture à évaluer sur un système reconfigurable (type FPGA). Del Valle et al. [60] ont adopté cette méthode. Le but ici est d'effectuer le DSE des MPSoC au niveau de trois composants : cores, mémoires et réseaux d'interconnexion. L'une des plus importantes caractéristiques de leur travail demeure la façon transparente d'extraire des statiques durant l'exécution du système. Au niveau de chaque composant, un moniteur qui envoie les statistiques concernant les événements à la machine hôte connectée au FPGA est ajouté. Toutes les statistiques sont collectées sur la machine hôte afin d'évaluer les alternatives architecturales correspondantes. Cette méthode permet une précision très élevée et une accélération de la simulation par rapport à un simulateur CABA. L'inconvénient de cette méthode est le coût élevé de la plateforme spécifique requise ainsi que le temps de développement de chaque composant. Ainsi, chacun des cores et des réseaux d'interconnexion évalués nécessite environ une semaine de développement. Cette méthode est aussi utilisée pour la simulation des systèmes à haute performance dans le cadre du Framework FAST pour *"FPGA-Accelerated Simulation Technologies"* [23].

2.3.4 Méthodes d'accélération hybride

De façon générale, l'émulation sur circuits reconfigurables est plus rapide et plus précise que la simulation réalisée par un logiciel (ou ISS). Mais cette émulation demande des connaissances détaillées sur la structure de chaque composant, ainsi qu'un temps de développement. Ainsi le développement sur un FPGA exige un effort important. Chung et al. [24] réduisent la complexité en proposant une méthode hybride (émulation / simulation par logiciel). Cette approche est basée sur le fait qu'une seule partie du système contribue plus largement dans l'exécution, alors que plusieurs comportements complexes se produisent rarement durant l'exécution. Pour cela, cette méthode hybride réduit la complexité de développement en synthétisant uniquement les opérations fréquentes et simples sur FPGA tandis que les comportements complexes et rares sont réalisés sur le simulateur. Ainsi dans leur cas, ils distribuent la complexité du système (par exemple les processeurs) sur un ou plusieurs FPGAs reliés à la machine hôte où se déroule la simulation.

Depuis un simulateur complet, ils sélectionnent les composants qui dominent la performance pour les implémenter sur FPGA. Quand un processeur émulé sur FPGA rencontre un événement non implémenté, par exemple un défaut dans la table de TLB, il est interrompu et son état est transporté à l'instance du processeur sur le simulateur. L'instance effectue l'événement au niveau du simulateur puis le nouvel état est transporté au processeur correspondant sur le FPGA afin de continuer l'émulation. Ce type de transport est nommé *"transplant"*.

Les auteurs ont remarqué que le transplant est parfois coûteux, ce qui augmente l'overhead de la méthode. Ainsi, ils ont proposé le *micro-transplant* qui consiste à simplifier le transport des états au simulateur sur la machine hôte en réalisant la simulation sur le processeur intégré sur le FGPA (PowerPC dans le cas de Xilinx). La difficulté dans ce travail est de connaître les événements qui vont être implémentés sur FPGA, ceux simulés sur le processeur intégré dans le FPGA et ceux simulés sur la machine hôte.

En résumé de ces trois sections, nous pouvons dire :
Parmi les méthodes d'accélération déjà citées, nous avons accordé une grande attention à la méthode par échantillonnage pour les raisons suivantes :

1. Les méthodes d'accélération par abstraction du niveau de description, la simulation statistique et la simulation analytique manquent de précision quand les architectures simulées sont complexes.
2. L'accélération par réduction du jeu de données en entrée risque de donner une déviation du comportement réel de l'application.
3. L'émulation sur circuits reconfigurables est certes performante mais demande des plateformes spécifiques et coûteuses. Le passage d'une configuration architecture à l'autre nécessite un temps important.
4. L'échantillonnage est une méthode relativement facile à implémenter. Elle accélère la simulation en éliminant de la simulation les parties répétitives de l'application. De plus cette méthode peut être combinée avec d'autres méthodes.

Comme nous l'avons déjà mentionné dans ce mémoire, nous nous intéressons à l'accélération de la simulation pour les MPSoC par échantillonnage de l'application. Plus précisément, l'échantillonnage considéré est basé sur la méthodologie par profilage et analyse des phases. Ceci permet de prendre en compte le comportement réel de l'application. A notre connaissance il n'y a pas eu de travaux fondés sur l'échantillonnage dans le cas de multiprocesseurs en dehors de celui présenté dans la section suivante. L'échantillonnage a été adopté surtout dans le cas des systèmes *"Simultaneuous Multithreading"*(SMT) ou les architectures Multi-flots et rarement dans le cas des architectures multiprocesseurs.

2.4 Application de la méthode par échantillonnage dans les architectures Multi-flots

Deux projets méritent d'être détaillés dans ce cadre. Il s'agit du CoGS-Sim [40] et du Cophase [21, 18].

2.4.1 CoGS-Sim

Kihm et al. ont étendu la méthode PGSS-Sim, (voir section 2.3.2.4), dans le cas des architectures multi-flots SMT à deux voies dans un projet nommé CoGS-Sim pour *"Co-phase Guided Small Sample Simulation"*. La figure 2.6 montre le flot de fonctionnement de CoGS-Sim. Ce dernier consiste à combiner les phases s'exécutant en parallèle formant ainsi une cophase dans les intervalles d'1 million d'instructions qui subissent la simulation fonctionnelle. La seule différence avec PGSS-Sim est que la variation de la performance est vérifiée séparément au niveau de chaque thread en déterminant le nombre d'échantillons demandé pour chaque cophase. Co-GS-Sim réagit rapidement quand l'une des phases s'exécutant en parallèle change tandis que PGSS-Sim réagit seulement quand une nouvelle phase a lieu. CoGS-Sim est aussi basé sur la technique nommée MASS-Sim pour "Multithreaded Architecture Small-Sample Simulation" [41], où SMARTS a été étendu pour les cas SMT. Mais la différence est que le saut d'exécution d'un thread est réalisé en utilisant l'IPC dans la Cophase détectée au lieu d'utiliser l'IPC dans le dernier échantillon. L'objectif du CoGS-Sim est de réduire le nombre de combinaisons de phases qui augmente avec le nombre de threads

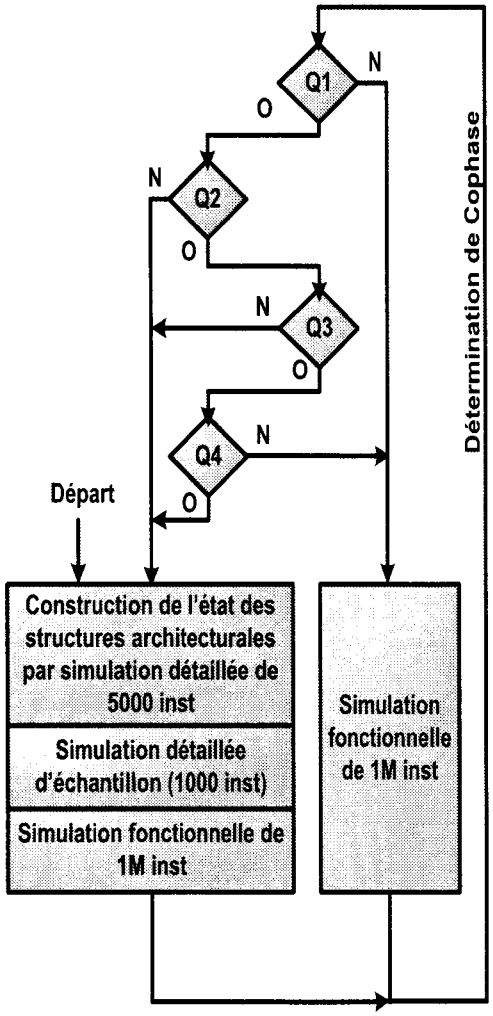


FIG. 2.6 – Le flot du fonctionnement de CoGS-Sim. La décision de prendre un sample est basée sur la quantité du temps depuis le dernier sample du Cophase et sur la variance de la performance du Cophase. La construction de l'état exact des structures architecturales est la même que celle de SMARTS et celle-ci est expliquée dans la suite de ce chapitre. Ce flot est présenté dans [40].

s'exécutant en parallèle. De même, quand la taille des applications est importante, le nombre de phases augmente au niveau de chaque thread ce qui va aussi augmenter le nombre de combinaisons de cophase. La méthode cophase expliquée dans la section suivante souffre de cet inconvénient.

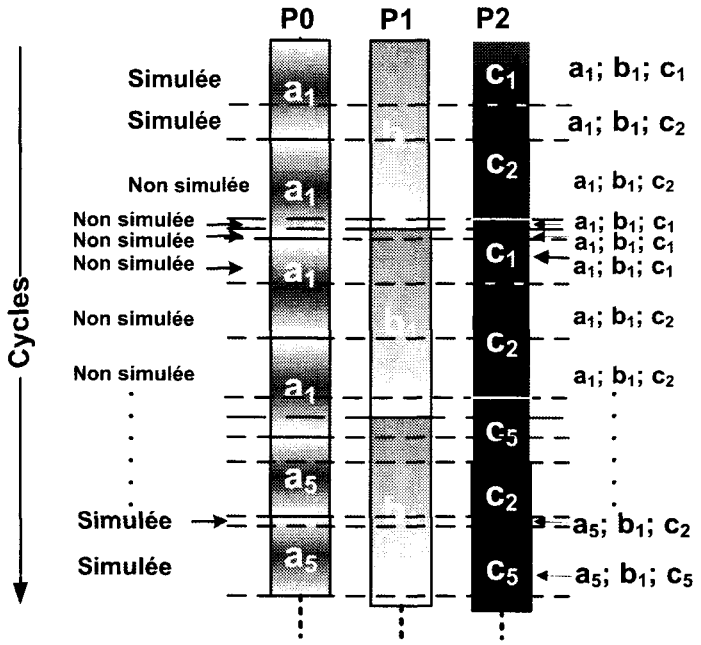


FIG. 2.7 – Méthode cophase appliquée à trois applications concurrentes : a, b et c. Chaque bloc correspond à une phase de l'application. Les cophases sont désignées par leur identificateurs.

2.4.2 Matrice de cophase

Cette méthode est proposée par Biesbroucky et al. [21, 18] pour une architecture multi-flots simultanés ou SMT. L'idée principale est que le comportement du système peut changer si un des flots parallèles change de phase. Ainsi, il faut garder l'historique de toutes les combinaisons de phases exécutées en parallèle afin de sauter les combinaisons qui se répètent.

SimPoint est d'abord utilisé afin de classer les intervalles de l'application de chaque thread. Une trace des phases (nommée aussi "phase-ID trace") est générée pour chaque thread. Cette méthode combine ensuite les phases s'exécutant en parallèle formant ainsi une *cophase* (voir figure 2.7). Les identificateurs des phases dans une cophase forment un identificateur unique de la cophase correspondante. Ce type de combinaison de phases est aussi appelé recouvrement des phases. Les cophases ayant le même identificateur sont considérées comme similaires, elles ont donc les mêmes performances. La figure 2.7 montre l'utilisation de la méthode cophase sur trois applications concurrentes. Comme on peut le voir, les cophases répétées ne sont pas simulées.

Une table nommée matrice de cophases est utilisée pour sauvegarder toutes les cophases exécutées durant la simulation. Ainsi lorsqu'une cophase est générée, une entrée est allouée dans la matrice contenant les statistiques correspondantes et en particulier l'IPC de chaque thread durant la simulation détaillée de la cophase correspondante. La table 2.1 montre la matrice contenant les cophases exécutées dans la figure 2.7. Le saut des cophases déjà simulées est réalisé en utilisant la simulation fonctionnelle pour faire avancer chaque thread. Les étapes suivantes résument le fonctionnement de cette méthode :

1. Utilisation de la matrice de cophases : les cophases exécutées sont représentées par un identificateur unique. Elles sont sauvegardées dans la matrice. Si la combinaison

Cophases	IPC du thread 0	IPC du thread 1	IPC du thread 2
$a_1; b_1; c_1$	ipc_a1	ipc_b1	ipc_c1
$a_1; b_1; c_2$	ipc_a2	ipc_b2	ipc_c2
$a_5; b_1; c_2$	ipc_a3	ipc_b3	ipc_c3
$a_5; b_1; c_5$	ipc_a4	ipc_b4	ipc_c4

TAB. 2.1 – Matrice contenant les cophases simulées dans la figure 2.7.

des phases suivantes dans les traces correspond à une cophase dans la matrice alors l'IPC de chaque thread est lu (voir tableau 2.1). Autrement une simulation détaillée est réalisée jusqu'à la fin d'une des phases. Les IPCs de chacun des threads dans la nouvelle cophase sont sauvegardés afin d'être réutilisés plus tard.

2. Détermination du nombre de cycles pour réaliser le saut de cophase : Pour chaque thread, le nombre d'instructions jusqu'à la nouvelle phase ainsi que son IPC lu de la matrice de cophases sont utilisés pour calculer le nombre de cycles nécessaires pour arriver à la nouvelle phase. Le thread ayant le plus petit nombre de cycles détermine de combien il faut avancer chaque thread pour réaliser le saut de cophase.
3. Réalisation du saut de cophase : Prendre le plus petit nombre de cycles de l'étape 2, et le multiplier par l'IPC de chaque thread de la cophase en question pour déterminer le nombre d'instructions pour faire avancer chaque thread. Reprendre à l'étape 1.

Pour déterminer la fin de l'échantillon les auteurs ont étudiés trois possibilités :

1. Quand la somme des instructions exécutées par tous les threads atteint 5 millions.
2. Quand un thread exécute exactement 5 millions d'instructions.
3. Quand chacun des threads a exécuté au moins 5 millions d'instructions.

Ainsi chaque échantillon de la simulation détaillée contient une ou plusieurs cophases. La taille des intervalles est de 10 millions d'instructions. De même dans le cadre de cette méthode, chacune des cophases est simulée plusieurs fois afin d'améliorer le niveau de précision. La nécessité de refaire la simulation d'une cophase est expliquée dans la section 2.7. Ainsi les cophases sont resimulées périodiquement. Les différentes périodes utilisées pour resimuler la même cophase sont 20 ou 100. A chaque fois que la cophase est simulée, ses statistiques sont ajoutées à celles collectées précédemment.

Michael Van Biebrouck et al. rapportent que les meilleurs résultats sont obtenus par cophase quand chaque thread exécute au moins 5 millions d'instructions avec une période de resimulation égale à 20.

De même la matrice statique de cophases est présentée dans [21]. Il s'agit de simuler toutes les combinaisons de phases possibles. Si le thread 1 contient a phases et le thread 2 contient b phases alors il y a $a * b$ combinaisons. Notons que chaque phase est représentée par un intervalle, nommé "intervalle représentatif", détecté par SimPoint. Ainsi, pour collecter les statistiques de chaque cophase, il faut simuler en parallèle les intervalles représentatifs des phases. Ce qui permet de générer statiquement la matrice de cophases. Cette matrice est utilisée pour faire avancer chaque thread grâce aux traces des identificateurs de phases. L'inconvénient de cette méthode de la matrice statique est que le nombre de combinaisons de phases est plus important que celui de la matrice dynamique de cophase. La matrice statique simule des combinaisons de phases qui n'existent pas dans l'exécution réelle.

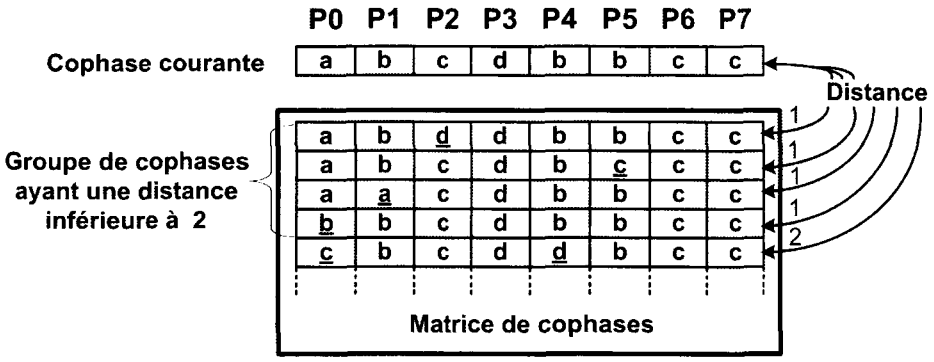


FIG. 2.8 – Synthèse des cophases. La distance avec la cophase courante est basée sur le nombre de phases différentes. Le seuil vaut 2.

La matrice statique de cophases est aussi utilisée par Biesbroucky et al. dans [20] où le problème traité est le changement dramatique du comportement du système SMT avec les points de démarrage (nommés starting points) des programmes exécutés par les threads. Le point de démarrage d'un programme est le point où commence la simulation. La taille des intervalles est de 5 millions d'instructions. Le point de démarrage de chaque programme est choisi au départ aléatoirement. A chaque fois que le nombre d'instructions exécutées atteint 10 milliards, le thread correspondant reprend l'exécution dés le début. Cela se répète jusqu'à ce qu'il n'y a plus de nouvelles cophases collectées. La matrice statique de cophases contenant toutes les combinaisons de phases possibles va guider la simulation.

Namkung et al [49] ont appliqué la méthode cophase dans le cas des systèmes MPSoC avec un nombre supérieur à 10 processeurs. Ils ont remarqué que l'accélération est trop faible, cela est due à l'augmentation du nombre de combinaisons de phases avec l'augmentation du nombre de threads. Ils ont proposé la synthèse de cophases pour réduire le nombre de combinaisons de phases. Cette synthèse est basée sur l'idée que la performance caractéristique d'une phase n'est pas toujours dépendante de toutes les autres phases qui s'exécutent simultanément. Autrement dit, pour synthétiser une cophase courante d'un groupe de cophases déjà simulées les étapes suivantes sont réalisées :

1. Si la cophase courante n'existe pas dans la matrice alors les étapes 2 et 3 sont réalisées sinon la cophase courante est non simulée.

2. Toutes les cophases, dans la matrice, dont la distance avec la cophase courante est plus petite que le seuil donné sont regroupées.

3. Chaque phase dans la cophase courante est vérifiée si elle se trouve, au niveau du même thread, dans l'une des cophases regroupées. Si c'est le cas alors la cophase courante est non simulée et ses IPCs sont la moyenne des IPCs de cophases regroupées. Sinon la cophase courante subit une simulation détaillée.

La figure 2.8 montre la synthèse des cophases. Dans la matrice de cophases les cophases dont les distances avec la cophase courante sont plus petites que le seuil 2 sont regroupées. Comme on peut le voir au niveau du chaque processeur, la phase dans la cophase courante se trouve dans l'une des cophases regroupées. Ainsi la phase courante est non simulée.

2.5 Construction de l'image du système pour l'échantillonnage

Dans la méthode d'accélération de la simulation par échantillonnage, l'une des difficultés est l'obtention de manière efficace de l'état exact du système au point du démarrage de la simulation détaillée (nommé aussi *sampling startup* [19]) de chaque échantillon soumis à la simulation détaillée. Ainsi avant chaque simulation d'un échantillon il est nécessaire de s'assurer que l'état du système est correct. Ce dernier est composé de deux sous-états :

1. L'image mémoire nommée SSI pour *"Sample Starting Image"*.
2. L'état des composants micro-architecturaux nommé *"Sample Warming-up"*.

2.5.1 Techniques de construction de *"Sample Starting Image"*(SSI)

Le SSI, appelé aussi l'état de l'architecture dans [62], concerne les valeurs de données qui se trouvent dans les registres et dans la mémoire. Ces données représentent le contexte de l'application. Elles sont indépendantes de la configuration architecturale. Nous citons dans la suite les techniques utilisées pour construire le SSI au point de démarrage de chaque échantillon.

1. La simulation fonctionnelle qui fait avancer la simulation vers un point donné du flot d'instructions de l'application en utilisant un modèle d'exécution simple en négligeant les détails de l'architecture du système. Pendant cette simulation fonctionnelle, le simulateur lit les instructions et les exécute sans prendre en compte les détails de l'architecture. Ainsi les détails des mémoires caches, des prédicteurs de branchements, du réseau d'interconnexion par exemple ne sont pas considérés. Ceci permet d'avancer rapidement dans la simulation mais ne permet pas d'estimer le temps d'exécution.
   SMARTS [64] adopte cette technique pour avancer la simulation entre les échantillons pris périodiquement. La technique de simulation fonctionnelle est nommée aussi *"fast-forwarding"*.
2. Szwed et al. [59] proposent une deuxième technique dite *"direct execution"* pour faire avancer la simulation entre les échantillons par une exécution native sur le processeur hôte. Cette technique utilise les points de contrôle ou les checkpointing (voir paragraphe suivant) pour fournir l'état de l'application au simulateur. Le simulateur réalise la simulation détaillée d'échantillons en utilisant le checkpoint. Quand la fin d'échantillons est atteinte, l'exécution native sur la machine hôte entre de nouveau en jeu pour faire avancer l'exécution jusqu'à l'échantillon suivant. Cette technique exige que la simulation soit effectuée sur un processeur ayant le même jeu d'instructions que le programme simulé.
3. Le checkpointing permet de sauvegarder un point de contrôle ou "checkpoint" représentant le SSI au point de démarrage d'échantillon. Avant de démarrer la simulation, le checkpoint est chargé. Le checkpoint contient les valeurs sauvegardées dans les registres et la mémoire au point correspondant. Cette technique a été adoptée dans [63] pour construise le SSI.

Il est à noter que la simulation fonctionnelle est relativement simple à implémenter sur les simulateurs. L'inconvénient majeur de la simulation fonctionnelle est qu'elle impose un ordre dans l'exécution des échantillons. Ainsi, il n'est pas possible de commencer un échantillon avant ceux qui le précédent. Par ailleurs, le temps de simulation dépend de la taille du

code entre les deux échantillons à simuler. Ainsi la simulation fonctionnelle limite la vitesse de la méthode d'échantillonnage [65].

L'utilisation de checkpoint élimine l'overhead de la simulation fonctionnelle [62, 65], mais risque d'être prohibitif et coûteux en termes d'espace de stockage mémoire quand le nombre d'échantillons à simuler est important. En plus, quand la taille du checkpoint est importante la méthode affecte le temps de simulation dû au chargement du checkpoint du disque vers la mémoire du système hôte. Pour cela, Biesbrouck et al. [19, 18] proposent deux techniques pour réduire les tailles des checkpoints pour le SSI.

La première technique, nommée TMI pour *"Touched Memory Image"*, consiste à sauvegarder les blocs mémoires, qui seront lus en cours d'échantillon à simuler ainsi que leurs adresses respectives. Il y a un TMI pour chaque échantillon. Durant la simulation, au point de démarrage d'échantillon, le TMI correspondant est chargé du disque et les blocs mémoires correspondants sont écrits dans la mémoire dans les adresses respectives. Les zones mémoire contenant des valeurs nulles consécutives ne sont pas prises en compte dans le TMI. Cette technique offre un gain d'espace de stockage en termes d'adresses mémoires puisque seule une adresse est sauvegardée pour une grande région de données consécutives. Nous avons utilisé une technique proche de TMI pour construire les checkpoints de SSI dans le chapitre 6. Une optimisation de TMI nommée RTMI pour *"Reduced Touched Memory Image"* est aussi proposée dans [19, 18] qui consiste à sauvegarder uniquement les blocs mémoires qui vont être lus avant une écriture.

La deuxième technique, nommée LVS pour *"Load Value Sequence"*, consiste à créer une trace des valeurs lues de la mémoire (*"load values"*) durant la simulation d'échantillon. La génération des traces de LVS peut être faite par une simulation fonctionnelle. Durant la simulation d'échantillon, la trace correspondante des valeurs est lue de façon concurrente avec la simulation afin de fournir les valeurs aux instructions mémoire *"load"* exécutées. Contrairement à TMI, LVS ne nécessite pas la sauvegarde des adresses mémoires. Néanmoins, les programmes réalisent généralement plusieurs lectures mémoire de la même adresse ainsi que des lectures de valeurs nulles. Ceci augmente la taille de LVS sans affecter celle de TMI. Une optimisation de LVS, nommée RLVS pour *"Reduced Load Value Sequence"*, consiste à utiliser un bit pour indiquer si la lecture doit être faite de trace ou de la mémoire. Ainsi, RLVS ne contient pas les valeurs des données qui ont été précédemment référencées par une lecture ou par une écriture. De même, les données nulles ne sont pas gardées dans la trace.

2.5.2 Techniques pour le *"Sample Warming-up"*

Le *Sample Warming-up* consiste à préparer l'état de la micro-architecture [62] : le contenu des composants du pipeline et de la hiérarchie des mémoires cache. Ces données dépendent de la configuration architecturale. Il y a différentes techniques pour réaliser le *Sample Warming-up*.

La première technique consiste à faire un préchauffement par **N** instructions avant chaque échantillon. Les **N** instructions additionnelles subissent la simulation détaillée afin de préparer les structures architecturales. A titre d'exemple, SMARTS [64] adopte un préchauffement de 2000 ou 4000 instructions avant chaque échantillon. Durant le préchauffement, les statistiques ne sont pas collectées. Avec SMARTS, la taille des échantillons est faible et vaut 1000 instructions. Le préchauffement est réalisé par un nombre fixe d'instructions. De plus, une simulation fonctionnelle entre les échantillons est réalisée pour les mémoires cache et les prédicteurs de branchement.

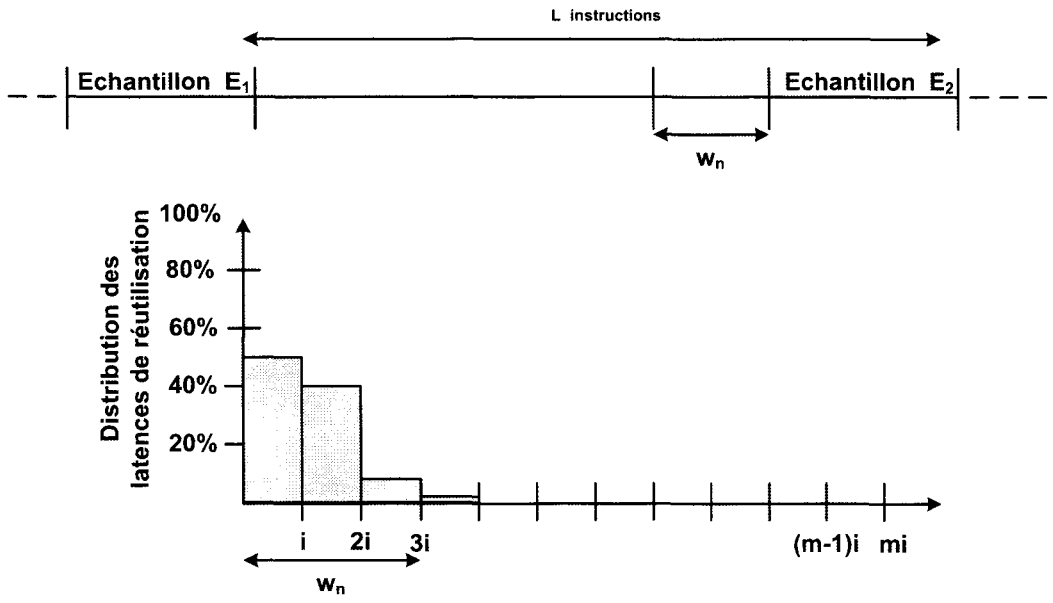


FIG. 2.9 – MRRL est appliquée depuis la fin d'échantillon E_1 jusqu'à la fin d'échantillon E_2. L correspondant au nombre d'instructions entre la fin du E_1 et la fin du E_2 est décomposée en m intervalles de *i* instructions. Le graphe montre la distribution des latences de réutilisation rencontrées dans les L instructions.

A l'instar de cette technique qui adopte un nombre fixe d'instructions pour effectuer le préchauffement, une autre technique, appelée MRRL pour *Memory Reference Reuse Latency* [33, 36], supporte un nombre variable d'instructions pour réaliser le préchauffement en se basant sur la notion de la latence de la réutilisation de la référence mémoire. Cette latence est définie comme étant le nombre d'instructions dynamiques entre deux références mémoire consécutives pour la même location mémoire. Pour calculer le nombre d'instructions de préchauffement pour un échantillon donné, MRRL calcule d'abord la distribution des latences de réutilisation pour l'ensemble des instructions depuis la fin d'échantillon précédent jusqu' à la fin d'échantillon courant. Cette distribution donne une idée sur le comportement de la localité temporelle. MRRL détermine par la suite le nombre d'instructions de préchauffement (nommé w_N), pour l'échantillon en question de façon à couvrir N% de la distribution des latences de réutilisation. Autrement dit, en prenant un intervalle de préchauffement de w_N instructions, on garantit que N% des latences de références mémoires sont couverts. La valeur de N% est fixée à 99% dans les travaux proposés. La figure 2.9 montre la distribution des latences de réutilisation entre la fin d'échantillon E_1 et la fin d'échantillon E_2. Comme on peut le voir, 50% de ces latences sont compris entre 1 et i instructions. De même 40% sont compris entre i et $2i$ instructions. Il faut que w_n soit égale à $3i$ instructions pour garantir que 99% de ces latences soient couverts. Notons que MRRL examine l'ensemble des instructions entre l'échantillon précédent et la fin d'échantillon courant pour déterminer le nombre d'instructions de préchauffement. Une optimisation de MRRL, nommée BLRL pour *"Boundary Line Reuse Latency"* [27]. Cette dernière consiste à analyser seulement l'échantillon en question pour déterminer le nombre d'instructions nécessaires pour effectuer le prechauffement.

La deuxième technique est le checkpointing adopté par plusieurs travaux pour construire

l'état des structures architecturales. Dans [63] le checkpointing est utilisé pour construire l'état de la micro-architecture pour chaque échantillon. Lauterbach [46] propose de construire les checkpoints en sauvegardant les étiquettes des adresses mémoire cache au début de chaque échantillon. Le checkpoint est collecté par une simulation fonctionnelle. Cette technique est similaire à la technique nommée MHS pour "Memory Hierrarchy State" [19] que nous avons utilisée dans le travail qui sera présenté dans le chapitre 6. MHS applique le chekpointing sur les contenus des mémoires cache qui donc ne nécessitent pas de préchauffement au début de la simulation des échantillons. La DSE exige que différentes configurations des caches soient simulées, mais le MHS permet de collecter les checkpoints une fois pour toute pour chacune des tailles de bloc et pour chacune des politiques d'emplacement. Ces checkpoints sont collectés d'un cache de référence et ils sont utilisés durant la phase de DSE pour construire des caches des tailles plus petites.

2.5.3 Avantage de la méthode par checkpoint

En cours de cette thèse nous nous sommes intéressés à la technique de checkpointing pour construire l'état du système et pour l'appliquer aux architectures multiprocesseurs. Nous avons choisi le checkpointing car cette technique élimine l'overhead nécessaire pour faire avancer la simulation d'un échantillon à l'autre. Ceci améliore les performances de la méthode d'accélération choisie. Les problèmes majeurs à résoudre pour le checkpoiting dans les systèmes multiprocesseurs sont :

1. Le nombre de checkpoints est plus important dans le cas de multiprocesseurs que dans le cas des systèmes monoprocesseurs. En effet, ce nombre augmente avec le nombre de processeurs à simuler.
2. L'impossibilité de savoir à priori où commence chaque échantillon, ce qui provoque une surestimation du nombre de checkpoints à collecter.

La méthode du checkpointing a été adaptée pour les systèmes SMT dans le cadre de Cophase [21, 20]. Les deux problèmes énumérés précédemment ont été résolus en utilisant la matrice statique de cophases (voir section 2.4.2). Elle consiste à générer statiquement toutes les combinaisons possibles de phases. Notons que dans cette matrice statique, il y a des cophases qui sont simulées mais ne se produisent jamais durant l'exécution des applications. Dans le chapitre 6, nous étudierons l'adaptation du checkpoint dans le cadre des méthodes d'accélération que nous proposons. Cette adaptation réduit le nombre de checkpoints grâce à une technique qui détecte à l'avance les points où les échantillons vont démarrer.

2.6 Outils de simulation

Il existe plusieurs environnements de simulation d'architecture multiprocesseurs open source. Dans la suite, nous présentons trois d'entre eux : SoCLib [55], UniSim [15] et MPARM [17].

1. **SoCLib** est un environnement de simulation des systèmes sur puce, basé sur le langage de simulation SYSTEMC. Il a été initié par l'équipe ASIM du laboratoire LIP6. De nombreuses équipes (LIP, LIRMM, TIMA, ENST, etc) l'utilisent et contribuent à son développement. SoCLib est une bibliothèque de composants pouvant être simulés au niveau CABA et au niveau TLM. Les composants sont synchrones, c'est à dire

qu'ils ont tous un signal d'horloge qui cadence leur fonctionnement. Différents composants sont disponibles dans SoCLIB tels que, des processeurs (MIPS R3000, ARM7 et PowerPC) avec des caches, des mémoires DRAM, des accélérateurs matériels (DMA, etc), des réseaux d'interconnexion (crossbar et des NOC comme SPIN et DSPIN) et des périphériques (affichage vidéo et Timer). SoCLib supporte un système d'exploitation embarqué (mutek). Ces composants une fois rassemblés forment un SOC complet.

2. **Unisim** est un environnement de simulation modulaire implémenté en SystemC. Il permet une simulation hybride TLM/CABA. Il offre plusieurs services en affectant à chacun d'eux un API (Application Programming interface). La modification et l'ajout d'un service sont assez simples dans le cadre d'Unisim. Parmi ces services, nous citons : différents modèles d'énergie, le checkpointing, le debugging et l'accélération de simulation par échantillonnage. Ce dernier point à été réalisé par SMARTS [64]. En plus, Unisim est conçu pour créer des simulateurs hétérogènes en enveloppant des simulateurs qui existent déjà dans des modules d'Unisim, ce qui permet une interaction entre ces simulateurs et les modules existants d'Unisim. Unisim offre une possibilité d'interopérabilité entre simulateurs existants ce qui, d'une part, réduit le temps de développement et d'autre part, offre un environnement de comparaison de ces simulateurs. La version courante du projet Unisim contient :
   - Au niveau TLM : des simulateurs pour des processeurs PowerPC 750, ARM V3 et ARMv5TE.
   - Au niveau CABA : des simulateurs monoprocesseurs comme PowerPC405 et ARMv5TE et des simulateurs multicore respectivement pour PowerPC405 et ARM9.

   Plusieurs efforts de recherche continuent pour améliorer de plus en plus Unisim. Par exemple, une bibliothèque est en cours de développement et permet de scanner automatiquement le DSE afin de maintenir le classement des meilleures architectures [15]. Un paramètre API est affecté à ce service de scanner de DSE.
3. **MPARM** est notre plateforme de simulation MPSoC détaillée dans le chapitre 3. Depuis plusieurs années les processeurs ARM occupent une place importante dans le marché des processeurs embarqués [53], environ 45% du marché en 2002. Ainsi, nous avons choisi MPARM car en 2005 où cette thèse a été lancée, il était l'unique simulateur complet et académique qui modélise les processeurs ARM. Rappelons que nos méthodes d'accélération proposées en cours de cette thèse sont indépendantes de tout outil de simulation.

2.7 Positionnement par rapport à l'état de l'art

Comme nous venons de le voir, la méthode cophase est basée sur l'échantillonnage des recouvrements de phases et, par conséquent, les statistiques de chaque recouvrement sont pris plusieurs fois pour atteindre un niveau de précision acceptable.

En effet, quand le nombre de processeurs est important et surtout où les applications s'exécutant en parallèle ont des comportements différents, "applications hétérogènes", les recouvrements de phases deviennent courts dans le cas de cophase. Ce qui augmente, pour un même recouvrement de phases (cophase), la variation de scénarios de phases conjoints. Par exemple, la figure 2.10.a montre que le recouvrement de phases est court et la variation de scénarios conjoints est trop importante pour les trois occurrences consécutives de cophase $a_1; b_1; c_1$. A cause de cette importante variation, la prise en compte de la première

occurrence de chaque recouvrement de phases comme représentant des autre occurrences de ce recouvrement, est une tâche difficile.

Par exemple, la cophase $a_1; b_1; c_1$ apparaît quatre fois dans la figure 2.10.a. Toutes ces occurrences représentent différents scénarios conjoints de phases et aussi différents nombres d'instructions simulées. De même, on ne peut pas supposer que l'exécution d'une phase soit homogène ou que chaque partie de la phase puisse être représentative de la phase entière. Généralement, SimPoint détecte la similarité des phases en se basant sur la distribution de leur BBV, mais il ne peut pas garantir que le comportement dans la phase est homogène. Un cas particulier de l'homogénéité d'une phase est associé à de grandes boucles régulières. C'est pour cela que pour cophase autant il y a des différents scénarios conjoints autant il faut prendre de échantillons pour atteindre un niveau acceptable de précision. La variation de scénarios conjoints ne peut pas être déterminée à priori, par conséquent le nombre d'échantillons pris par cophase peut être surestimé.

Cependant, dans le cours de nos travaux, nous considérons que la phase est une unité de travail atomique. La phase ne peut pas être divisée. De plus, le temps d'exécution d'une phase dépend des phases exécutées précédemment. Cette dépendance entre les phases dépend de la configuration architecturale, elle ne peut pas être prise en compte durant l'étape du profilage statique de l'application. Pour des échantillons de grande taille, cette dépendance peut avoir une faible influence sur la précision de la simulation. En effet, des échantillons de petite taille sont préférés pour obtenir une importante accélération de simulation pour explorer un grand DSE. Ces difficultés apparaissent surtout dans les systèmes MPSoC à cause du niveau élevé de l'hétérogénéité des applications s'exécutant en parallèle. Dans ces conditions, nous proposons en cours de cette thèse une méthode nommée AS pour *"Adaptive Sampling"* (voir chapitre 3) qui adopte deux mécanismes : 1) La discrétisation des découvrements de phases en utilisant des barrières de simulation qui définissent le début et la fin de chaque recouvrement. 2) La détection des répétitions dans la simulation en générant des recouvrements de phases comprenant une séquence de phases de chaque processeur. La figure 2.10.b montre un exemple de la méthode AS. L'un des avantages de cette méthode est qu'un échantillon unique de chaque recouvrement de phases généré est suffisant pour obtenir un niveau élevé de précision, contrairement aux deux méthodes CoGS-Sim et Cophase.

Comme nous l'avons déjà montré dans l'état de l'art, SimPoint supporte les intervalles de taille variable pour détecter les phases réelles de l'application. Cette technique est valable dans le cas des monoprocesseurs mais dans le cas des multiproceseurs, les applications ont des comportements ainsi que des phases qui se distinguent de ceux du cas monoprocesseur [21]. Afin de détecter dynamiquement les phases de chaque application concurrente et afin de réduire le nombre de combinaisons de phases, nous avons proposé dans le cadre de cette thèse une deuxième méthode nommée MGS pour *"Multi-Granularity Sampling"* (voir chapitre 5). Cette méthode consiste à analyser chaque application pour différentes tailles des intervalles , nommées aussi "granularités", qui correspondent aux tailles des phases réelles des applications. Durant la simulation, à chaque point de discrétisation, la granularité de phase est associée dynamiquement pour chaque application de telle sorte que chaque recouvrement de phases comprend une phase par processeur (voir figure 2.10.c). Ainsi les granularité des phases dans les recouvrements dépendent de l'exécution de l'application correspondante, ces granularités peuvent donc être différentes. Cette stratégie d'une phase dans chaque recouvrement garantit la détection de phases des applications dans les cas de multiprocesseurs. Comme pour AS, un échantillon unique de chaque recouvrement de phases est

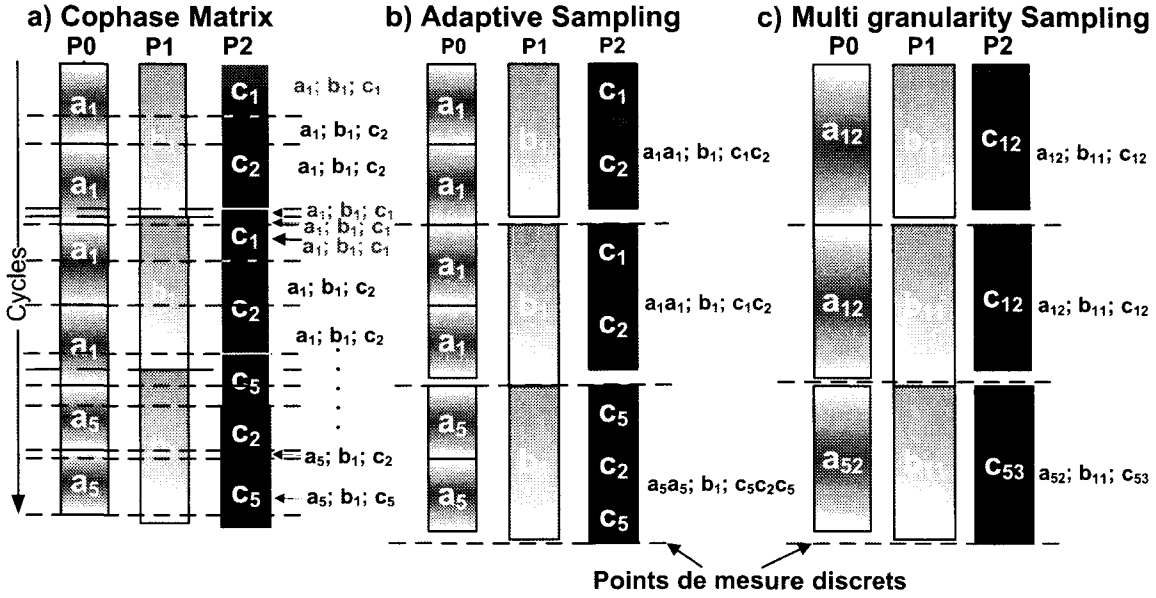


FIG. 2.10 – Différentes méthodes d'échantillonnage pour des systèmes multiprocesseurs. Chaque bloc correspond à une phase de l'application.

suffisant pour obtenir un niveau élevé de précision dans le cadre MGS.

2.8 Conclusion

Dans ce chapitre nous avons présenté les différentes méthodes qui accélèrent l'évaluation de la performance des systèmes en réduisant le temps de simulation. Nous avons expliqué en détails les méthodes auxquelles nous nous intéressons dans le cadre de cette thèse à savoir : la méthode par échantillonnage et en particulier les méthodologies basées sur Simpoint. En effet, les méthodes par échantillonnage nécessitent des techniques supplémentaires pour construire le contexte du programme et l'état des structures architecturales au début de chaque simulation détaillée. Ces techniques, en particulier le checkpointing que nous étudions dans le cadre de cette thèse, sont aussi analysées dans ce chapitre. Dans le chapitre suivant, nous présenterons de façon plus détaillée notre première méthode d'accélération de la simulation qui applique l'échantillonnage dans le cas des systèmes multiprocesseurs MPSoC.

Chapitre 3

Accélération de la Simulation par Échantillonnage Adaptatif des applications

3.1 Introduction

Comme cela a été précisé dans les 2 chapitres précédents, cette thèse a pour objectif de développer des techniques performantes pour accélérer la simulation dans les systèmes MP-SoC. Nous nous intéressons en particulier à l'accélération basée sur l'échantillonnage d'applications concurrentes. Chacune de ces applications est décomposée en phases et l'un des problèmes posé concerne la détermination des phases parallèles qui s'exécutent simultanément par les différents processeurs. En effet la disposition de phases parallèles entre les processeurs peut changer complètement d'une configuration architecturale à l'autre. Notre méthode d'échantillonnage adaptatif (notée AS pour *Adaptive Sampling* [3, 4, 5, 1]) génère des échantillons en recouvrant parfaitement les phases parallèles des applications. Ce type de recouvrement est appelé "recouvrement de phases" dans la suite de cette thèse. Un seul échantillon de chaque recouvrement est alors suffisant pour obtenir une estimation précise des performances des applications. Dans chaque recouvrement de phases généré dynamiquement, le groupe de phases, au niveau de chacun des processeurs, est nommé *séquence de phases* (ou *phase string*).

Ce chapitre présente la méthode AS qui réalise l'échantillonnage de l'application pour accélérer la simulation. Chaque échantillon correspondant à une sorte de recouvrement parfait de phases nommé CS (pour *Clusters of Strings*) comprend une séquence de phases par processeur. Comme nous le verrons de façon détaillée dans la suite de ce chapitre, un temps d'attente entre les processeurs est injecté pour obtenir un recouvrement parfait entre les CSs. La performance est estimée en simulant une seule occurrence de chaque CS. Ainsi, durant la simulation, si un CS n'a pas été rencontré, la simulation détaillée de ce dernier est réalisée. AS offre les avantages suivants :

- La formation de séquences de phases est effectuée dynamiquement en fonction du comportement réel des applications concurrentes et automatiquement sans intervention de l'utilisateur.
- Les recouvrements portant les mêmes séquences de phases s'exécutant en parallèle ont des temps d'exécution identiques. Ce qui allège le besoin de plusieurs échantillons du même recouvrement de séquences de phases.

3.2 Échantillonnage Adaptatif AS

Notre méthode d'échantillonnage adaptatif est capable d'accélérer la simulation avec une précision raisonnable dans l'estimation de la performance sans changer régulièrement le pourcentage d'échantillons de la simulation détaillée. En effet, ce pourcentage est déterminé dynamiquement en fonction du nombre de séquences de phases rencontrées durant la simulation et non pas d'un pourcentage d'échantillons statique défini à priori. En réduisant le temps associé à l'évaluation de chaque configuration architecturale de l'espace d'exploration, AS réduit la phase de mise sur le marché du système embarqué. L'algorithme 1 résume le fonctionnement en deux étapes de la méthode AS que nous détaillerons par la suite.

Première étape : Générer séparément d'une trace de phases pour chaque application.
Deuxième étape : Pour chaque CS simulé dans la CST, vérifier si une similarité existe avec les phases suivantes dans les traces de phases.
Si (une similarité existe) **Alors**

- Un saut dans l'exécution est réalisé.
- La performance est estimée en se basant sur le CS déjà simulé.

Sinon

- La simulation détaillée continue jusqu'à la génération dynamique d'un nouveau CS.
- Allouer une entrée dans la CST pour le CS généré puis sauvegarder les statistiques correspondantes.

Fin Si
Répéter la deuxième étape jusqu'à la terminaison d'une trace de phases.

Algorithme 1: Échantillonnage de phases adaptatif.

3.3 Première étape : génération d'une trace de phases par programme

Dans cette étape, une trace de phases (nommée *phase-ID trace*) est générée séparément pour chacune des applications parallèles. La trace de phases pour une application donnée est indépendante à la fois des phases des autres applications et de la configuration architecturale sur laquelle se fait l'exécution des applications concurrentes. Elle ne dépend que de l'application elle-même et de son jeu de données d'entrée. Ainsi, cette étape est faite une seule fois pour tout l'espace de configurations architecturales à explorer. Nous utilisons l'outil de classification des phases SimPoint [57] afin de générer la trace (voir chapitre 2). Néanmoins, tout autre outil de classification des phases peut être utilisé [37]. Cette étape est indépendante de la configuration architecturale et elle est réalisée une seule fois pour toute. Ainsi, une trace de phases des différentes applications concurrentes peut être obtenue en quelques minutes grâce à une simulation fonctionnelle.

Le tableau 3.1 montre un exemple de deux traces de phases générées durant cette étape pour deux applications exécutées sur un système MPSoC de deux processeurs P0 et P1. Dans cette figure, dans l'application exécutée par P0, le premier et le cinquième intervalles sont considérés similaires par SimPoint et ils ont le même identificateur de phase *'a'*. Dans ce tableau, la taille des intervalles d'instructions considérés pour la génération des phases est de 50K instructions.

3.4 Deuxième étape : génération et utilisation de CSs

Dans cette étape, les traces de phases générées dans la première étape seront utilisées. Pour chaque processeur de l'architecture MPSoC, une ou plusieurs phases consécutives sont

Intervalle (en K instructions)	P0 phases	P1 phases
0-50	a	x
50-100	b	y
100-150	c	z
150-200	d	w
200-250	a	w
250-300	b	x
300-350	f	y
...	...	z

TAB. 3.1 – Exemple de deux traces d'identificateurs de phase pour deux applications exécutées respectivement par les processeurs *P0* et *P1*. Les deux traces sont générées par profilage des applications. La signification des couleurs sera expliquée par la suite.

combinées pour former une séquence de phases. Ainsi, si on note P le nombre de processeurs du MPSoC, il y aura P séquences de phases parallèles. Ces P séquences représentent l'exécution concurrente des applications dans un intervalle de temps. Le nombre de phases, dans la séquence, est déterminé dynamiquement. Comme on le verra dans le paragraphe suivant, l'ensemble des P séquences forme un cluster de séquences (CS) par l'utilisation de barrières de simulation. Nous avons opté pour les barrières de simulation pour synchroniser dynamiquement les processeurs à la fin de leurs intervalles permettant ainsi la discrétisation de l'exécution continue des applications.

Dans le tableau 3.1, nous avons trois CSs. Dans le premier CS, nous avons deux séquences de phases a, b et x, y, z. Dès qu'une occurrence d'un nouveau CS est trouvée, une entrée dans la table de CSs, notée CST pour "**Cluster of String Table**", est allouée. L'idée générale de notre approche est fondée sur le fait que les CSs contenant les mêmes séquences de phases ont le même comportement sur la plate-forme matérielle. De cette façon pour connaître les performances des CS identiques (en termes de temps d'exécution et de consommation de puissance), il suffit de simuler une seule occurrence. L'entrée dans la CST contient les informations pour toutes les occurrences de la même CS. Cette deuxième étape est détaillée dans la section suivante.

3.4.1 Utilisation de la barrière de simulation

Durant la simulation, lorsqu'un processeur termine l'exécution d'un intervalle, le simulateur estime le nombre maximum de cycles restants (ΔC) de telle sorte que tous les autres processeurs terminent leurs intervalles respectifs.

Comme cela est montré dans l'algorithme 2, le ΔC de chaque processeur dépend du nombre d'instructions par cycle (IPC) depuis le début du cluster jusqu'à ce point de la simulation. Les ΔCs peuvent être différents d'un processeur à l'autre à cause de la différence de leur IPCs. En effet les processeurs exécutent des applications avec des instructions différentes, ce qui influe sur leur IPC respectif. Le pseudo-code de cette technique de synchronisation est présenté dans l'algorithme 2. Dans cet algorithme, *Interval* correspond à la taille de l'intervalle en nombre d'instructions, I_{proc} et IPC_{proc} correspondent respectivement au nombre d'instructions exécutées et à l'IPC du processeur *Proc* depuis le début de l'intervalle. Ces statistiques sont données par le simulateur au cours de la simulation et servent à estimer

```
ΔC = 0
Pour Proc = 1 to P faire
  | Si ( Proc ≠ Mon_Proc) Alors
  |   | ΔC = max(ΔC,(Intervalle-I_proc)/IPC_proc))
  | Fin Si
Fin Pour
Si ( ΔC/C ≤ TWSB) Alors
  | Déclenchement de barrière de simulation pour tous les procs
Fin Si
```

Algorithme 2: Algorithme exécuté par chacun des processeurs pour la génération de barrière de simulation. *Mon_Proc* correspond au processeur qui a fini son intervalle. *Intervalle* est la taille de l'intervalle en instructions, I_{proc} et IPC_{proc} sont respectivement le nombre d'instructions et l'IPC dans le processeur *Proc* depuis le début de l'intervalle.

le nombre de cycles nécessaire à chaque processeur pour terminer son intervalle actuel. Si le rapport du plus grand ΔC sur le nombre de cycles (C) total, depuis le commencement du CS, est inférieur à un seuil donné, le simulateur génère une barrière de simulation et force tous les processeurs à arrêter la simulation à la fin de l'intervalle. Ce seuil est appelé TWSB[3]. Lorsque tous les processeurs ont rejoint la barrière de simulation, les séquences de phases exécutées en parallèle sont combinées pour former un nouveau CS dont les statistiques seront stockées dans la CST.

Autrement, si ΔC/C est supérieur au seuil, le simulateur continue l'exécution des instructions du processeur en question sur l'intervalle suivant. Notons, qu'à cause de l'injection des temps d'attente pour certains processeurs avant la barrière de simulation, la disposition de phases exécutées en parallèle peut être différente de la situation réelle, sans barrière de simulation. En effet, le décalage dans la disposition des phases peut provoquer une erreur d'estimation supplémentaire à celle provoquée par la classification de phases réalisée dans la première étape de l'algorithme. Notons que le nombre de phases exécutées par chaque processeur dans un CS peut être différent d'un processeur à l'autre du fait que les IPCs dans les processeurs s'exécutant en parallèle ne sont pas similaires.

La figure 3.1 montre que les processeurs testent, à la fin de chaque intervalle, la condition pour la génération de la barrière de simulation. A la fin de l'intervalle *a*, P0 teste la condition de la barrière et trouve qu'elle n'est pas vérifiée. P0 continue la simulation sur l'intervalle (*b*). P1 à la fin de ces deux intervalles *x* et *y* décide de continuer, la condition de la barrière n'est pas vérifiée. A la fin de l'intervalle *b* la condition de la barrière est vérifiée, P0 décide alors d'attendre P1 et une barrière de simulation est déclenchée. Ainsi un nouveau CS (*a,b-x,y,z*) est généré.

La probabilité de générer une barrière de simulation est proportionnelle à la valeur du TWSB. Le nombre de CSs générés augmente ainsi avec la valeur du TWSB. Quand le TWSB est relativement grand, les CSs générés sont relativement nombreux alors il y a une grande probabilité pour rencontrer des CSs qui se répètent. Cela augmente le facteur d'accélération. Dans ces conditions, les temps morts injectés avant les barrières de simulation seront

[3]**Threshold Waiting at Simulation Barrier**.

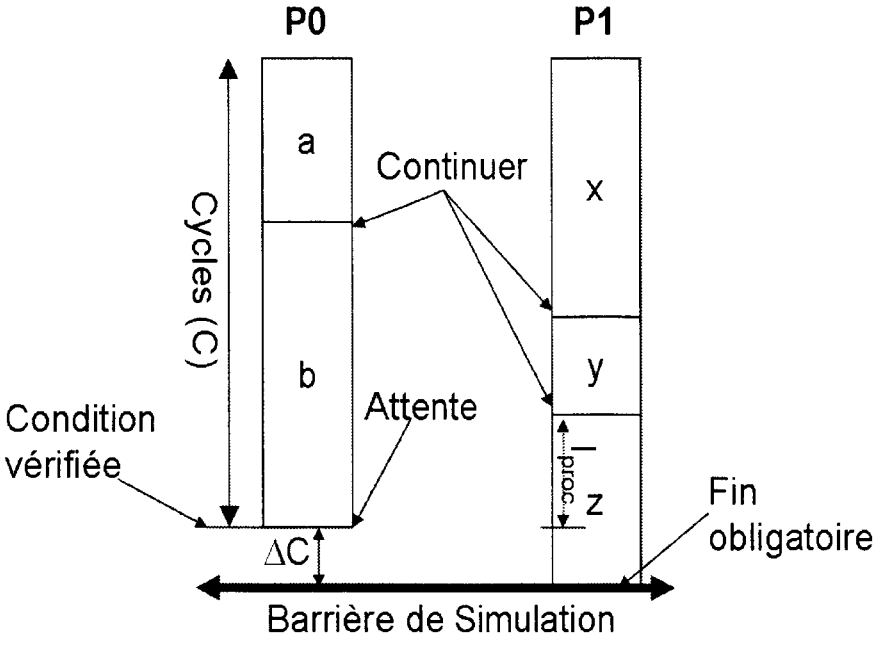


FIG. 3.1 – Barrière de simulation pour la génération des CSs.

importants, ce qui provoque aussi une erreur d'estimation relativement grande. Nous déduisons que le TWSB contrôle le compromis entre le facteur d'accélération et la précision de la performance estimée. Les concepteurs des systèmes intéressés à accélérer la simulation tentent d'augmenter le TWSB tandis que ceux qui privilégient la précision de la performance estimée tentent de diminuer le TWSB.

Les CSs sont séparés par des barrières de simulation et chaque séquence de phases comprend les phases qui sont exécutées par le processeur correspondant. La combinaison des identificateurs de phases qui s'exécutent en parallèle représente un identificateur unique du CS. Le tableau 3.2 montre les possibles CSs composés de phases des deux applications données dans le tableau 3.1. Le premier CS généré et noté par a,b-x,y,z contient deux séquences de phases : a,b et x,y,z respectivement pour P0 et P1. On considère que les CSs contenant les mêmes séquences de phases ont la même performance. La CST est mise à jour dynamiquement ; une fois qu'un nouveau CS a été simulé, une nouvelle entrée est allouée dans la CST et ses statistiques de performances, nombre de cycles, nombre d'instructions, consommation d'énergie, etc., sont sauvegardés.

3.4.2 Utilisation de la CST dans l'accélération de la simulation

Comme cela est montré dans l'algorithme 1, à la fin de chaque CS, si une similarité est détectée entre un CS de la CST et les phases suivantes dans les traces de phases, ces phases ne sont pas simulées. Un saut dans l'exécution est alors réalisé et l'entrée correspondante dans la CST est mise à jour. Le nombre d'instructions que chaque processeur doit sauter (nouveau point de démarrage de la simulation) est calculé depuis la CST (voir tableau 3.2). Pour réaliser ce saut de simulation, plusieurs méthodes sont possibles. Parmi ces méthodes, nous avons étudié la simulation fonctionnelle et les points de contrôle (checkpoints). Notre

CS	Compteur Inst. du P0 en K inst	Compteur Inst. du P1 en K inst	Cycles	Énergie	Répétition
a,b-x,y,z	100	150	200	100	2
c,d-w,w	100	100	300	150	1

TAB. 3.2 – Le contenu de CST pour l'exemple dans le tableau 3.1

méthode AS est compatible avec ces deux méthodes (voir chapitre 2). Ainsi le saut des CS(s) qui se répètent est réalisé en utilisant l'une de ces deux méthodes.

Dans le cas où les prochaines phases ne correspondent à aucune entrée de la CST, la simulation détaillée est effectuée jusqu'à la génération dynamique d'un nouveau CS par le mécanisme de barrière de simulation, comme expliqué précédemment.

Le tableau 3.2 représente la CST utilisée dans l'exemple du tableau 3.1. La simulation s'effectue jusqu'à la génération dynamique du premier CS (a,b-x,y,z) par le mécanisme de barrière de simulation. Une entrée est allouée pour ce CS dans la CST (voir tableau 3.2). Dans ce nouveau CS, P0 exécute 2 intervalles de 50K et P1 exécute 3 intervalles de 50K instructions. Ainsi les nombres d'instructions exécutées par P0 et P1 (respectivement 100K et 150K instructions) sont sauvegardés. Par la suite, la similarité est testée entre le CS (a,b-x,y,z) qui est unique dans la CST et les phases suivantes dans les deux traces de phases du tableau 3.1. Comme on peut le voir, les prochaines phases ne correspondent pas au CS (a,b-x,y,z) déjà simulé, donc une simulation détaillée est à nouveau effectuée jusqu'à la génération dynamique du deuxième CS (c,d-w,w). Là aussi, une nouvelle entrée dans la CST est allouée pour (c,d-w,w). Chacun des deux CSs est représenté par une couleur dans le tableau 3.1. Dans l'exemple du tableau 3.1, après la simulation de deux premiers CSs, le CS (a,b-x,y,z) réapparaît. Comme ce dernier CS existe déjà dans la CST, il ne sera pas simulé. Une simulation fonctionnelle ou l'utilisation du checkpointing permet d'aller au prochain CS. D'après les statistiques sauvegardées dans la CST, P0 et P1 seront respectivement avancés de 100K et 150k instructions. De même la CST sera modifiée et le nombre de répétitions associé au CS répété sera de deux.

Nous estimons la performance de l'application entière en nous basant sur les informations de CSs et leur contribution relative au programme entier.

3.5 Résultats expérimentaux pour la méthode AS

Afin d'évaluer les performances de notre méthode AS pour le DSE dans la conception de MPSoC, nous présentons dans cette section les résultats de plusieurs expériences qui ont été menées.

3.5.1 Environnement de simulation

La plateforme de simulation MPSoC utilisée dans nos expériences est MPARM [17]. Cette plateforme contient des modèles de processeurs ARM connectés à des modèles mémoires à travers un modèle de bus SOC AMBA (voir figure 3.2). Tous les composants de la plateforme sont décrits au niveau "**Cycle Accurate and Bit Accurate**" (CABA) par SystemC.

Le module de processeur ARM est représenté par un simulateur au niveau instruction ou ISS pour "Instruction Set Simulator". Ce dernier est implémenté en C/C++ et il est encapsulé dans un emballage ou dans un "**wrapper**" implémenté en SystemC (voir figure 3.2). Ce wrapper réalise l'interface (comprenant les liens aux ports de bus AMBA qui permettent

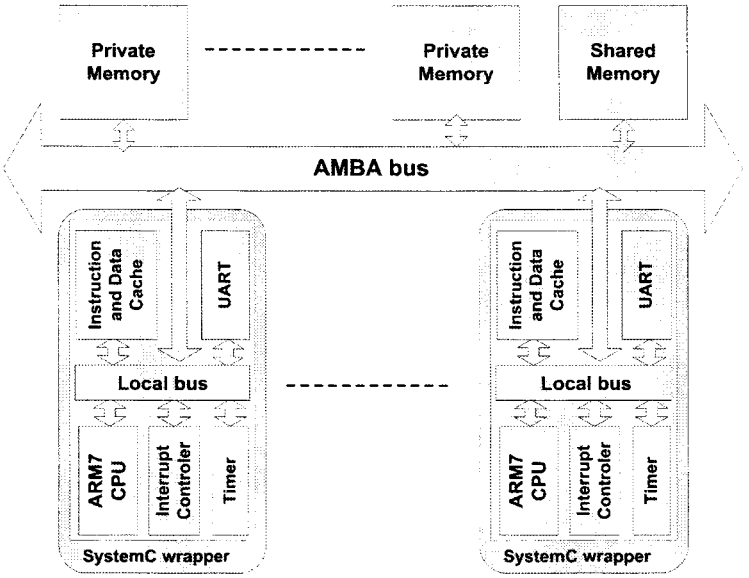


FIG. 3.2 – Architecture du MPARM

I-cache	8KB accès direct, latence 1 cycle
D-cache	4KB associative par ensembles à 4 voies, latence 1 cycle
Mémoire	latence 64 cycles
Core	Arm version 7

TAB. 3.3 – Configuration du processeur MPARM.

d'accéder à la mémoire partagée) et la synchronisation entre l'ISS et le reste de la plateforme de simulation. Chaque processeur ARM est composé d'un core ARM7, de la mémoire cache du premier niveau et des périphériques (UART, timer, interrupt controller). Tous ces composants sont implémentés en C/C++ et sont liés par un bus local. L'ISS ARM utilisé est dérivé du simulateur SWARM "**SoftWare ARM**" [25] qui décrit au niveau "**Cycle Accurate**" le core ARM7. La configuration du processeur qui a été utilisée dans les expériences est montrée dans la table 3.3.

Les mémoires partagées connectées au bus AMBA (voir figure 3.2), sont dérivées d'un modèle de mémoire implémenté en SystemC.

L'outil de compilation ou "cross-compilation" du code exécuté sur la plateforme est le gcc-3.0.4 pour la famille de processeurs ARM. De même MPARM contient le système d'exploitation RTEMS pour les systèmes temps réel multiprocesseurs. Ce système d'exploitation utilise deux dispositifs "timer" et "interrupt controller" dans chaque processeur ARM pour la gestion multi-tâche. Comme MPARM contient une mémoire pour les données partagées, alors les primitives de synchronisation (exemple les locks, les barrières de synchronisation, etc.) de processeurs sont aussi supportées par le simulateur pour gérer la cohérence des données.

En plus MPARM intègre un modèle de puissance pour chaque composant, y compris le réseau d'interconnexion, permettant l'évaluation de la consommation d'énergie. Autrement dit, pour estimer la consommation, un modèle de consommation correspondant à

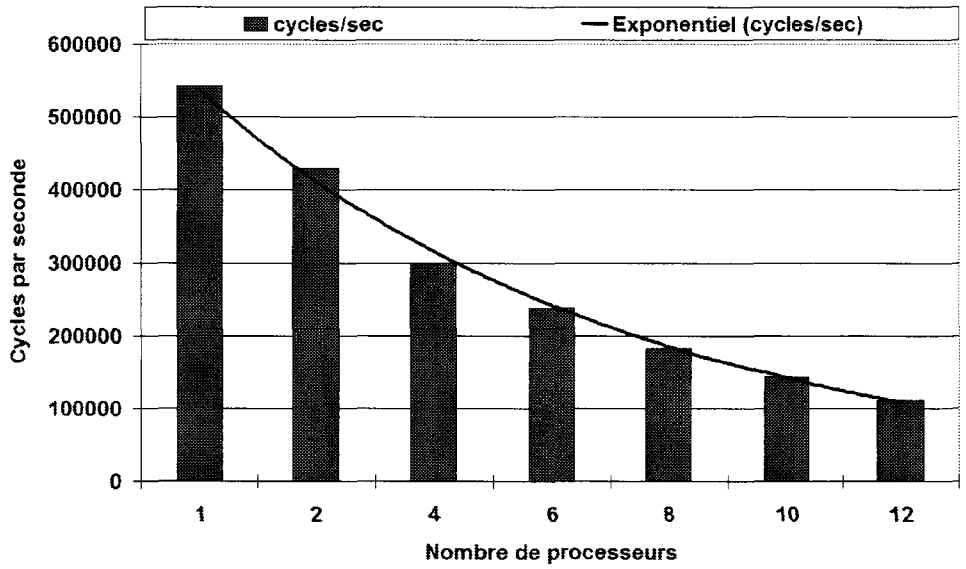


FIG. 3.3 – Variation du nombre de cycles simulés par seconde avec l'augmentation de nombre de processeurs dans le cadre du MPARM

chaque composant est intégré. Ainsi au cours de la simulation, la consommation est calculée à chaque cycle en se basant sur les occurrences des activités pertinentes des composants. Comme les composants du MPARM sont décrits au niveau CABA en SystemC alors la simulation devient très lente et augmente avec le nombre de processeurs. La figure 3.3 montre que le nombre de cycles simulés par seconde est très élevé, environ 530K cycles/sec pour deux processeurs. De même ce nombre diminue exponentiellement avec l'augmentation du nombre de processeurs et devient environ 100K cycles/sec pour 12 processeurs.

Quand l'un des benchmarks est simulé entièrement en mode CABA, selon la configuration architecturale du MPSoC, plusieurs heures d'attente sont nécessaires. La simulation est effectuée sur un Pentium4 de 3 GHZ de fréquence. La figure 3.4 montre que le temps de simulation augmente exponentiellement avec le nombre de processeurs simulés. Par exemple, une simulation totale détaillée avec 12 processeurs nécessite approximativement cinq jours de simulation. Cela montre la motivation de l'accélération de la simulation, cette motivation a été détaillée dans l'introduction.

Le facteur d'accélération est définie comme le ratio entre le nombre total d'instructions exécutées par tous les processeurs sans échantillonnage et le nombre d'instructions simulées en mode détaillé (CABA) en utilisant l'échantillonnage. Notons que notre méthode est indépendante de la plateforme de simulation adoptée.

3.5.2 Benchmarks utilisés

Les benchmarks testés pour l'évaluation de AS qui font partie de la suite Mibench [30] sont répartis en deux catégories télécommunication et sécurité.

En effet, les benchmarks de cette suite ont été portés sur notre plateforme MPSoC. MiBench a été conçue pour représenter les applications réelles des systèmes embarqués et Mibench est sans doute la suite la plus utilisée par les chercheurs. Le portage de chaque bench-

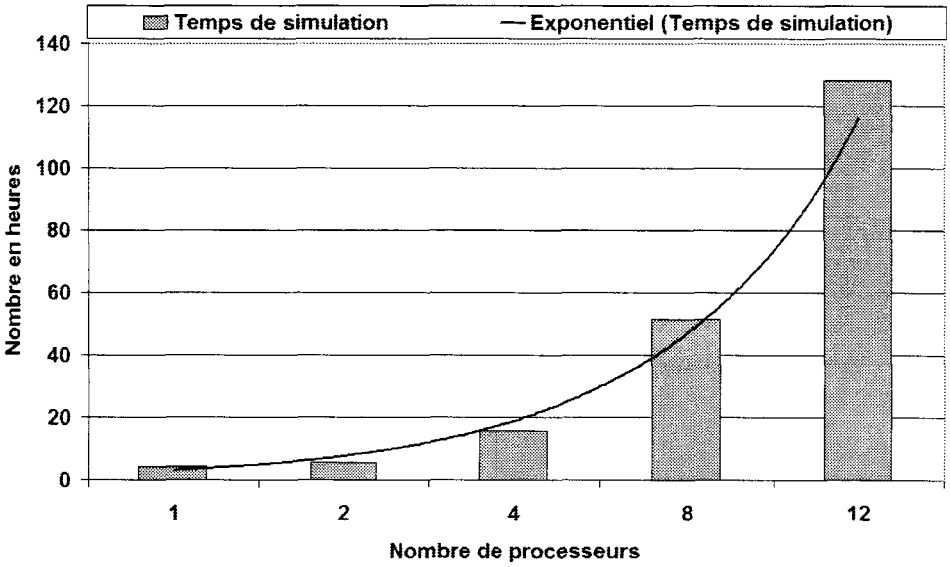


FIG. 3.4 – Variation du temps de simulation en fonction du nombre de processeurs simulés.

Série	Catégorie	Benchmark	Version	Nombre d'instructions	Nombre d'accès mémoire
Mi-Bench	Télécommunication	adpcm	encode	949 890 274	121 861 490
Mi-Bench	Télécommunication	adpcm	decode	607 441 786	73 631 141
Mi-Bench	Télécommunication	FFT	-	168 008 964	45 879 275
Mi-Bench	Télécommunication	FFT	Inverse	184 626 007	50 342 138
Mi-Bench	Télécommunication	Gsm	encode	1 251 462 740	286 061 835
Mi-Bench	Télécommunication	Gsm	decode	536 235 207	59 124 973
ISO	Télécommunication	H264	-	129 203 680	29 467 818
Mi-Bench	Sécurité	Rijndael	encode	332 082 997	141 510 968
Mi-Bench	Sécurité	Rijndael	decode	349 339 153	140 996 494
Mi-Bench	Sécurité	Blowfish	encode	311 616 420	88 958 241
Mi-Bench	Sécurité	Blowfish	decode	314 864 383	88 953 459

TAB. 3.4 – Benchmarks.

mark a nécessité des modifications au niveau du code source afin de compiler et l'exécuter sur la plateforme et surtout pour charger le fichier contenant les données d'entrée nommées aussi "input data set" dans la mémoire de notre plateforme. A cause de problèmes techniques relatifs à la lecture des fichiers d'entrée par la plateforme et pour réduire le temps, seuls cinq benchmarks de la suite Mibench ont été choisis (voir tableaux 3.4).

La catégorie "Télécommunication" contient les benchmarks suivants :

FFT/IFFT : Exécute la transformation de fourrier et son inverse. Cette transformation est utilisée dans le traitement du signal pour détecter les fréquences d'un signal donné.

GSM encode/decode : Elle est nommée GSM pour "Global Standard for Mobile communications" [10]. C'est un standard pour le codage et le décodage de la voix en Europe et dans plusieurs autres pays.

ADPCM encode/decode : Elle est nommée ADPCM pour "Adaptive Differential Pulse Code Modulation" [8]. C'est un algorithme non standardisé de compression de données, il est utilisé dans le cas des fichiers audio, en particulier les échantillons vocaux. Il repose sur la présence de la prédiction et sur le codage des erreurs entre la prédiction et le signal original.

Lors du décodage, les erreurs sont ajoutées au signal issu de la prédiction pour obtenir un signal plus ou moins fidèle à l'original.

H264 decode : C'est une norme du décodage vidéo nommée aussi MPEG-4 AVC (Advanced Video Coding) [11]. Elle est adaptée à une très grande variété de réseaux et de systèmes, tels que la diffusion de la télévision et le stockage HD DVD. L'application H264 ne fait pas partie de Mibench.

La catégorie "Sécurité" contient les benchmarks suivants :

rijndael encode/decode : C'est un standard de la cryptographie [7]. Il utilise des clés dont la taille est de 128 , 192 et 256 bits. Les données en entrée sont des caractères ASCII stockés dans un fichier texte.

blowfish encode/decode : C'est un algorithme de cryptographique [9]. Il utilise des clés de longueur variable allant de 32 à 448 bits. Les données d'entrée sont les mêmes que celles utilisées pour rijndael.

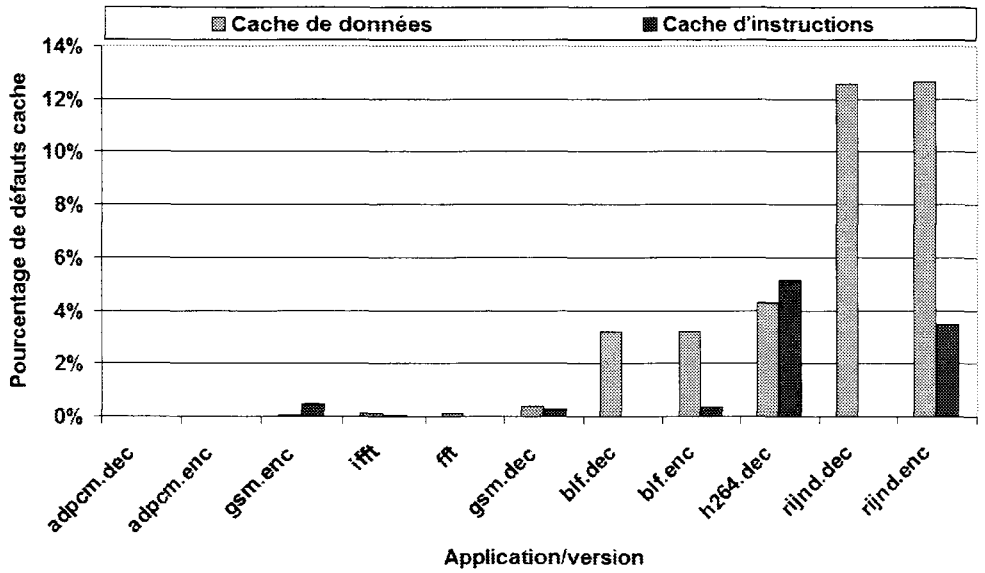


FIG. 3.5 – Pourcentage de défauts cache pour chaque application/version.

La figure 3.5 montre les pourcentages de défauts de cache de données et la cache d'instructions pour chaque application/version obtenus sur notre plateforme. Les tailles de deux mémoires cache sont celles présentées dans le tableau 3.3. Il apparaît clairement que les pourcentages de défauts cache sont très différents, rijndael ayant le pourcentage de défauts cache le plus élevé. Généralement, le pourcentage de défauts au niveau de cache d'instructions est faible car les références aux instructions ont une forte localité.

Dans le cadre de ce travail, nous nous intéressons à la mise en oeuvre de la méthode AS sur des applications parallèles indépendantes ou non communicantes. Nous ne traitons pas le cas des applications parallèles communicantes. Ces dernières contiennent généralement des primitives de synchronisation telles que les locks et les barrières de synchronisation. L'application de la méthode AS pour ce type d'application sera une extension de ce travail.

Pour chacun des cinq benchmarks, les deux versions "encode" et "decode" ont été utilisées. Durant les expériences, le nombre de processeurs varie de 2 à 12. Pour les benchmarks qui s'exécutent en parallèle, une moitié des processeurs exécutent la version "encode"

et l'autre moitié la version "decode". Pour l'application H264, les processeurs exécutent la même version "decode".

3.5.3 Génération de phases des applications

Durant les expériences, nous avons utilisé 50K instructions comme taille d'intervalle. Cette taille de 50K instructions a été choisie parmi différentes tailles. En effet une petite taille facilite la détection des phases du comportement réel de l'application et permet une accélération de la simulation.

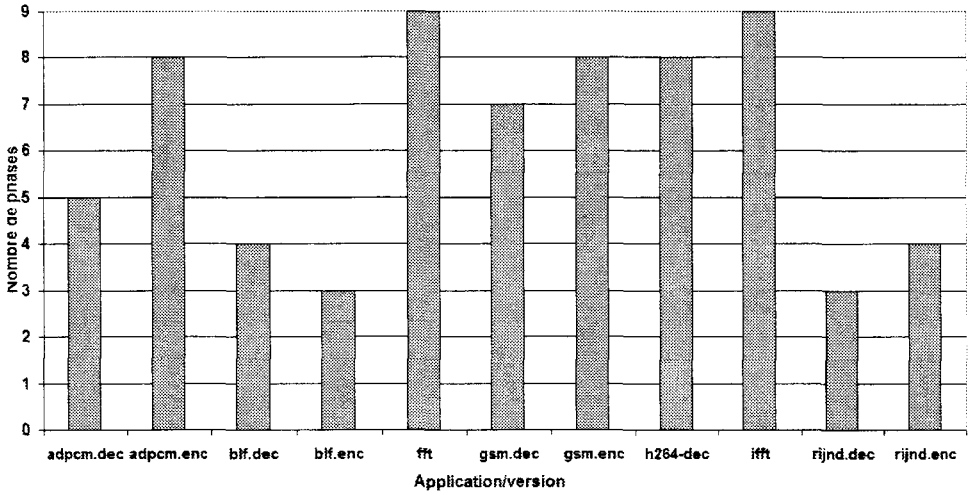


FIG. 3.6 – Nombre de phases détectées pour chaque application/version.

SimPoint permet à l'utilisateur de fixer le nombre maximum de phases pour chaque application par le paramètre **Max_K**. Le nombre de phases trouvées dépend de la valeur du **Max_K**, donc cette dernière valeur influe sur le temps de simulation. Si SimPoint trouve un nombre de phases égal à **Max_K** alors cela peut indiquer que le nombre de phases soit insuffisant pour représenter le comportement de toute l'application [31]. Dans ce cas, **Max_K** sera doublé et l'analyse de SimPoint sera reprise. Comme nous nous intéressons à accélérer la simulation, nous avons choisi un **Max_k** égal à 10. La figure 3.6 montre le nombre de phases détectées pour chaque application/version. Le nombre de phases détectées est plus petit que 10. Ceci indique que la valeur de **Max_K** est suffisante pour représenter le comportement total de nos applications/versions. Dans le cas de cophase [18], un **Max_K** de 20 et des intervalles de 10 millions d'instructions pour des applications dont la taille moyenne est de 100 milliards d'instructions sont utilisés.

3.5.4 Relation entre TWSB et l'accélération de la simulation

La figure 3.7 montre l'effet du seuil du TWSB sur le nombre total de CSs générés ainsi que sur le nombre de groupes de CSs similaires. Chaque groupe de CSs similaires a un seul CS représentatif qui subit la simulation en mode CABA. Ainsi le nombre de CSs représentatifs de l'application entière correspond au nombre de groupes de CSs similaires. Dans la

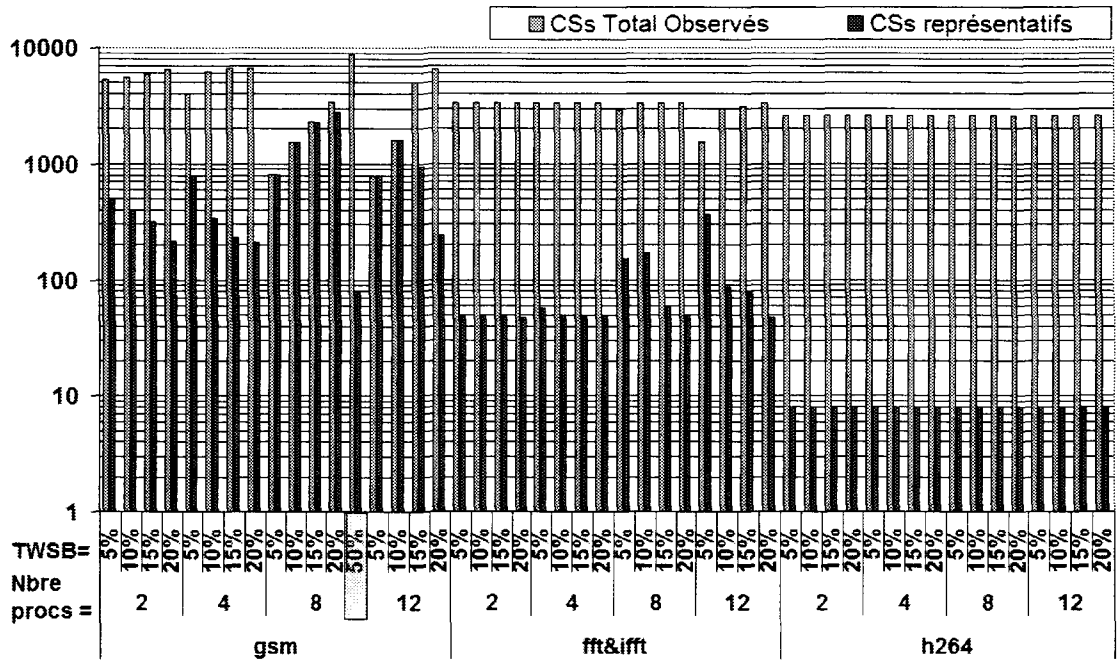


FIG. 3.7 – Le nombre total de CSs générés et le nombre de CSs représentatifs avec 4 valeurs du TWSB. La même application est dupliquée sur 2, 4, 8 et 12 processeurs. Le TWSB de 50% est fait seulement pour GSM sur 8 processeurs.

suite de cette thèse, le nombre de groupes de CSs similaires est appelé "nombre de CSs représentatifs". Comme on peut le voir, lorsque la valeur choisie pour TWSB est importante, la probabilité de générer une barrière de simulation devient plus grande puisque nous autorisons des temps d'attente importants au niveau des barrières de simulation. Dans ce cas, la taille moyenne de CSs en termes de nombre de phases est réduite. Ainsi, le nombre total de CSs générés augmente et le nombre de CSs représentatifs diminue avec l'augmentation du TWSB. La figure 3.7 montre pour chaque application exécutée sur un nombre de processeurs donné que le nombre de CSs représentatifs diminue et le nombre total de CSs générés augmente avec l'augmentation du TWSB. Celui-là offre une opportunité d'avoir des CSs qui se répètent et, par conséquent, un facteur d'accélération plus important, (voir figure 3.8).

La figure 3.8 montre l'augmentation du facteur d'accélération de simulation avec l'augmentation du TWSB, sur 2, 4, 8 et 12 processeurs. Les trois applications blowfish, rijndael et h264 montrent un facteur d'accélération important. Le meilleur facteur d'accélération est approximativement de 795 et 280 respectivement pour blowfish et rijndael. Cela est dû au fait que les applications blowfish (encode et decode), rijndael (encode et decode) ont un nombre de phases réduit (voir figure 3.6). Autrement dit, dans ces deux applications, il existe un nombre réduit de blocs de base. Pour l'application h264, comme c'est la même application qui est exécutée sur tous les processeurs, ces derniers ont le même comportement et finissent presque en même temps leurs intervalles respectifs. Ainsi les CSs générés comportent une phase par processeur pour toutes les valeurs du TWSB. Le nombre total de CSs générés ainsi que le nombre de CSs représentatifs se stabilisent quelque soit la valeur du TWSB (voir fi-

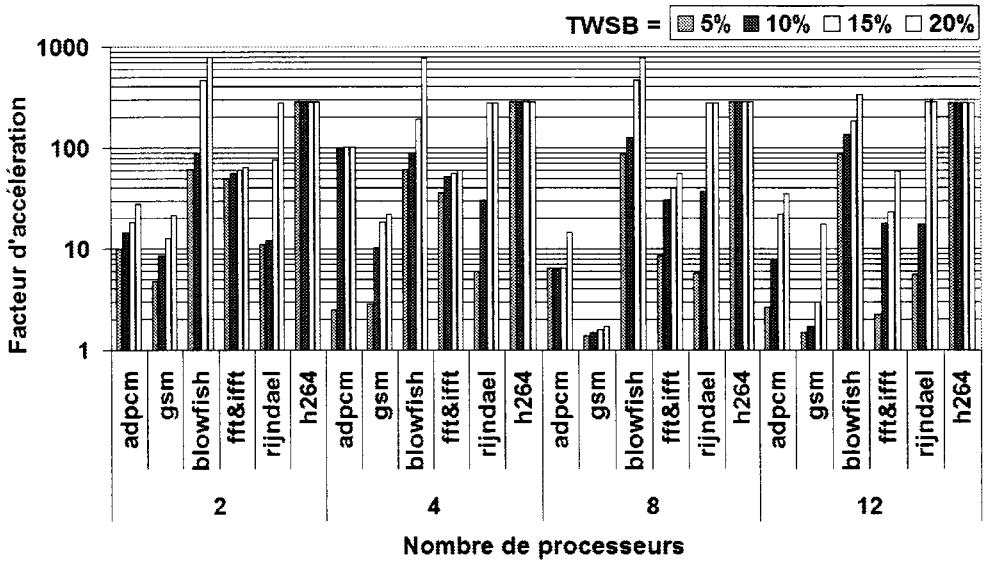


FIG. 3.8 – La variation du facteur d'accélération (en log) avec quatre valeurs du TWSB. La même application est exécutée sur 2, 4, 8, 12 processeurs.

gure 3.7). Par conséquent, le facteur d'accélération sur les 4 configurations de processeurs est quasiment constant (proche de 300). Pour les mêmes raisons, le facteur d'accélération de la simulation, pour les applications fft&ifft, est quasiment constant quand le TWB varie entre 10% et 20%. Les CSs générés contiennent une phase par processeur dès que le TWSB devient plus grand que 10%. En effet, la figure 3.7 montre que le nombre total de CSs générés et le nombre de CSs représentatifs pour fft&ifft se stabilisent quand TWSB dépasse 10%.

Le facteur d'accélération de gsm sur 8 processeurs est proche de 1.5. La raison de cette faible performance est le nombre important de CSs représentatifs ainsi que l'irrégularité de gsm.encode et gsm.decode. Par conséquent, quand le nombre de processeurs dépasse 8, la taille de CSs augmente de façon sensible. Dans ce cas, le nombre de CSs observés baisse et ainsi le facteur d'accélération décroit. Pour un TWSB de 20%, le nombre total de CSs observés est environ 6691 et 3410 respectivement sur 4 et 8 processeurs (voir figure 3.7). Une valeur du TWSB de 50% pour gsm sur 8 processeurs augmente le nombre total de CSs observés environ jusqu'à 9000 (voir figure 3.7). Ce qui permet d'augmenter le facteur d'accélération à 97.7 sans sacrifier la précision dans l'estimation de la performance. L'erreur est environ 4.5%. Un contrôle dynamique du TWSB selon le comportement de l'application est proposé comme perspective de la thèse. Cela va permettre d'obtenir un facteur d'accélération plus important pour les applications ayant un comportement irrégulier comme c'est le cas de gsm.

Nous remarquons que, lorsque la même application est répliquée sur tous les processeurs, le facteur d'accélération reste important quand le nombre de processeurs simulés augmente (voir figure 3.8). Généralement les processeurs qui exécutent la même application vont effectuer les même intervalles dans chaque CS. Ainsi la même séquence de phases dans les CSs est détectée quelque soit le nombre de processeurs simulés. Par conséquent, le facteur d'accélération sera presque le même sur tous les processeurs simulés. La figure 3.8 montre que, pour un TWSB de 20%, blowfish, fft&ifft et rijndael ont séparément le même

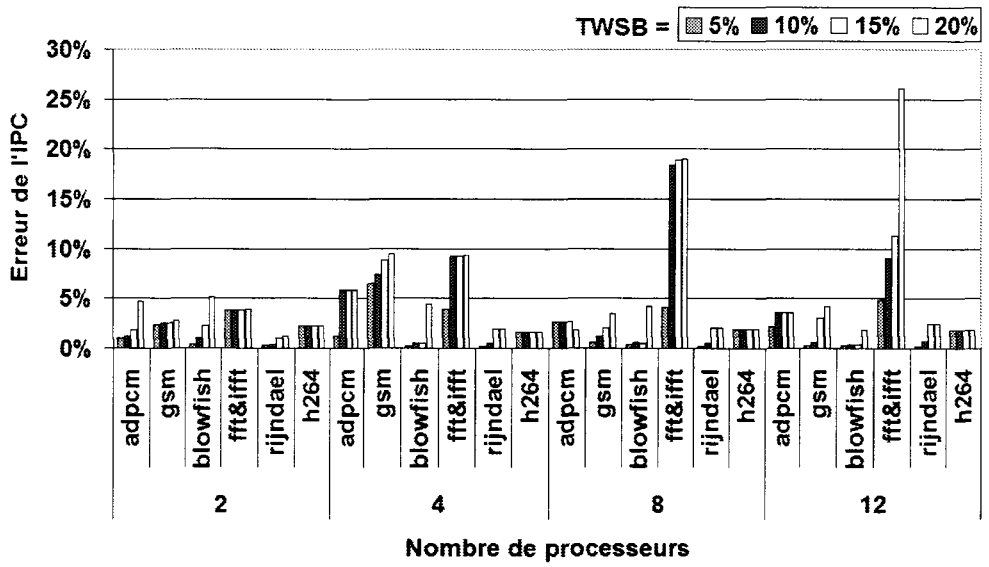


FIG. 3.9 – La variation de l'erreur de l'IPC avec quatre valeurs du TWSB. La même application est dupliquée sur 2, 4, 8, 12 processeurs.

facteur d'accélération sur 2, 4, 8 et 12 processeurs.

3.5.5 Relation entre TWSB et l'erreur d'estimation

L'erreur dans l'estimation de l'IPC et de la consommation de puissance provient de 2 sources. La première source vient du fait de l'approximation de la valeur de l'IPC des différents CSs par l'IPC de leur CS représentatif. Les intervalles identifiés comme faisant partie d'une même phase, en utilisant Simpoint, peuvent être en réalité différents. Ainsi l'hypothèse de similarité des CSs peut être une source d'erreur.

La deuxième source d'erreur est due à l'insertion des barrières de simulation. Cette erreur est due au nombre de cycles d'attentes injectés lors de la simulation au niveau des barrières de simulation. Les cycles d'attentes causent deux problèmes :

1. Comme aucune instruction n'est exécutée durant l'attente à une barrière de simulation, l'IPC est sous estimé.
2. Le recouvrement des applications parallèles est légèrement différent du recouvrement réel de ces applications sans barrières de simulation. En effet, l'ajout de barrières de simulation (en particulier pour des TWSB importants) risque de générer des décalages importants entre applications concurrentes, ce qui risque de modifier la mesure des performances.

La figure 3.9 montre l'erreur de l'IPC estimé en fonction de quatre valeurs différentes du TWSB pour six applications répliquées sur 2, 4, 8 et 12 processeurs. Ces applications sont les mêmes que celles de la figure 3.8. Nous observons que pour chaque configuration de processeurs, l'erreur est corrélée avec la valeur de TWSB. En effet, quand TWSB augmente, le nombre de barrières de simulation injectées augmente et par conséquent l'erreur due à l'injection de cycles d'attentes des barrières augmente. Il y a deux cas où l'erreur de l'IPC varie de façon irrégulière. Ces deux cas sont : gsm sur 2 processeurs et adpcm sur 12 processeurs.

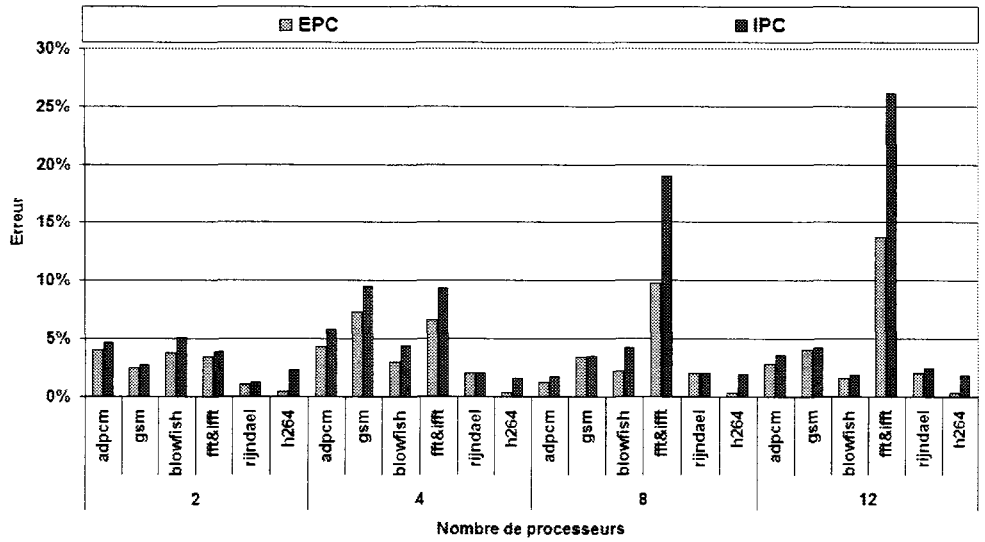


FIG. 3.10 – L'erreur de l'EPC et de l'IPC pour un TWSB de 20%. La même application est dupliquée sur 2, 4, 8, 12 processeurs.

Cette variation irrégulière de l'erreur est due à la compensation de deux sources d'erreurs détaillées précédemment. En fait, l'erreur totale est une accumulation de ces deux erreurs. Généralement l'erreur due à l'insertion des barrières de simulation augmente avec la valeur du TWSB, tandis que l'erreur due à l'association des IPCs varie de façon irrégulière.

Dans le cas de l'application h264, l'erreur ne varie pas avec TWSB. Comme il a été expliqué dans la section 3.5.4, ce résultat est dû à la stabilisation du nombre total de CSs. Ainsi le nombre de cycles injectés par les barrières de simulation est le même pour les quatre valeurs du TWSB. Cette analyse s'applique aussi pour fft&ifft, où l'erreur se stabilise quand la valeur du TWSB est plus grande que 10%. Cela est dû à la stabilisation en nombre de barrières de simulation ainsi qu'en nombre de CSs générés donc le nombre de cycles injectés par les barrières de simulation ne change plus quand TWSB dépasse les 10%. Avec l'augmentation du TWSB, l'erreur de fft&ifft varie entre 4% et 9% sur 4 processeurs, entre 4% et 18% sur 8 processeurs et entre 5% et 26% sur 12 processeurs. Dans ce cas, l'utilisateur peut choisir une valeur réduite du TWSB pour améliorer l'erreur sans sacrifier l'accélération. Par exemple, un TWSB de 5% peut être choisi afin d'améliorer l'erreur à 4% et obtenir environ un facteur d'accélération de 36, 9 et 3 respectivement pour 4, 8 et 12 processeurs (voir figure 3.8).
Ainsi l'avantage de notre approche est que, grâce au TWSB, un compromis entre la précision et le facteur d'accélération puisse être trouvé. Nos résultats indiquent qu'en utilisant les barrières de simulation, la quantité d'instructions simulées dans le mode détaillé est ajustée automatiquement. Pour un TWSB de 5%, le pourcentage d'erreur de l'IPC est de moins de 4% tandis que le taux d'instructions simulées en détails varie entre 1.5% et 21%. A l'opposé, les autres approches d'échantillonnage peuvent difficilement offrir ce compromis.

En ce qui concerne l'évaluation de la consommation de puissance, la figure 3.10 montre l'erreur dans l'estimation de *"Énergie Par Cycle"* ou EPC . Cette figure compare aussi l'erreur de l'EPC à celle de l'IPC (reprise depuis figure 3.9). La valeur du TWSB utilisée est 20%. Les applications sont les mêmes que celles de la figure 3.9. Comme on peut le voir, l'erreur de

l'EPC est plus faible que celle de l'IPC. On remarque aussi que les erreurs de l'IPC et celles de l'EPC ont les mêmes variations. Autrement dit, les applications qui sont les plus précises au niveau IPC sont aussi les plus précises au niveau EPC. Par exemple, H264 est la plus précise en IPC et en EPC pour chacune des configurations. Comme la présision de l'EPC peut être approximée par celle de l'IPC et comme l'estimation de l'IPC est plus difficile que celle de l'EPC, dans la suite de cette thèse nous nous intéressons uniquement à l'erreur sur l'IPC.

3.5.6 Comparaison de AS avec la méthode Cophase

Dans cette section, nous comparons le facteur d'accélération obtenu par AS à celui obtenu par la méthode cophase. Comme nous l'avons montré dans le chapitre précédent, trois façons différentes ont été utilisées pour la génération des échantillons dans le cadre de cophase [18, 21]. La sélection de nouveaux échantillons se terminent dans un de trois cas :

1. Quand la somme des instructions exécutées par tous les processeurs est égale à 5 millions.
2. Quand un processeur exécute exactement 5 millions d'instructions.
3. Quand chacun des processeurs a exécuté au moins 5 millions d'instructions.

Ainsi chaque échantillon qui subit la simulation détaillée contient une ou plusieurs cophases. La taille des intervalles est de 10 millions d'instructions et les différentes périodes utilisées pour re-simuler la même cophase sont 20 et 100. Michael Van Biebrouck et al. rapportent que les meilleurs résultats sont obtenus par cophase quand chaque processeur exécute au moins 5 millions d'instructions. Dans nos expériences, nous avons retenu le même principe pour tester la méthode cophase : dans chaque échantillon, chaque processeur exécute au moins 25K instructions pour des tailles d'intervalles de 50K. Comme nous nous intéressons à l'accélération, nous avons choisi la plus grande période de re-simulation de cophases adoptée (période = 100). Dans le cas de la méthode AS, nous avons choisi le plus grand TWSB.

La figure 3.11 montre le facteur d'accélération (en log) donné par AS et celui donné par cophase sur 2, 4, 8 et 12 processeurs. Comme on peut le voir, AS offre une accélération plus importante que cophase. En plus, cophase n'accélère plus pour 8 et 12 processeurs. Cette faible accélération de cophase est due à un nombre important de cophases représentatives générées (nommé aussi nombre de cophases observées) et elle est due aussi à la re-simulation des mêmes cophases.

La figure 3.12 montre le nombre de cophases représentatives et le nombre de CSs représentatifs pour AS. Pour cophase, le nombre de scénarios de phases conjoints est important. Par ailleurs ce nombre augmente avec le nombre de processeurs. Comme on peut le voir, le nombre de cophases représentatives est beaucoup plus important que le nombre de CSs représentatifs. Le premier peut dépasser le 100000 alors que le deuxième est au maximum de 1200. Généralement dans le cadre de AS, les barrières de simulations de AS permettent aux mêmes intervalles des mêmes applications concurrentes de s'exécuter en même temps. Autrement dit, il y a une synchronisation entre les mêmes applications de telle sorte que les mêmes intervalles se trouvent dans les mêmes CSs. Ce qui réduit le nombre de scénarios de phases s'exécutant en parallèle réduisant ainsi le nombre de CSs représentatifs. Cet aspect augmente le facteur d'accélération tout en gardant une estimation précise de la performance.

La figure 3.13 montre l'erreur dans l'estimation de l'IPC par AS et par cophase pour 2 et 4 processeurs. L'erreur pour 8 et 12 processeurs n'est pas présentée dans cette figure car

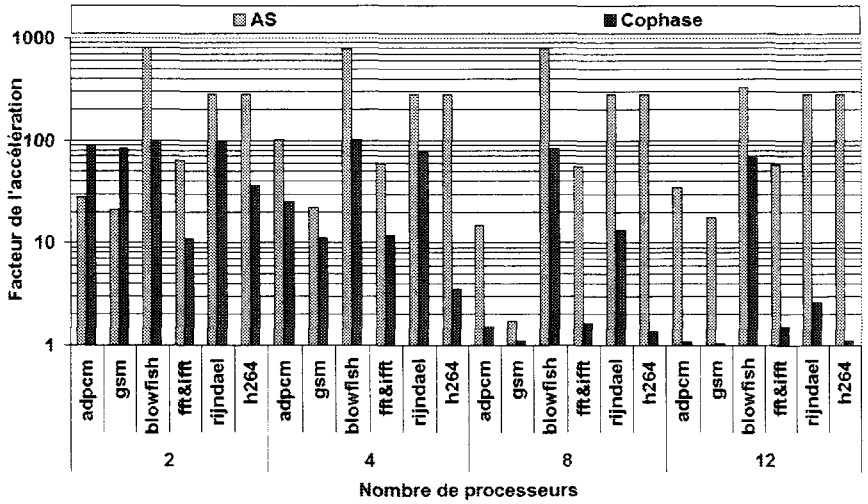


FIG. 3.11 – Le facteur d'accélération donné par AS (TWSB = 20%) et par cophase. La même application est dupliquée sur 2, 4, 8 et 12 processeurs.

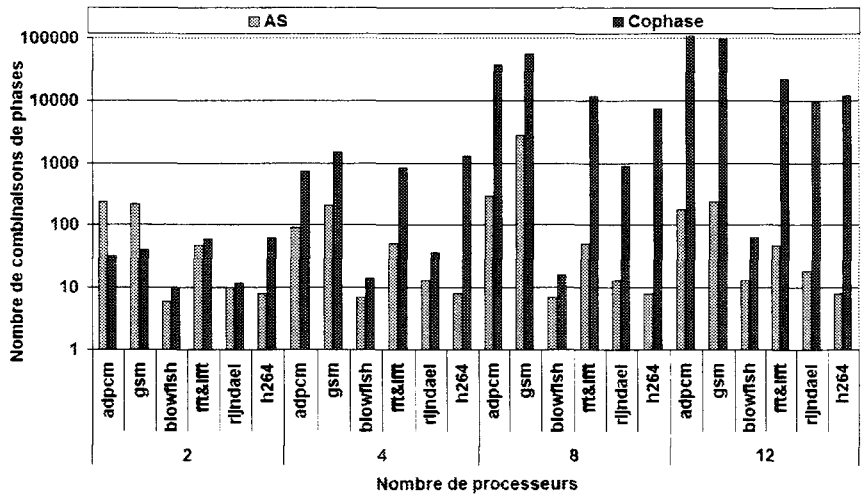


FIG. 3.12 – Le nombre de combinaisons de phases (en log) donné par AS (TWSB = 20%) et par cophase. La même application est dupliquée sur 2, 4, 8 et 12 processeurs.

cophase n'accélère pas la simulation. Les erreurs par AS et par cophase sont proches l'une de l'autre malgré les différences dans le facteur d'accélération. Les deux applications où l'erreur de AS vaut plus que le double de l'erreur de cophase sont respectivement adpcm et fft sur 4 processeurs. Le facteur d'accélération donné par AS est respectivement de 102 et 60 pour adpcm et fft, tandis que celui donné par cophase vaut respectivement 25 et 11. En effet, le facteur d'accélération de AS est quatre fois plus grand que celui obtenu par cophase pour ces deux applications. En réduisant le TWSB à 5% nous obtenons un facteur d'accélération de 30 pour fft sur 4 processeurs avec une erreur égale à celle de cophase (voir les deux figure 3.8 et 3.9). Ainsi pour la même précision le facteur d'accélération donné par AS est plus important que celui donné par cophase. La figure 3.11 montre que, pour les applications adpcm et

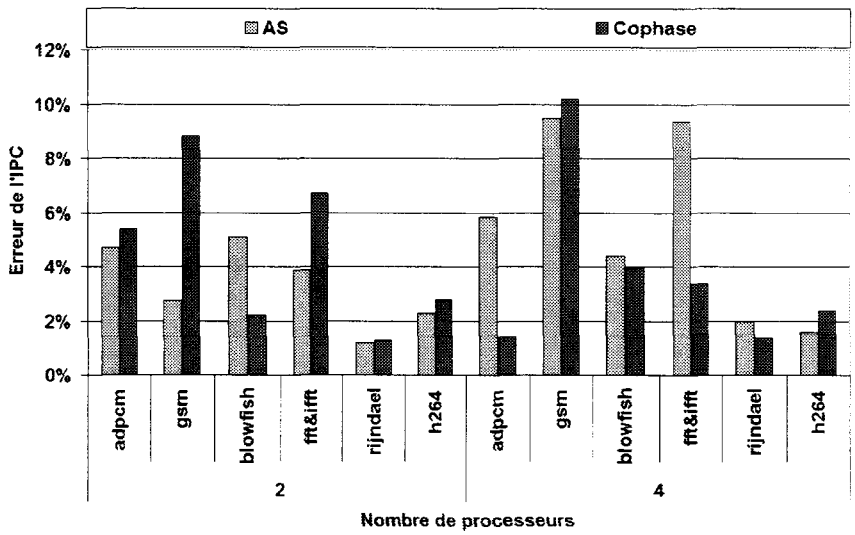


FIG. 3.13 – L'erreur de l'IPC qui est donné par AS (TWSB = 20%) et par cophase. La même application est dupliquée sur 2 et 4 processeurs.

gsm sur 2 processeus, le facteur d'accélération donné par cophase est plus grand. En augmentaut le TWSB à 40% les deux facteurs d'accéleration de adcpm et gsm sur 2 processeurs augmentent respectivement à 198 et à 98 tout en gardant une erreur plus petite que 10%. De même pour un TWSB de 50% le facteur d'accélération de gsm sur 8 processeurs augmente à 97,7 avec une erreur de 4,5% (voir section 3.5.4) tandis que le facteur obtenu par cophase est de 1,71 (voir la figure 3.11). Celui-là montre que le TWSB contrôle le compromis entre le facteur d'accélération et la précision de la performance estimée. Ce contrôle n'est pas offert par la méthode cophase.

3.5.7 Accélération pour des applications différentes

La figure 3.14 montre la variation du facteur d'accélération avec 4 valeurs du TWSB pour différents groupes d'applications de la suite Mibench exécutées sur 4, 8 et 12 processeurs. Les applications qui s'exécutent en parallèle sur les 4 processeurs sont différentes. Ainsi,chacun des quatre processeurs exécute une application différente puisque 2 versions (encode et décode) dans les différentes applications sont considérées. Quand le nombre de processeurs simulés est 8 ou 12, la même version de l'application est exécutée respectivement par 2 ou 3 processeurs. Nous désignons par "les applications différentes" celles qui s'exécutent en parallèle et qui ont une grande différence dans leur comportement et dans leur performance. Ces applications sont aussi appelées "applications hétérogènes".

Les deux figures 3.15(a) et 3.15(b) donnent les tailles de séquences des CS générées pour les groupes d'applications exécutés sur 4 processeurs. Dans cette expérience, le TWSB est fixé à 20%. Les tailles des séquences montrées dans cette figure sont calculées en nombre de phases.

Comme on peut le voir, le facteur d'accélération des groupes d'applications de la figure 3.14 est relativement faible (proche de 1.5). Ceci est dû à l'augmentation du nombre de combinaisons de phases quand les applications qui s'exécutent en parallèle sont différentes.

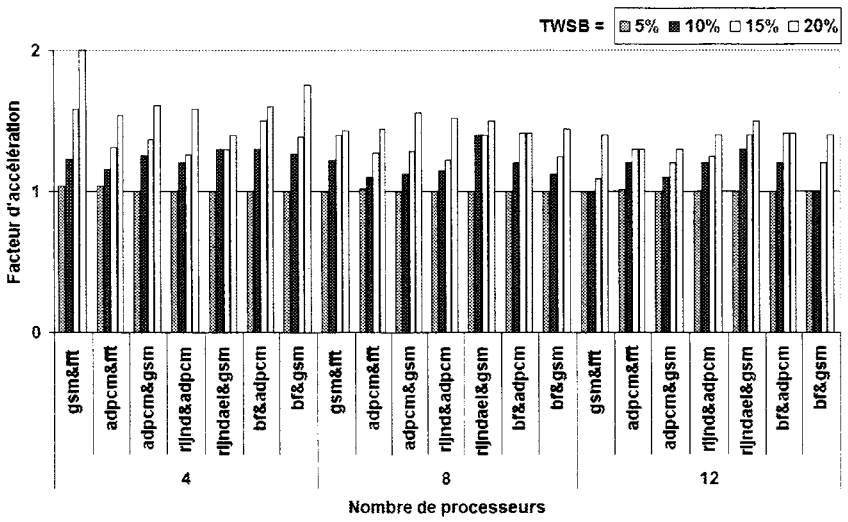
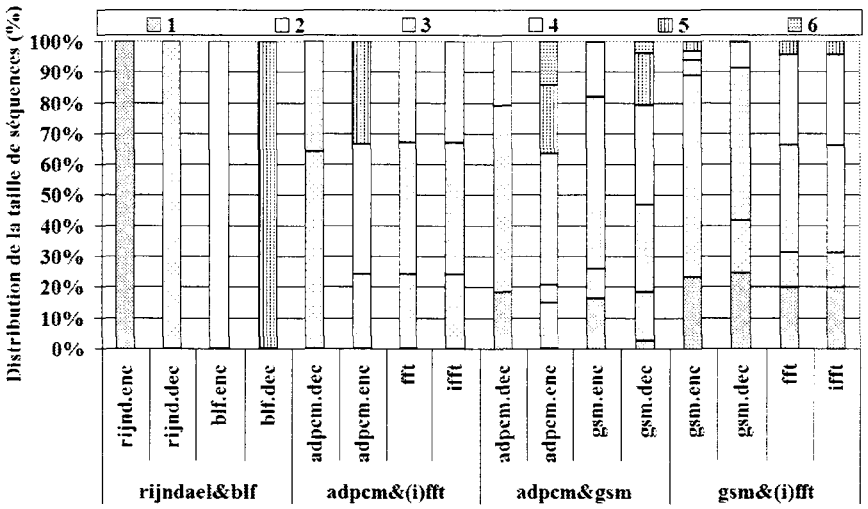


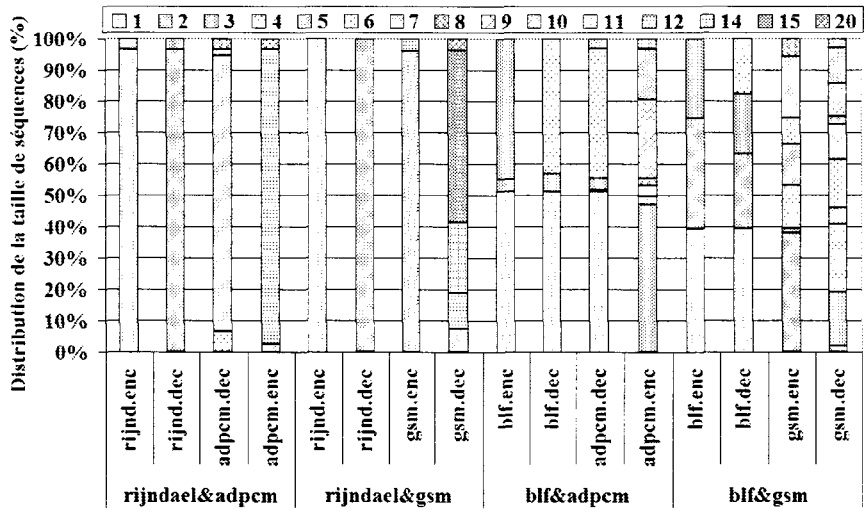
FIG. 3.14 – La variation du facteur d'accélération avec quatre valeurs du TWSB. Des applications différentes sont exécutées sur 4, 8, 12 processeurs

Lorsque des applications avec des besoins différents en termes de ressources matérielles, s'exécutent en parallèle, le nombre de combinaisons de phases est très élevé et par conséquent le nombre CSs représentatifs augmente, d'où un très faible facteur d'accélération. La figure 3.14 montre ce cas. Le facteur d'accélération de la plupart des applications de cette figure est d'environ 1.5. Le pourcentage des CSs représentatifs augmente avec l'augmentation du nombre de processeurs et précisément quand les applications qui s'exécutent en parallèle sont différentes. Dans ce cas, le nombre d'instructions demandées (instructions de tous les CSs simulés en détail) pour accomplir la simulation par échantillonnage s'approche de celui de la simulation totale détaillée.

Il y a deux facteurs qui produisent un accroissement du nombre de combinaisons de phases. Le premier facteur est le nombre de phases au niveau des applications qui s'exécutent en parallèle ; le deuxième facteur concerne la taille de la séquence de phases dans les CSs. Cette taille devient grande quand les applications qui s'exécutent en parallèle ont des comportements différents en particulier en termes d'accès mémoires. En effet, lorsque l'une des applications a un taux de défauts de cache relativement plus faible que les autres applications, la taille de la séquence des phases obtenue pour cette application est plus importante. Les deux figures 3.15(a) et 3.15(b) montrent les différentes tailles de séquences de phases pour les applications que nous avons considérées. A titre d'exemple, la séquence de phases de gsm et de adpcm peut contenir jusqu'à 20 phases quand l'un ou l'autre de ces 2 benchmarks est exécuté en parallèle avec rijndael. En effet, le benchmark rijndael a un nombre de défauts cache très élevé comparativement à gsm ou adpcm (voir figure 3.5). Par conséquent il y a une difficulté pour détecter une similarité entre ces longues séquences de phases. De même quand les séquences de phases deviennent relativement longues, le nombre de CSs générés diminue. Cette diminution provoque une faible probabilité pour retrouver des CSs qui se répètent, ce qui fait diminuer le facteur d'accélération. La figure 3.14 montre le besoin d'une technique supplémentaire à notre approche afin de traiter le cas des applications différentes ou hétérogènes. Cette nouvelle technique devra réduire le nombre CSs représen-



(a) Combinaison de benchmarks ayant au maximum 6 phases dans les CSs.



(b) Combinaison de benchmarks ayant au maximum 20 phases dans les Css.

FIG. 3.15 – Le pourcentage des nombres différents de phases dans les CSs est montré pour chaque benchmark/version. Chaque combinaison de benchmarks est exécutée sur 4 processeurs et le TWSB est fixé à 20%.

tatifs et ainsi augmenter le facteur de l'accélération dans ce cas. Cette réduction doit être faite en diminuant le nombre de combinaisons de phases sans sacrifier la précision. Dans le chapitre suivant nous proposons une technique nommée "*Synthèse de Clusters*" pour traiter les applications hétérogènes.

3.5.8 Surcharge de la méthode AS sur le temps d'exécution

Dans cette section, nous nous intéressons à mesurer le temps supplémentaire (ou overhead) au niveau de la simulation induit par l'utilisation de la méthode AS. Ce n'est donc

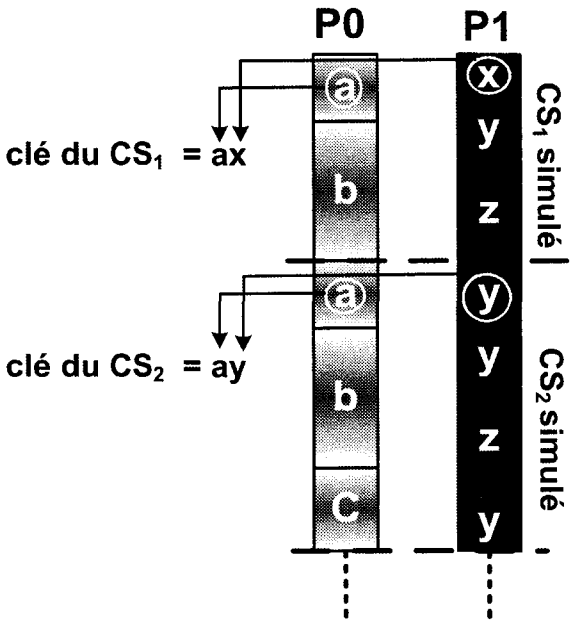


FIG. 3.16 – Génération des clés des 2 CSs , CS_1 et CS_2.

pas le temps de simulation totale obtenu. Rappelons que dans le calcul du facteur d'accélération, ce temps n'a pas été pris en compte (considéré nul). Le temps supplémentaire de la méthode AS correspond au nombre d'accès à la table de CST multiplié par le temps d'accès à cette table plus le temps de comparaison des CSs. La CST a été implémentée sous forme d'une table de hachage où chaque entrée est une liste de CSs possédant la même clé. La clé est une combinaison de lettres de la première phase de chaque séquence du CS. La figure 3.16 montre les clés des deux CSs. Ainsi les CSs dont les séquences de phases démarrent par les mêmes phases appartiennent à une même entrée (même liste). Cette technique, malgré sa simplicité, permet de réduire le nombre d'accès à la CST.

Rappelons qu'à la fin de chaque CS (simulé ou non), il est nécessaire de réaliser plusieurs comparaisons afin de rechercher les phases suivantes des applications au niveau de la CST.

Afin d'évaluer l'overhead de la méthode AS, nous avons réalisé plusieurs expériences en utilisant l'outil Perfmon [34]. Ce dernier permet d'obtenir le nombre de cycles consommés sur le calculateur hôte réalisant la simulation.

La figure 3.17 donne le nombre de cycles nécessaires pour simuler les CSs représentatifs ainsi que le nombre total de cycles consommés uniquement par AS. Les deux groupes d'applications rijndael&bl et rijndael&gsm qui ont été utilisés dans les expériences sont exécutés sur 4, 8 et 12 processeurs. La valeur de TWSB est fixée à 20%. Nous avons choisi rijndael&blowfish car cette combinaison donne le plus important facteur d'accélération. De même, nous avons choisi rijndael&gsm car cette combinaison donne le plus faible facteur d'accélération. Ces deux groupes d'applications représentent par conséquent les deux cas extrêmes.

La figure 3.17 montre que le temps de traitement additionnel induit par la méthode AS est largement négligeable par rapport au temps de simulation des CSs représentatifs (environ $10^2/10^8$). De même, la figure 3.17 montre que la croissance en temps du traitement de As est

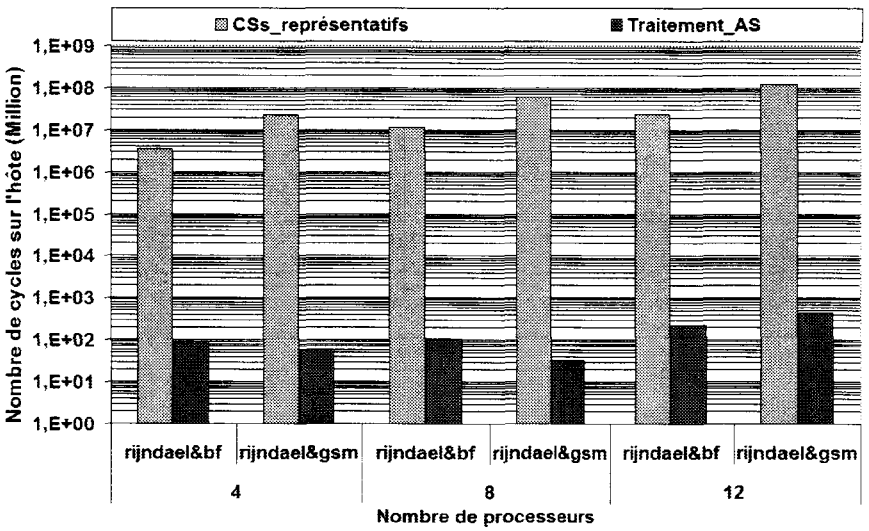


FIG. 3.17 – . Le nombre de cycles (en log) consommé par l'ordinateur hôte durant la simulations des CSs représentatifs et durant le traitement total de la méthode AS. Les groupes d'applications sont exécutés sur 4, 8 et 12 processeurs et le TWSB est fixé à 20%.

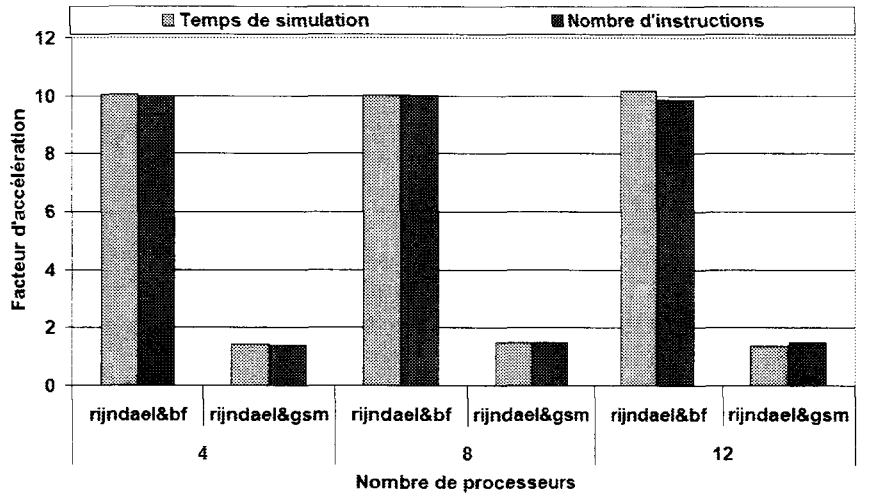


FIG. 3.18 – . Le facteur d'accélération en utilisant le nombre d'instructions et en utilisant le temps de simulation pour deux groupes d'applications rijndael&bf et rijndael&gsm sur 4, 8 et 12 processeurs. Le TWSB est fixé à 20%.

très faible avec l'augmentation du nombre de processeurs simulés contrairement au temps de simulation de CSs représentatifs.

Dans ce travail, nous avons utilisé le facteur d'accélération en nombre d'instructions pour évaluer la performance de notre méthode. Ce facteur est défini dans la section 3.5 comme le ratio entre le nombre total d'instructions exécutées par tous les processeurs et le nombre d'instructions simulées en mode détaillé (nombre d'instructions de tous les CSs représentatifs). Tandis que le facteur d'accélération en temps de simulation correspond au ratio entre le nombre total de cycles consommé par la simulation totale détaillée sur l'ordinateur hôte et

le nombre total de cycles de la simulation détaillée de tous les CSs représentatifs et du traitement de AS. La figure 3.18 compare ces deux facteurs d'accélération pour deux groupes d'applications (les mêmes groupes d'applications que la figure 3.17) exécutés sur 4, 8 et 12 processeurs. Le TWSB est fixé a 20%. Il est clair que les deux méthodes donnent des valeurs trés proches. Dans la suite de ce travail, le facteur d'accélération sera calculé en utilisant le nombre d'instructions.

Nous observons que le facteur d'accélération en nombre d'instructions est égal à celui du temps de simulation. Comme nous l'avons présenté dans la section 3.5.1, le temps moyen nécessaire pour réaliser une simulation sur 12 processeurs est d'environ 5 jours. Le plus faible facteur d'accélération donné par notre méthode AS pour nos simulations sur 12 processeurs est d'environ 17,7 pour gsm (voir figure 3.8). Alors que le temps de simulation accéléré correspond à 7 heures et 30 min.

3.6 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle méthode pour l'accélération de la simulation pour les MPSoC. Cette méthode permet d'adapter dynamiquement les échantillons ainsi que le pourcentage des échantillons prélevés de telle sorte que l'erreur de performance estimée reste acceptable. Les échantillons générés par AS correspondent à des recouvrements de séquences de phases qui sont associées à des applications qui s'exécutent en parallèle. Seuls les nouveaux recouvrements subissent la simulation détaillée. Les expériences montrent que la méthode proposée permet un échantillonnage adaptatif des applications en se basant sur leur comportement dynamique sur la plateforme architecturale. Dans le cas des applications homogènes, un facteur d'accélération atteint le 800 avec une erreur inférieure à 10%. De même, en comparant AS avec la méthode Cophase pour les applications homogènes, nous avons remarqué que AS est plus performante au point de vue accélération et précision.
Cependant, quand les applications qui s'exécutent en parallèle comportent des instructions avec des besoins différents en ressources matérielles, les séquences de phases deviennent longues. La détection de la similarité entre les recouvrements contenant de longues séquences de phases devient difficile. Ainsi, il y a une faible probabilité pour retrouver des recouvrements de phases qui se répètent et par conséquent un faible facteur d'accélération. Le facteur d'accélération obtenu pour ce type d'applications parallèles est faible, proche de 1.5. Pour cela nous avons proposé la technique de synthèse de CSs pour améliorer le facteur d'accélération de ce type d'applications. Cette technique est présentée dans le chapitre suivant.

Chapitre 4

Techniques de synthèse et d'adaptation de taille des intervalles des applications

4.1 Introduction

Dans le chapitre précédent, nous avons montré que l'accélération de la simulation obtenue par la méthode AS est insuffisante ; quand les applications s'exécutant en parallèle ; ont des comportements différents. Cette combinaison d'applications concurrentes est nommée ici *"applications hétérogènes"*. Le facteur d'accélération de ces applications hétérogènes est faible car la détection de similarité de longues séquences de phases est difficile par AS.

Dans ce chapitre, nous présentons une technique complémentaire à AS permettant d'optimiser le facteur d'accélération même lorsque les séquences de phases sont longues. Cette technique [2] repose sur le fait que 2 phases, même si elles sont différentes en termes de BBV, leurs performances en termes d'IPC et d'EPC peuvent être identiques. Ainsi, durant la simulation, on s'autorise à ne pas simuler des CS lorsque la différence avec un CS simulé est faible. Dans la suite de ce chapitre, cette opération de remplacement d'une phase par une autre phase dans le processus de recherche sera appelée "synthèse de phases". Ainsi à partir de CS simulé on synthétise ou on représente des CS contenant des séquences de longueur identique mais contenant aussi quelques phases différentes. Le nombre de phases qui diffère avec le CS simulé dépend du pourcentage de synthèse autorisé. La question posée concerne l'influence de cette technique sur l'accélération de la simulation et sur la précision de l'estimation. La deuxième technique proposée ici concerne l'adaptation de la taille de phase afin de réduire le nombre de phases dans les séquences. Cette adaptation est basée sur le nombre de références mémoires de chaque application. Elle consiste à augmenter la taille des intervalles des applications ayant un nombre réduit de références mémoires. Comme on le verra dans ce chapitre, ceci réduit le nombre de phases dans les séquences, ce qui simplifie la détection de similarité entre les CSs générés par la méthode AS.

4.2 Technique de Synthèse de CSs

La technique de synthèse de CSs que nous présentons ici réduit le nombre de CSs simulés en diminuant le nombre de combinaisons de phases quand les applications sont hétérogènes. L'idée de la synthèse de CS consiste à considérer les CSs qui ont une faible différence au niveau des phases comme étant des CS similaires et donc possédant les mêmes performances. Autrement dit, une combinaison de phases est considérée similaire à un CS déjà simulé, donc présent dans la CST, si le pourcentage de phases différentes entre les séquences de phases adjacentes (au niveau du même processeur) est plus petit ou égal à un pourcentage de synthèse donné. Lorsqu'on applique la technique de synthèse de phases, l'ordre des phases dans les 2 CS est respecté. Pour un pourcentage de synthèse de $n\%$, les phases suivantes dans les traces sont similaires à celles d'un CS simulé, si le nombre de phases différentes entre les 2 séquences de phases, est plus petit ou égal à $(n/100) * l$, où l représente la taille de la séquence de phase. Ici l'ordre des processeurs et des phases est respecté.

Le premier tableau de la table 4.1, montre, pour 4 tailles différentes de séquences de phases, le nombre maximum de phases différentes entre 2 séquences qui peut être accepté. Par exemple, dans le cas du pourcentage de synthèse de 25% (n=25), pour une séquence de 4 phases (l=4) une seule phase au maximum ($(25/100) * 4$) peut être différente. Le deuxième tableau de la table 4.1 montre un exemple de synthèse de 25%. Ce tableau montre les 4 CSs qui ont été synthétisés du CS a,b-x,y,z,w avec un pourcentage de 25%. Les phases remplacées, entre le CS synthètisé et le CS simulé (a,b-x,y,z,w), sont représentées en gras dans le

Taille séq-phases \ Pourcentage de synthèse	25%	50%	75%
1	0	0	0
2	0	1	1
3	0	1	2
4	1	2	3

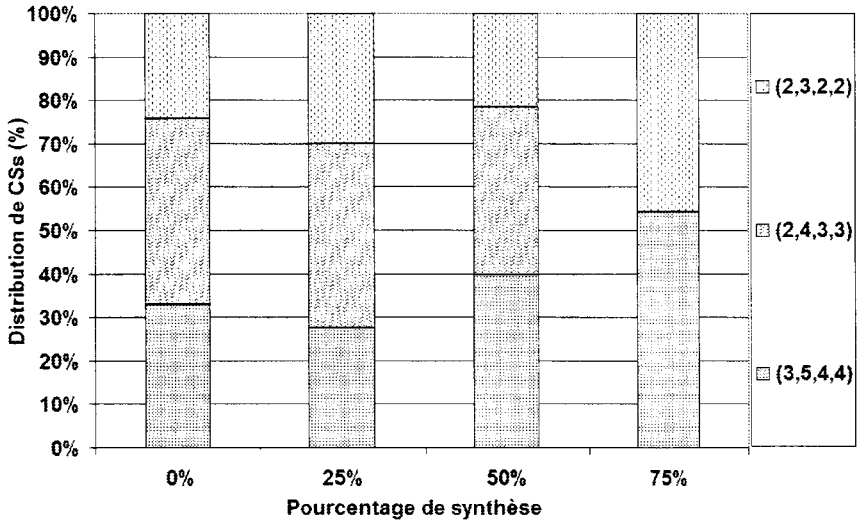
	CS simulé	CSs synthétisés			
séquence 1	a	a	a	a	a
	b	b	b	b	b
séquence 2	x	x	x	x	**y**
	y	y	y	**z**	y
	z	z	**w**	z	z
	w	**x**	w	w	w

TAB. 4.1 – Le premier tableau montre le nombre maximum de phases qui peut être remplacé dans une séquence de phases de taille donnée pour un pourcentage de synthèse donné. Le deuxième tableau montre quatre CSs synthétisés du CS ab-xyzw pour un pourcentage de synthèse de 25%. Le CS ab-xyzw contient deux séquences de phases ab et xyzw dont chacune est associée à un processeur.

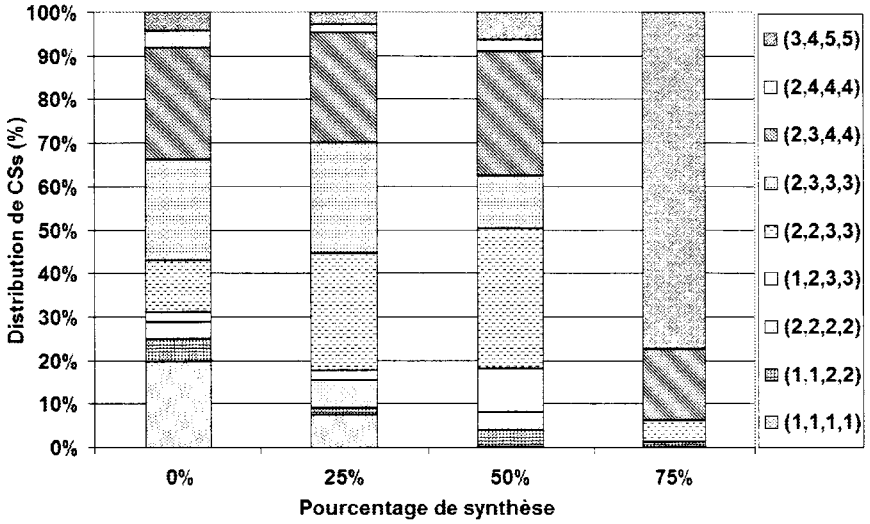
tableau. A titre d'exemple, au niveau du deuxiéme CS synthètisé (a,b-x,y,w,w), la séquence (x,y,w,w) différe de la séquence simulée (x,y,z,w) seulement par la phase w. On peut voir dans cet exemple que l'ordre des phases dans les séquences synthétisées est identique à celui dans les séquences du CS simulé (a,b-x,y,z,w).

La figure 4.1 montre la distribution en nombre de phases qui s'exécutent en parallèle dans les CSs pour trois pourcentages de synthèse 25%, 50% et 75%. Un pourcentage de 0% signifie que la synthèse n'est pas appliquée. Dans la figure 4.1(a) adcpm et (i)fft sont exécutées en parallèle sur 4 processeurs. Il y a trois groupes de CSs. Un groupe comprend tous les CSs ayant des longueurs identiques de séquences de phases. De même dans la figure 4.1(b) gsm et (i)fft sont exécutées sur 4 processeurs. Il y a 9 groupes de CSs. La figure 4.1 est présentée pour montrer que la disposition des phases entre les processeurs varie très peu quand le pourcentage de synthèse est inférieur à 50%. A l'opposé quand le pourcentage de synthèse est supérieur à 50% la disposition de phases entre les processeurs varie d'une façon remarquable. Le premier tableau 4.1 montre, dans le cas de synthèse de 75%, que le nombre de phases qui peuvent être remplacées est important d'où cette variation dans la disposition de phases entre les processeurs (voir figure 4.1).

La figure 4.2 montre, pour 2 applications concurrentes, les CSs générés avec une synthèse de 25% et de 50%. Pour une sytnèse de 25%, le nombre de phases dans la séquence doit être supérieur à 4 pour qu'au moins une phase soit remplacée (voir Le premier tableau 4.1). Ainsi dans la figure 4.2.a on peut voir qu'aprés les 2 CSs simulés, un troisième CS a été synthètisé du deuxième CS. Ce CS synthètisé comprend une séquence de 3 phases pour P0 et une séquence de 4 phases pour P1. En appliquant la synthèse de 50% sur les applications de la figure 4.2.a, la figure 4.2.b montre que les même CSs ont été simulés mais le troisième CS a été synthétisé du premier CS au lieu du deuxième CS. Ce CS synthétisé comprend ainsi une séquence de 2 phases pour P0 et une séquence de 3 phases pour P1. Comme on peut voir, les



(a) La distribution de CSs en nombre de phases pour adpcm&(i)fft sur 4 processeurs.



(b) La distribution de CSs en nombre de phases pour gsm&(i)fft sur 4 processeurs.

FIG. 4.1 – Distribution de CSs en nombre de phases pour deux combinaisons d'applications adpcm&(i)fft et gsm&(i)fft. Les applications sont exécutées sur 4 processeurs et le TWSB est fixé ici à 20%.

deux CSs synthètisés ont des longueurs de séquences de phases differentes. Ceci explique la variation de la disposition des phases entre les différents pourcentages de synthèse montrée dans la figure 4.1.

4.2.1 Effet de synthèse sur le facteur d'accélération

La figure 4.3 donne, pour chaque groupe d'applications la variation du facteur d'accélération avec 4 pourcentages de synthèse sur 4, 8 et 12 processeurs. Dans cette expérience,

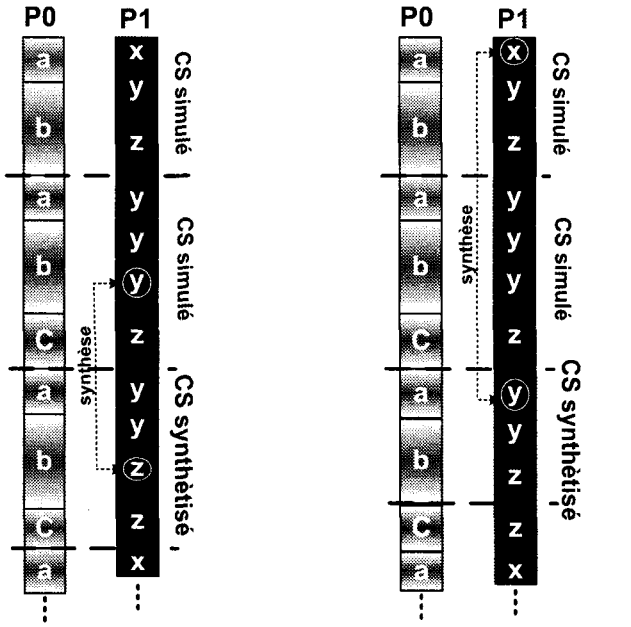


FIG. 4.2 – Pour une synthèse de 25%, le CS synthètisé comprend 3 phases pour P0 et 4 phases pour P1. Tandis que pour une synthèse de 50% le CS synthètisé comprend 2 phases pour P0 et 3 phases pour P1.

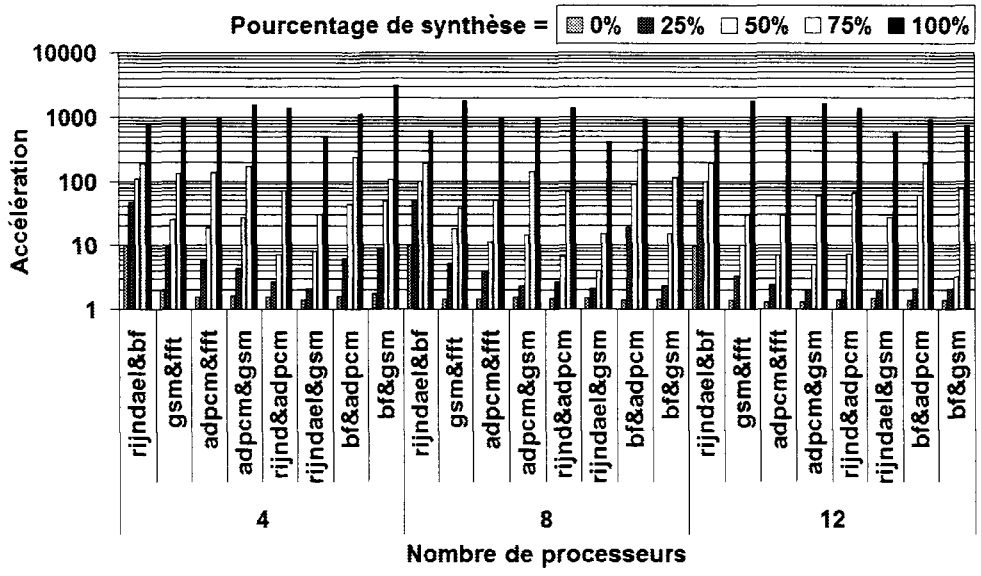


FIG. 4.3 – La variation de facteur de l'accélération (en log) avec les pourcentages de synthèse. Les groupes d'applications sont exécutés sur 4, 8 et 12 processeurs avec le TWSB fixé à 20%.

nous avons utilisé un TWSB de 20%. Les groupes d'applications sont ceux de la figure 3.8 où nous avons obtenu un facteur d'accélération très faible. Dans cette figure, le facteur d'accélé-

ration augmente avec le pourcentage de synthèse. Cette accélération est due à la diminution du nombre de combinaisons de phases qui provoque la diminution du nombre de CSs représentatifs.

Comme on peut le voir dans la figure 4.3, le facteur d'accélération est amélioré au moins de 5 fois par rapport au cas sans synthèse. Nous nous intéressons ici en particulier à la synthèse de 50% car la présicion de l'estimation est faible quand la synthèse est supérieure à 50%. Pour une synthèse de 50%, le facteur d'accélération est plus grand que 10 sur 4, 8 processeurs et plus grand que 5 sur 12 processeurs. Le plus faible facteur que nous ayons obtenu est pour rijndael&adpcm et rijndael&gsm. En comparant séparément rijndael respectivement à adpcm et à gsm, nous pouvons voir que rijndael a un nombre de défauts de cache relativement important. Ainsi les tailles de séquences de phases de adpcm et de gsm deviennent importantes quand ces applications s'exécutent en parallèle avec rijndael. Dans ces conditions, la séquence de phases de adpcm ou de gsm contient jusqu'à 20 phases (voir figure 3.15(b)). Comme les séquences de phases deviennent très longues, il en résulte une difficulté pour détecter la similarité entre ces séquences. Ce qui limite l'accélération de la simulation. Dans le cas de synthèse de 100%, nous permettons que toutes les phases entre les séquences de phases adjacentes soient différentes. Ainsi le premier CS généré va subir la simulation détaillée et va représenter le comportement total de l'application. Ceci rend le facteur d'accélération très important et l'erreur de la performance estimée inacceptable dans un DSE (voir section 4.2.2).

4.2.2 Effet de la synthèse sur l'erreur de l'estimation

Les expériences que nous avons réalisées montrent que l'erreur dans l'estimation de performance augmente avec l'augmentation du pourcentage de synthèse. Quand le pourcentage de synthèse devient important, les CSs considérés similaires peuvent contenir des phases ayant en réalité des comportements et des performances différents. En plus, il devient plus probable que les CSs non simulés ne vérifient pas la contrainte du TWSB déjà expliquée en section 3.4.1. En effet, la décision d'attendre sur une barrière de simulation est prise quand le temps d'attente estimé est inférieur au seuil TWSB fixé. Dans le cas de synthèse, il se peut que le temps d'attente des CSs synthétisés soit plus grand que le TWSB. Ce décalage apparaît en particulier quand le pourcentage de synthèse est important. Par conséquent, l'IPC estimé diminue à cause de l'intégration de cycles d'attente. De plus, la disposition de phases entre les applications varie avec l'augmentation du pourcentage de synthèse.

La figure 4.4 montre l'erreur dans l'estimation de l'IPC en fonction de 4 pourcentages de synthèse. Les applications exécutées sont ceux de la figure 4.3. La valeur du TWSB est fixée à 20%. Les barres qui dépassent le cadre du graphe, (erreur supérieure à 40%), ne sont pas totalement représentées. L'erreur des applications est calculée par rapport à l'IPC de cas réel où les applications sont exécutées en totalité. Comme nous l'avons expliqué ci-dessus, l'erreur augmente avec l'augmentation du pourcentage de synthèse. Il existe des cas où l'erreur varie irrégulièrement. C'est le cas de rijndael&gsm sur 4 processeurs et rijndael&blowfish sur 8 processeurs. Cette irrégularité est due à la compensation entre les deux sources d'erreurs : l'erreur de l'approximation de l'IPC des CSs par l'IPC des CSs représentatifs (erreur d'association) et l'erreur due à l'injection de barrières de simulation. Généralement, cette compensation est liée à l'erreur de l'approximation de l'IPC car l'erreur due aux barrières augmente générallement avec l'augmentation du pourcentage de synthèse. La figure 4.5 donne la variation des trois erreurs : erreur due à l'approximation, erreur due aux barrières

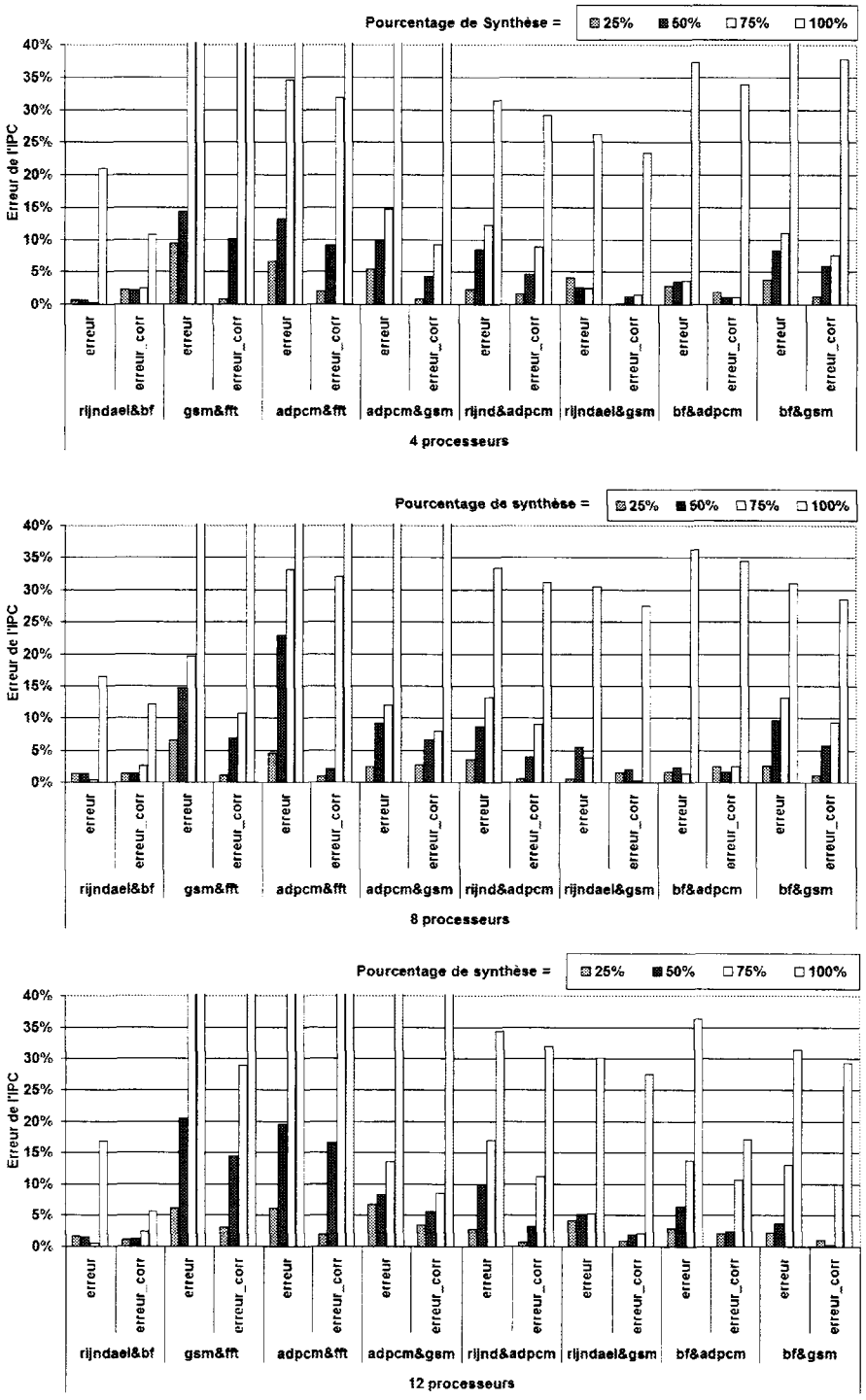


FIG. 4.4 – Variation de l'erreur sur l'estimation de l'IPC (avec et sans correction) avec 4 pourcentages de synthèse. Le TWSB est fixé ici à 20%.

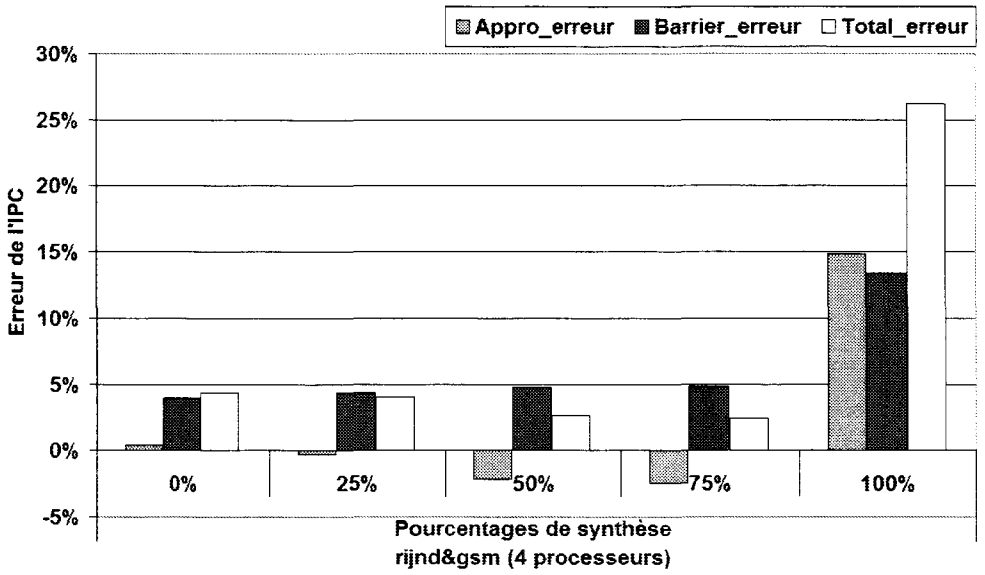


FIG. 4.5 – Variation avec le pourcentage de synthèse des trois types d'erreurs : erreur de l'approximation, erreur de barrière de simulation et l'erreur totale. Le TWSB est fixé à ici à 20%.

de simulation et l'erreur totale en fonction des pourcentages de synthèse. Comme on peut le voir, l'erreur de barrières augmente avec le pourcentage de synthèse tandis que l'erreur due à l'approximation varie irrégulièrement. Cette variation de l'erreur de l'approximation rend l'erreur totale irrégulière.

La figure 4.4 nous indique que l'erreur de synthèse est relativement importante en particulier pour des pourcentages de synthèse supérieurs à 75%. Dans ce cas, l'erreur dépasse les 20%. Comme expliqué précédemment, l'un des facteurs qui augmente l'erreur d'estimation est l'injection dans le simulateur de cycles d'attente au niveau des barrières de simulation. Généralement cette opération sous estime l'IPC.

En effet, le nombre moyen de cycles ajoutés par les attentes durant les barrières de simulation peut être mesuré. Cette valeur est donnée par la formule (4.1). Elle est notée par $Avr_{wait-cyc}$ et correspond au nombre de cycles d'attente injectés pour chaque CS (noté $wait - cyc_i$) multiplié par le nombre d'occurrences (noté $Freq_i$) du CS correspondant. Le nombre moyen de cycles d'attente est utilisé pour corriger l'IPC, réduisant ainsi l'erreur. A cet effet nous proposons la formule (4.2) qui estime l'IPC corrigé (noté par Cor_{IPC}) en soustrayant le $Avr_{wait-cyc}$ du nombre total de cycles (noté par Tot_{cycles}). Notons que la correction de l'IPC n'est pas appliquée dans le cas des applications homogènes (même application exécutée sur tous les processeurs), car l'erreur de l'IPC obtenue dans ce cas est relativement faible (voir la figure 3.8).

$$Avr_{wait-cyc} = \frac{(\sum_{i=1}^{Nb_{CS}} Est_{wait-cyc_i} * Freq_i)}{Nb_{proc}} \tag{4.1}$$

$$Cor_{IPC} = \frac{Total_{instr}}{(Tot_{cycles} - Avr_{wait-cyc})} \tag{4.2}$$

La figure 4.4 montre aussi l'erreur de l'IPC corrigé, pour chaque groupe d'applications

sur 4, 8 et 12 processeurs, en fonction des pourcentages de synthèse. La variation de l'erreur corrigée est similaire à celle de l'erreur sans correction. Celle-ci augmente avec l'augmentation du pourcentage de synthèse. En effet, l'erreur corrigée est corrélée avec l'erreur de l'approximation de l'IPC. Cette dernière augmente avec le pourcentage de synthèse due à la différence entre l'IPC de la phase simulée et les IPCs de phases synthétisées. D'autre part, l'erreur corrigée des groupes rijndael&bf sur 4, 8 et 12 processeurs est légèrement plus grande que l'erreur sans correction. Ce décalage est due à la compensation entre l'erreur de l'approximation et l'erreur de barrières. Ainsi en éliminant la moyenne de cycles d'attente (voir formule (4.2)), l'erreur corrigée devient plus grande que l'erreur sans correction. Cette situation se produit quand l'erreur due aux barrières est relativement négligeable.

La réduction de l'erreur varie entre 29% et 99% pour une synthèse de 25%. Le poucentage de 50% est le pourcentage le plus elevé où le facteur d'accéleration le plus important peut être obtenu tout en garantissant la précision de la performance estimée, erreur généralement proche de 10%. Pour un tel pourcentage de synthèse, la réduction de l'erreur varie entre 37% et 59% pour les applications considérées. Ainsi l'erreur corrigée est inférieure à 10% quand une synthèse de 50% est appliquée (voir figure 4.4). Rappelons que nous avons utilisé un TWSB de 20%. De plus, en appliquant la synthèse, l'erreur peut être diminuée en choisissant un TWSB plus petit.

En effet, une amélioration de la précision de synthèse peut être obtenue sans sacrifier l'accélération. Ceci est effectué en réalisant une analyse au niveau des phases afin de détecter celles qui ont une performance proche avant d'appliquer la synthèse. Ainsi seule les phases considérées proches peuvent être remplacées. Cette analyse qui rend la synthèse plus performante est proposée dans les perspectives de cette thèse. Cette analyse n'a pas été effectuée en cours de cette thèse car d'une part, les contraintes du temps ne l'ont pas permise et d'autre part car la méthode que nous présenterons dans le chapitre suivant satisfait tous les besoins concernant l'accélération et la précision.

4.2.3 Surcharge du travail de AS dans le cas de synthèse

Comme nous l'avons expliqué dans la section 3.5.8, le temps additionnel induit par l'utilisation de AS dépend du nombre d'accès à la table CST. Dans le cas de l'utilisation de AS, nous avons constaté que l'utilisation d'une clé associée à une liste de CSs réduit le nombre d'accès au CST (voir section 3.5.8). Tandis que dans le cas de synthèse, cette clé n'est pas utilisée. La figure 4.6 montre les clés des 2 CSs, CS_1 et CS_3. Pour un pourcentage de 50%, CS_3 a été synthètisé à partir du CS_1 malgré le fait que ces 2 CSs aient des clés différentes. Ceci explique le fait que la clé ne soit pas utilisée dans le cas de synthèse. En effet, les prochaines phases dans les traces doivent être comparées à chaque CS de CST. Ainsi pour chaque comparaison des prochaines phases avec les CSs de CST, le nombre maximum d'accès correspond au nombre de CSs dans la CST. Ceci va augmenter l'overhead de synthèse.

La figure 4.7 compare le temps d'exécution induit par As (avec un pourcentage de synthèse égal à 0%) avec le temps d'éxecution lorsque la synthèse est utilisée pour les trois pourcentages : 25%, 50% et 75%. Les groupes d'applications utilisés dans la figure 4.7 sont ceux de la section 3.5.8. Ce temps correspond au nombre de cycles consommés sur la machine hôte. Il est obtenu grâce à l'outil Perfmon [34]. Nous constatons que l'overhead de AS avec la synthèse est plus grand que celui de AS sans synthèse. Néanmoins ce temps décroît avec l'augmentation du pourcentage de synthèse. Cette baisse est due à la diminution du nombre de CSs présents dans la CST avec l'augmentation du pourcentage de synthèse.

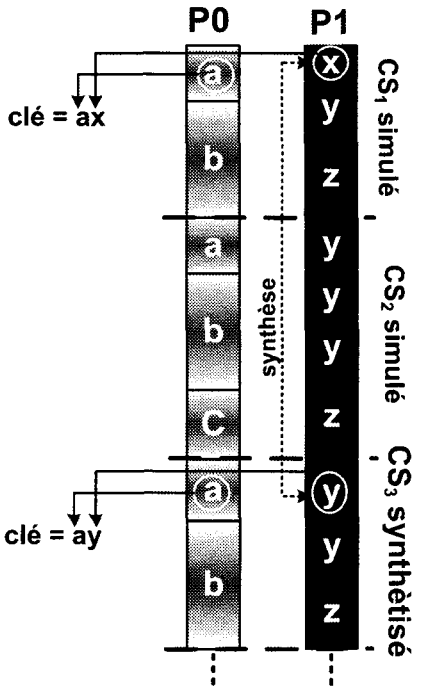


FIG. 4.6 – Les clés des CS_3 et CS_1 sont différentes. CS_3 est synthétisé du CS_1 pour un pourcentage de 50%.

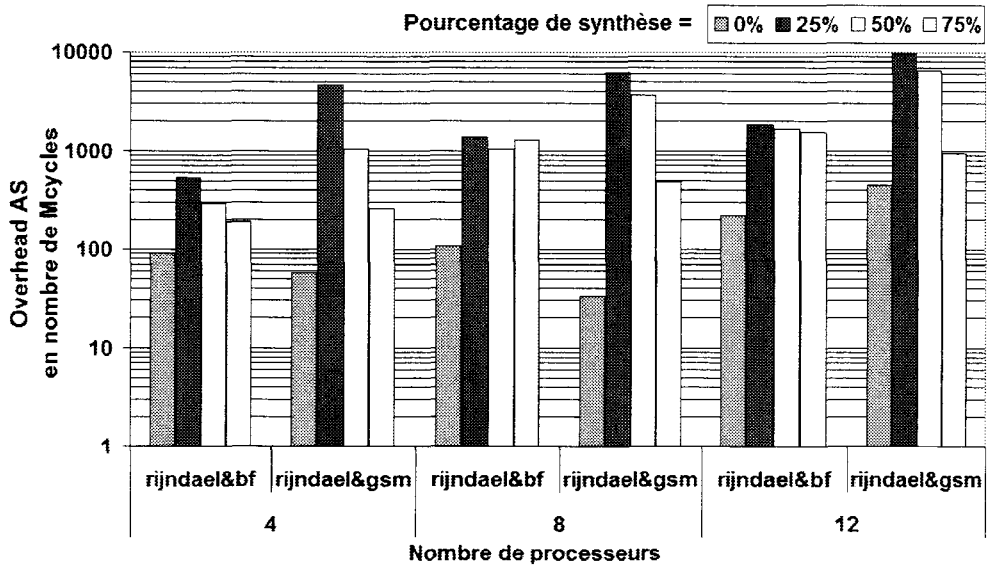


FIG. 4.7 – Comparaison du nombre de cycles (en log), sur l'ordinateur hôte, consommés par AS pour 3 pourcentages de synthèse. Le TWSB fixé à 20%.

Malgré l'augmentation de l'overhead avec la synthèse, ce temps reste négligeable par rapport au temps nécessaire à la simulation. La figure 4.8 montre que le temps de simula-

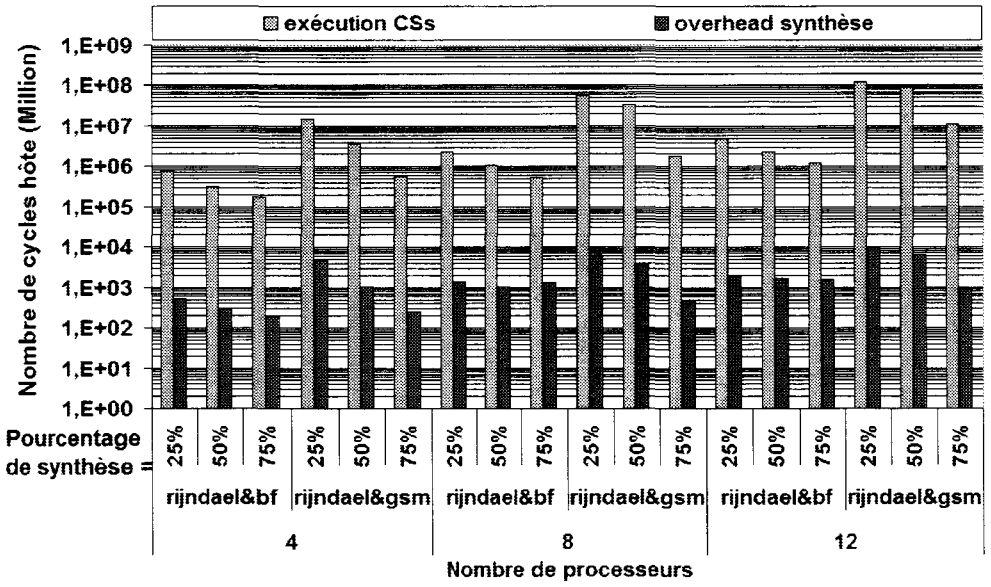


FIG. 4.8 – Nombre de cycles (en log) consommés par la simulation (exécution des CSs) et par l'overhead de synthèse. Le TWSB est fixé à 20%.

tion des CSs représentatifs est largement plus grand que l'overhead induit par AS ou par la synthèse.

4.3 Adaptation des tailles des intervalles des applications concurrentes

Dans le cas des systèmes multiprocesseurs à mémoire partagée, le réseau d'interconnexion, le bus en occurrence, représente la ressource pour laquelle les processeurs rentrent en compétition pour accéder à la mémoire partagée. Par conséquent, à travers ce bus partagé, une interaction au niveau des performances des processeurs est constatée. Ainsi, les performances d'un processeur dépendent des activités des autres processeurs. Dans le cas de notre méthode d'échantillonnage, cette interaction entre processeurs peut réduire le facteur d'accélération. En effet, lorsque les applications exécutées sont hétérogènes, les CSs obtenus ne sont pas équilibrés en termes de nombre de phases. Les processeurs qui réalisent un grand nombre de lectures/écritures mémoire ont des séquences relativement courtes (1 phase par CS) alors que les autres processeurs ont des séquences très longues. Plus précisément, les applications avec un faible nombre de défauts cache auront de longues séquences de phases alors que celles possédant un nombre de défauts cache important auront des séquences courtes. Par conséquent, comme la probabilité de retrouver de longues séquences de phases est faible, l'accélération obtenue est réduite. Dans la section 3.5.7 du chapitre précédent, nous avons montré que les séquences de phases de adpcm et de gsm contiennent jusqu'à 20 phases lorsque l'une de ces applications est exécutée en parallèle avec rijndael qui a un nombre important de défauts cache. Dans ces conditions, le facteur d'accélération est faible.

Pour résoudre ce problème, nous avons proposé la technique de synthèse de CSs. Néan-

moins pour quelques applications, même avec un pourcentage de synthèse de 50%, le facteur d'accélération reste faible quand les tailles de séquences de phases sont importantes. Le facteur d'accélération de rijndael&adpcm et de rijndael&gsm sont deux illustrations de cette situation (voir figure 4.3). Nous déduisons que la technique de synthèse est moins performante quand les séquences de phases sont longues. Autrement dit, cette technique ne permet pas la diminution du nombre de phases dans les séquences de CSs générés. Ce qui provoque une faible accélération de simulation. Pour améliorer le facteur d'accélération en diminuant le nombre de phases dans les séquences, nous proposons une autre technique. Cette dernière est aussi complementaire à AS. L'idée consiste à adapter les tailles des intervalles des applications concurrentes en utilisant le nombre d'accès au bus. Ainsi pour obtenir des séquences de phases ayant un nombre réduit de phases, la taille des intervalles utilisés lors de la phase de génération de traces de phases n'est pas fixe (identique) pour toutes les applications. Ainsi des intervalles relativement grands sont associés aux applications avec de faibles nombres de défauts cache tandis que des tailles réduites sont associées à celles ayant un nombre important de défauts cache. Cette technique est nommée *"Adaptation des tailles des intervalles"*.

Le nombre d'accès mémoire est ainsi détecté pour chaque application concurrente. Ainsi les applications ayant des comportements différents du point de vue accès mémoire auront des tailles des intervalles différentes. Idéalement, l'adaptation des tailles des intervalles doit utiliser le nombre de défauts cache. Mais l'obtention de ces facteurs nécessite une connaissance à priori des configurations des caches de processeurs. Ce qui n'est pas possible dans une opération de DSE et augmente l'overhead de cette méthode. Afin d'explorer plus de configurations dans le DSE et afin de réduire l'overhead de la méthode, nous proposons l'utilisation de références mémoires, c'est à dire le nombre d'instructions "load" et "store". Ce nombre est utilisé comme un indicateur du nombre de défauts cache. Cette solution est facile à réaliser puisque le nombre de références mémoire est obtenu facilement durant la génération des traces de phases. Comme on peut le voir dans la figure 4.9, dans la plupart des applications testées, le nombre de références mémoires est corrélé avec le nombre de défauts cache (voir la figure 4.9). Une extension possible de cette technique pour améliorer la précision est d'utiliser les notions de localité ou le "working set" [16, 35] comme un indicateur de défauts cache. En se basant sur les instructions mémoires (load et store) exécutées, les tailles des intervalles des applications concurrentes sont adaptées.

$$AMRI = \frac{(Total_{Accs\_mmoire} * Inter_{taille})}{Total_{Inst}} \tag{4.3}$$

$$ACMI = \frac{(Total_{Defauts\_cache} * Inter_{taille})}{Total_{Inst}} \tag{4.4}$$

Pour chaque application à exécuter, les deux formules 4.3 et 4.4 sont évaluées pour estimer respectivement le nombre moyen de références mémoire, nommé AMRI pour *"Average Memory References per Interval"*, et le nombre moyen de défauts cache nommé ACMI pour *"Average Cache Misses per Interval"*. Les quatre valeurs $Total_{Acces\_mmoire}$, $Inter_{taille}$, $Total_{Inst}$ et $Total_{Defauts\_cache}$ correspondent respectivement aux : nombre total de références mémoire, taille d'un intervalle donné, nombre total des instructions exécutées et le nombre total de défauts cache.

La figure 4.9 montre les AMRI et ACMI des applications pour des tailles d'intervalles de 50K instructions. Les deux courbes sont assez parallèles. Ainsi notre approximation, basée

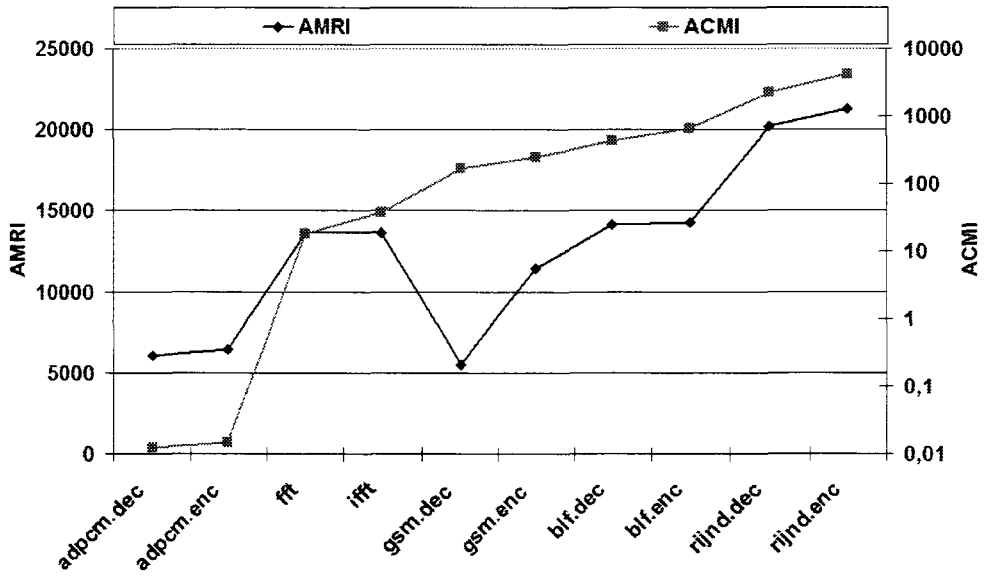


FIG. 4.9 – Nombre moyen de références mémoire (AMRI) et le nombre moyen de défauts cache de données et d'instructions (ACMI) par un intervalle de 50K instructions.

sur les AMRIs, pour adapter les tailles des intervalles semble assez précise.

L'idée est d'adapter les tailles des intervalles des applications concurrentes de telle sorte que ces intervalles contiennent approximativement le même nombre de références mémoire. Ainsi, la formule (4.5) est utilisée pour calculer la taille des intervalles (notée $Inter_{taille}$) pour chaque application concurrente. L'indice "a" correspond à l'application ayant le plus grand AMRI. La valeur "$Inter_{taille}(a)$" correspond à la taille des intervalles de l'application "a" et elle est égale à 50K instructions dans notre cas.

$$Inter_{taille} = \frac{AMRI(a)}{AMRI} * Inter_{taille}(a) \quad \{a = application\ avec\ le\ plus\ grand\ AMRI\} \tag{4.5}$$

La technique d'adaptation des tailles des intervalles a été appliquée aux combinaisons d'applications hétérogènes suivantes : rijndael&gsm, rijndael&adpcm, blowfish&adpcm et blowfish&gsm (voir tableau 4.2). Comme nous l'avons montré précédemment, ces combinaisons ont le plus faible facteur d'accélération dans le cas de synthèse.

4.4 Effet de l'adaptation des tailles des intervalles sur l'accélération

La figure 4.10 montre la variation du facteur d'accélération, quand la technique de l'adaptation des tailles est appliquée, en fonction de quatre valeurs du TWSB. Comme expliqué précédemment, on s'intéresse ici aux combinaisons d'applications hétérogènes. Les applications de la figure 4.10 sont exécutées avec des tailles des intervalles différentes en se basant sur la procédure de l'adaptation des tailles. Ainsi ces tailles des intervalles sont celles données par le tableau 4.2. Le facteur d'accélération avec l'adaptation des tailles des intervalles

Combinaisons des applications	Applications	Tailles d'intervalles
rijnd&adpcm	rijnd	50k instructions
	adpcm	170k instructions
rijnd&gsm	rijnd	50k instructions
	gsm	200k instructions
bf&adpcm	bf	50k instructions
	adpcm	120k instructions
bf&gsm	bf	50k instructions
	gsm	140k instructions

TAB. 4.2 – Tailles des intervalles obtenues en appliquant la formule (4.5).

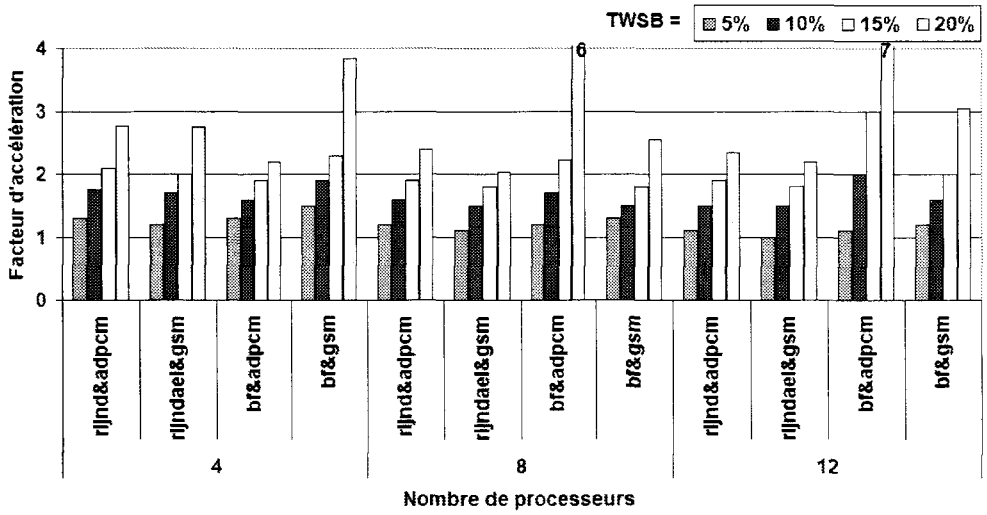


FIG. 4.10 – Accélération obtenue avec l'adaptation des tailles d'intervalles des applications.

est légèrement plus grand que celui dans le cas avec AS (voir la figure 3.14). Il est généralement supérieur à 2 pour un TWSB de 20%. Par comparaison du facteur d'accélération de la technique de synthèse montré dans la figure 4.3, le facteur d'accélération obtenu par l'adaptation des tailles est plus petit. Cela est vrai même dans le cas d'un pourcentage de synthèse de 25%.

Il devient par conséquent intéressant d'étudier la possibilité de combiner les deux techniques de synthèse et d'adaptation.

La figure 4.11 montre la variation de l'accélération de la simulation, quand l'adaptation des tailles et la synthèse sont appliquées simultanément, en fonction de 4 pourcentages de synthèse. Les applications et leurs tailles des intervalles sont les mêmes que celles de la figure 4.10. Le TWSB est fixé à 20%. La figure 4.11 montre que le facteur d'accélération augmente avec l'augmentation du pourcentage de synthèse.

La figure 4.12 montre le nombre maximal de phases dans les séquences pour les applications exécutées en parallèle sur 4 processeurs. Les applications et les tailles d'intervalles adaptées sont les mêmes que celles de la figure 4.11. Le TWSB est fixé à 20%. Comme on peut le voir dans la figure 4.12, le nombre de phases dans les séquences a bien été réduit

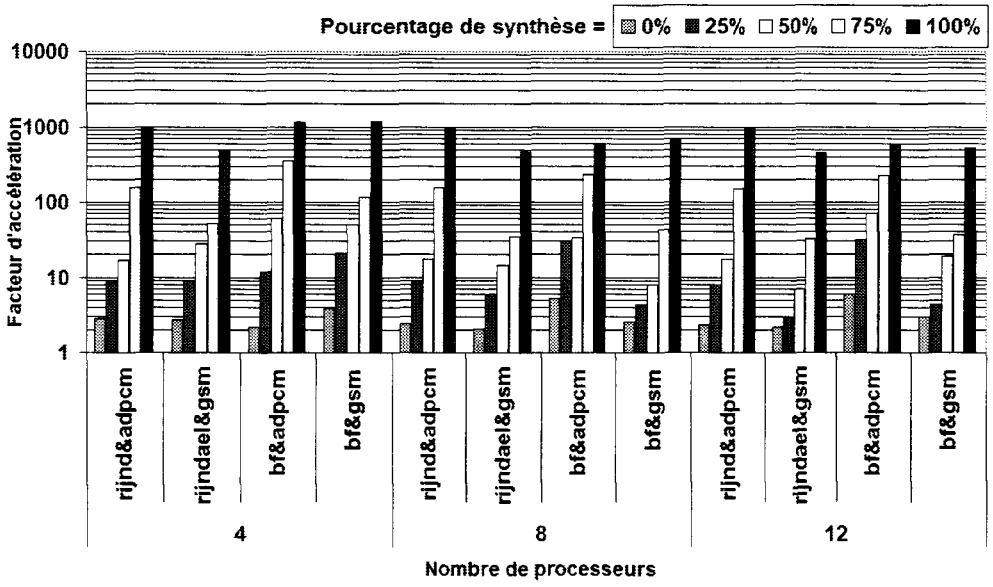


FIG. 4.11 – Variation du facteur d'accélération (en log) avec différents pourcentages de synthèse (TWSB égale à 20%). La taille des intervalles est celle du tableau 4.2.

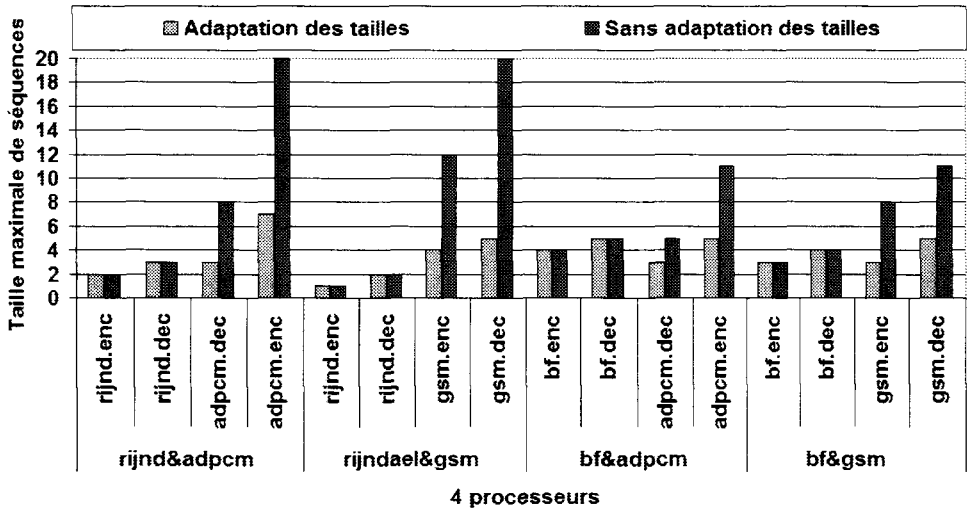


FIG. 4.12 – Nombre maximal de phases dans les séquences pour 2 cas, avec et sans adaptation des tailles d'intervalles. Les tailles adaptées sont celles du tableau 4.2. Le TWSB est fixé à 20%.

par l'adaptation des tailles des intervalles. Par exemple, pour rijndael&gsm, les séquences de phases de gsm passent de 20 phases à 5 phases par séquence. Rappelons que cette baisse réduit le nombre de combinaisons de phases et par conséquent, augmente la probabilité de retrouver des CSs simulés. Nous remarquons aussi que le nombre maximal de phases pour les applications rijndael et blowsish est équivalent dans les 2 cas, avec et sans adaptation. Comme ces 2 applications ont relativement le plus grand nombre de défauts cache, la taille de leurs intervalles n'a pas été augmentée par l'adaptation et elle est la même dans les 2 cas,

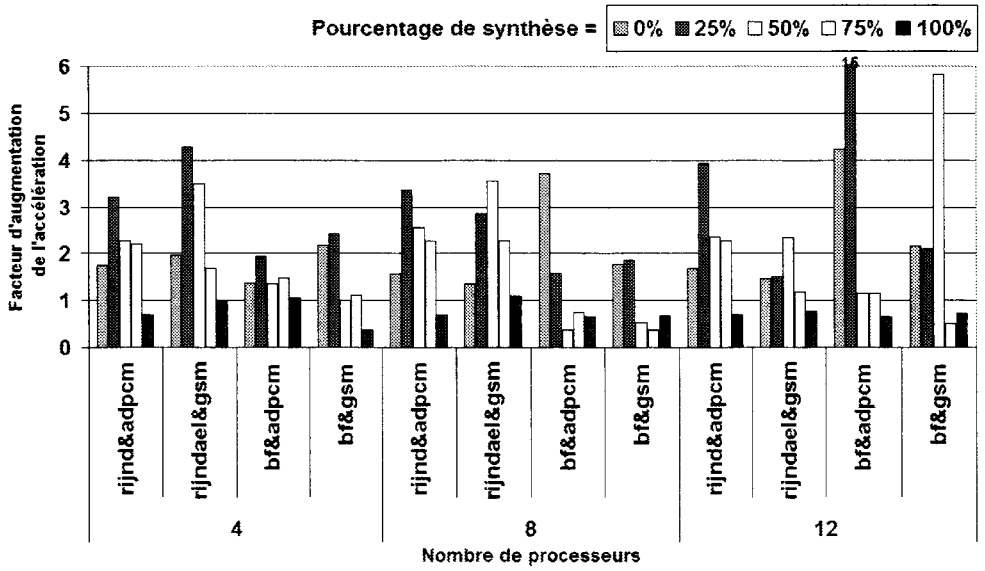


FIG. 4.13 – Ratio (accélération avec adaptation des tailles + synthèse) / (accélération sans adaptation des tailles + synthèse).

avec et sans adaptation des tailles.

La figure 4.13 montre le ratio du facteur d'accélération, dans le cas de l'adaptation des intervalles par rapport au facteur d'accélération dans le cas sans adaptation. Ce ratio a été fait pour 4 pourcentages de synthèse sur 4, 8 et 12 processeurs. Comme on peut le voir, le facteur d'accélération est plus grand quand l'adaptation des tailles est appliquée. La grandeur du facteur d'accélération varie d'un pourcentage de synthèse à l'autre et d'un groupe d'applications à l'autre (voir la figure 4.13).

Quand l'adaptation des tailles est appliquée sans la synthèse (pourcentage de synthèse égal à 0%), l'optimisation moyenne en accélération varie entre 60% et 211%. Cette optimisation moyenne atteint 500% et 213% respectivement pour une synthèse de 25% et de 50%. De plus pour les 2 pourcentages de synthèse 50% et 75%, le facteur d'accélération est plus petit dans le cas de l'adaptation des tailles pour bf&adpcm sur 8 processeurs et pour bf&gsm sur 8 et 12 processeurs (le ratio corespondant est plus petit que 1 dans la figure 4.13). En appliquant l'adaptation des tailles d'intervalles sur ces deux combinaisons d'applications, le nombre de phases dans les séquences passe sous la barre de 3 phases dans la plupart des CSs, ce qui réduit le nombre de phases qui peuvent être synthétisées dans ces applications. Ceci explique la baisse de l'accélération de simulation pour ces 2 combinaisons d'applications avec l'adaptation des tailles. Mais, malgré la baisse, cette accélération reste importante, elle atteint 33 pour bf&adpcm sur 8 processeurs et 37 pour bf&gsm sur 12 processeurs (voir figure 4.11). Pour une synthèse de 100%, il n'y a pas d'augmentation de l'accélération avec l'adaptation des tailles car quelque soit la taille de l'intervalle, seul le premier CS subit la simulation détaillée.

Nous pouvons déduire de tous les résultats montrés précédemment que AS est plus performante dans le cas des applications hétérogènes quand elle est appliquée en combinaison avec les techniques de synthèse et de l'adaptation des tailles des intervalles. Cette performance est du point de vue facteur d'accélération.

4.5 Effet de l'adaptation des tailles des intervalles sur l'erreur de l'estimation

La figure 4.14 montre, pour chacune des applications de la figure 4.11, l'erreur sur l'estimation de l'IPC avec et sans correction (formule 4.2) en fonction de 4 pourcentages de synthèse. Le TWSB étant égal à 20%. Les barres qui dépassent des graphes ne sont pas totalement présentées. Nous rappelons que la formule (4.2) de correction de l'IPC élimine le nombre moyen de cycles d'attente estimé du nombre total de cycles estimé. La figure 4.14 montre une augmentation de l'erreur avec l'augmentation du pourcentage de synthèse. Cette augmentation a été expliquée dans la section 4.2.2. L'irrégularité est due à la compensation entre l'erreur de l'approximation de l'IPC et l'erreur due aux barrières. Ceci a été expliqué dans la section 4.2.2. Cette compensation est liée à l'erreur de l'approximation de l'IPC car l'erreur due aux barrières augmente généralement avec l'augmentation du pourcentage de synthèse. Elle apparaît quand l'erreur de barrière augmente légèrement avec le pourcentage de synthèse (erreur de barrières est inférieure à 1%). De plus, cette compensation rend parfois l'erreur de l'IPC faible pour une synthèse de 75%.

Dans le cas de synthèse de 50%, l'erreur de l'IPC est acceptable, elle est inférieure à 10%. Ceci est due au nombre réduit de phases remplacées durant la synthèse. En fait, le nombre de phases exécutées dans les séquences de phases diminue en augmentant les tailles des intervalles. Ce qui provoque une diminution du nombre de phases qui peuvent être remplacées pour un pourcentage de synthèse donné. Malgré le nombre réduit du nombre de phases remplacées, l'erreur dépasse 10% pour un pourcentage de synthèse de 75% (voir figure 4.14).

De plus, la figure 4.14 montre que l'erreur avec correction (nommée "cor-error") est plus grande que l'erreur sans correction. Ainsi la formule de correction n'est pas performante dans le cas de l'adaptation des tailles des intervalles. En effet, les expériences menées montrent que cette adaptation diminue le nombre de phases dans les séquences mais au même temps augmente le nombre d'instructions exécutées dans ces séquences. Cette augmentation est due à l'accroissement des tailles d'intervalles. Ainsi les processeurs qui exécutent les séquences de phases dont la taille a été augmentée vont finir plus tôt leur exécution. Comme dans notre cas, la simulation se termine quand au moins un processeur achève son exécution, ainsi l'IPC total est surestimé malgré l'injection des cycles d'attente aux barrières de simulation. De même la formule (4.2) qui corrige l'IPC surestime à son tour cet IPC en éliminant les cycles d'attente des barrières. Ainsi l'IPC corrigé sera largement surestimé, ce qui va augmenter l'erreur. Ceci explique le fait que l'erreur avec correction soit plus grande que l'erreur sans correction.

4.6 Conclusion

Dans ce chapitre, nous avons tenté de résoudre les problèmes posés par la méthode AS. Nous avons expérimenté deux solutions. En effet quand AS est appliquée, le nombre de phases augmente dans les séquences, en particulier quand les applications sont hétérogènes. Ceci augmente le nombre de combinaisons de phases et réduit l'accélération.

Pour résoudre ce problème, une technique complémentaire à AS a été proposée. Cette technique améliore l'accélération de la simulation par la synthèse d'une séquence en plusieurs séquences semblables. L'accélération augmente alors jusqu'à 100 fois tout en gardant une erreur inférieure à 10%.

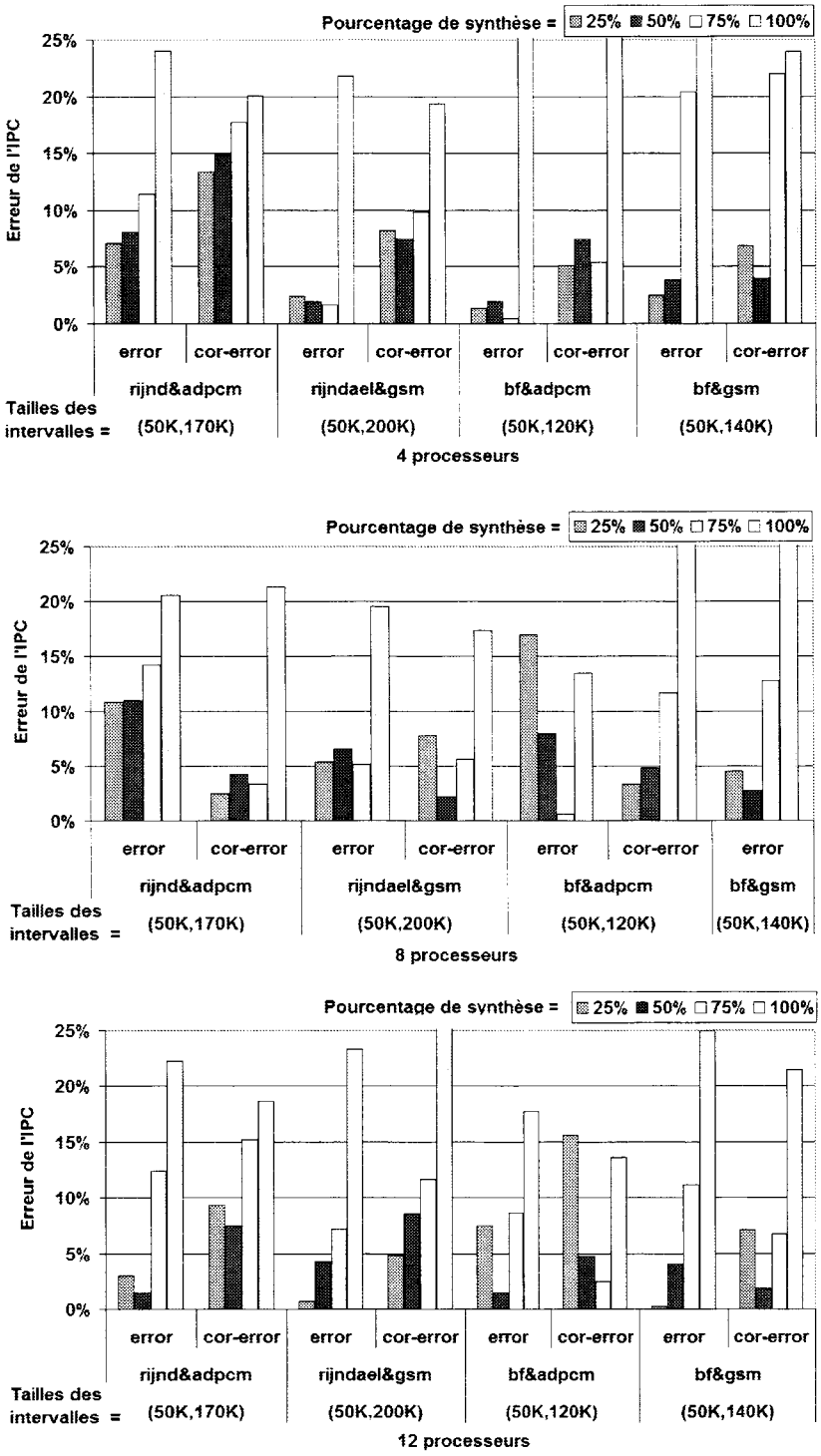


FIG. 4.14 – Variation de l'erreur sur l'estimation de l'IPC (avec et sans correction) avec 4 pourcentages de synthèse. TWSB est fixé à 20% et la taille des intervalles est celle du tableau 4.2.

Malheureusement, la synthèse de phases offre des performances limitées lorsque la taille des séquences de phases devient importante. Dans ce cas, même en utilisant la synthèse de phases l'accélération ne dépasse pas la valeur 10.

Pour résoudre cette nouvelle problématique, nous avons proposé une technique qui fixe la taille des intervalles des applications de façon "intelligente". Ainsi au lieu d'avoir une taille identique pour toutes les applications, une taille est déterminée afin d'avoir des CSs avec un nombre réduit de phases. La méthode que nous avons proposée n'exige pas des temps d'exécution importants. Néanmoins, même si deux techniques améliorent le facteur d'accélération de la plupart des combinaisons d'applications, les performances obtenues dans certains cas ne sont pas excellentes. C'est notamment le cas lorsque les applications gsm et rijndael sont exécutées de façon concurrentes. Ceci est l'objet du chapitre suivant.

Chapitre 5

Échantillonnage par Multi-Granularité

5.1 Introduction

Dans ce chapitre, nous nous intéressons à l'accélération de la simulation sur architectures MPSoC d'applications parallèles hétérogènes. Par hétérogénéité nous désignons des applications concurrentes ayant des comportements différents et des besoins différents en ressources plutôt que le changement entre les phases au niveau de la même application. Comme cela a été expliqué dans le chapitre précédent, l'accélération de la simulation de ces applications pose plusieurs problèmes. Les besoins en ressources matérielles, en mémoire en particulier, de ces applications varient d'une application à l'autre. Par conséquent, les CSs obtenus dans AS sont déséquilibrés, augmentant d'une part le nombre de combinaisons de phases et réduisant d'autre part l'accélération. En effet, les séquences de phases sont, dans ce cas, très longues (voir section 3.5.7). Comme AS est très conservatrice dans la détection de similarités entre les CSs alors la détection des scénarios répétitifs devient difficile. De même les deux techniques de synthèse de CSs et de l'adaptation des tailles des intervalles des applications proposées précédemment optimisent le facteur d'accélération dans le cas des applications hétérogènes. Mais ces deux techniques nécessitent une connaissance approfondie des applications étudiées. La technique de synthèse nécessite l'analyse des phases afin de détecter celles qui ont une performance proche avant d'appliquer la synthèse. La technique de l'adaptation des tailles, figée, consiste à adapter la taille des intervalles analytiquement en considérant que ces derniers ont le même nombre de références mémoire. Ce qui n'est pas toujours vrai dans l'exécution réelle. De plus, l'accélération obtenue par ces 2 techniques n'est pas excellente dans certains cas. Dans le cas de la méthode CoPhase, le nombre de recouvrements de phases augmente avec le nombre de processeurs. Ceci a pour conséquence d'augmenter le nombre de scénarios de phase conjoints (voir section 2.4.2 et 2.7) et de diminuer le facteur d'accélération dans le cas des applications hétérogènes. Dans ce chapitre, nous détaillons la méthode d'échantillonnage par multi-granularité nommée MGS pour "*Multi-Granularity Sampling*" [6]. Cette méthode simplifie la détection de scénarios répétitifs des CS. En se basant sur le comportement dynamique des applications s'exécutant en parallèle, MGS choisit dynamiquement parmi plusieurs granularités de phases, ce qui permet d'augmenter l'accélération. En effet, pour réduire la taille des séquences de phases, MGS réduit le nombre de phases au minimum, à savoir une phase pour chaque application. Ainsi dans chaque recouvrement de phases généré, une seule phase avec une granularité appropriée est choisie par processeur. Ceci va faciliter la détection des recouvrements répétés de phases. Contrairement à d'autres méthodes, un seul échantillon de chaque recouvrement de phases est suffisant pour obtenir un niveau acceptable de précision et il n'est donc pas nécessaire de resimuler à nouveau le même recouvrement.

5.2 Échantillonnage avec multi-granularité (MGS)

La méthode MGS, que nous présentons dans ce chapitre, réduit la taille des séquences de phases de notre méthode d'accélération par échantillonnage. Cette méthode est simple à mettre en oeuvre et permet la détection des recouvrements qui se répètent avec une faible erreur d'estimation.

Comme il a été vu précédemment, la taille des séquences de phases augmente lorsque les applications concurrentes ont un comportement différent. La détection de similarité entre ces longues séquences de phases devient très difficile, d'où un faible facteur d'accélération.

Contrairement à AS, la méthode MGS forme des séquences avec une phase par application concurrente quelque soit le nombre d'instructions exécutées durant la même période de cycles de simulation. Chaque échantillon correspondant à une combinaison d'une phase par application est appelé MPC pour *"Multi-Phase Cluster"*. Cette combinaison facilite la détection des recouvrements de phases qui se répètent, d'où une simplicité de mise en oeuvre et une augmentation de l'accélération. Ainsi, MGS répond au besoin d'exécuter différents nombres d'instructions de différentes applications concurrentes durant une même période de cycles de simulation. Le temps de simulation sera largement réduit car toutes les phases réelles répétitives seront détectées. Ainsi la méthode MGS est fortement convenable pour les systèmes embarqués dont les applications concurrentes ont des comportements extrêmement différents. De plus, l'utilisation de MGS n'exige pas une connaissance à priori de l'application. Dans ce chapitre, nous désignons par granularité d'un intervalle le nombre d'instructions qui compose l'intervalle. Pratiquement, la granularité la plus fine (ou d'ordre 1) correspond à une taille de 50K instructions. Des granularités plus importantes (ou d'ordre > 1) peuvent être obtenues en considérant des intervalles de tailles plus grandes. Ainsi une granularité d'ordre N consiste à considérer des intervalles avec des tailles de $(50 * N)$K instructions. La méthode MGS se déroule en deux étapes détaillées dans les sections suivantes.

5.2.1 Première étape : Création d'une matrice de phases par programme

Cette étape est réalisée une seule fois durant l'opération de DSE. Elle commence par une génération des traces de phases dites *"traces d'indentificateurs de phases"*. Une trace est générée pour chaque application individuellement en utilisant une simulation fonctionnelle. Ces traces ne dépendent ni des applications concurrentes qui s'exécutent sur les autres processeurs ni de la configuration architecturale du MPSoC.
Cette première étape est réalisée comme suite : L'application est décomposée en intervalles de granularité d'ordre 1. La taille de ces intervalles, référencée par la lettre t, est la plus petite des tailles analysées des intervalles. La valeur de t vaut 50K instructions dans les expériences (voir les explications dans la section 5.4.1). Pour chaque intervalle, le vecteur de blocs de base BBV (*Basic Block Vector*) contenant les fréquences de blocs de bases exécutés dans l'intervalle correspondant est formé. La trace de phases est alors générée en examinant la similarité entre ces BBVs grâce à l'outil de classification de SimPoint [57]. Comme dans le cas de AS, tout autre outil de classification peut également être utilisé [37].

En utilisant le même point de départ, c'est-à-dire à chaque point de discrétisation de t instructions, des intervalles de plus grosse granularité sont formés. Durant nos expériences, nous avons considéré que t vaut 50K instructions. Les intervalles de plus grosse granularité ont respectivement pour tailles 100K, 150K, etc. Après la formation des intervalles de toutes les granulartités, Simpoint est successivement utilisé afin de détecter la similarité et générer les traces de phases pour chaque ordre de granularité.
Pratiquement, pour une granularité d'ordre g, un BBV de granularité d'ordre 1 est combiné avec les g-1 BBVs qui lui succèdent en additionnant les fréquences des BBVs. Les nouveaux BBVs, correspondant à des intervalles de g*t instructions, vont alors subir la procédure de classifications de phases. Ainsi une trace de phases est générée pour chaque ordre de granularité.

La figure 5.1.A montre la décomposition de l'application a en intervalles pour chacun des quatre premiers ordres de granularité (1 à 4). Pour la granularité d'ordre 1, l'application a est décomposée en cinq intervalles. Ensuite pour la granularité d'ordre 2, chaque intervalle

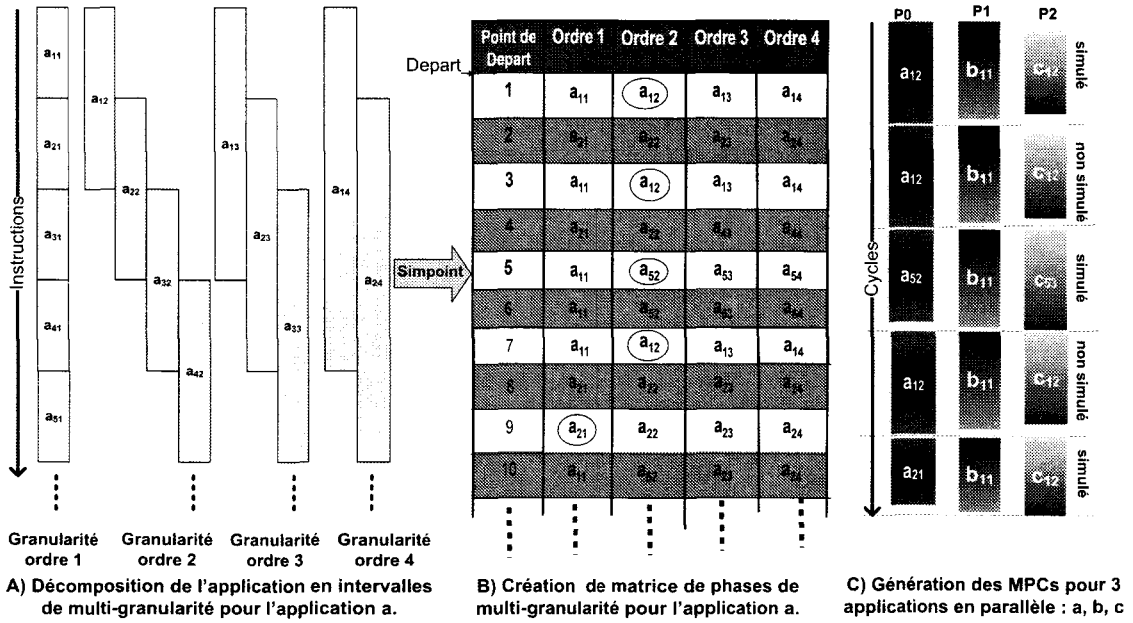


FIG. 5.1 – Les deux étapes de la méthode MGS. Les deux figures A et B correspondent à la première étape et la figure C correspond à la deuxième étape. Notation a_{ij} est le i^{ieme} intervalle de granularité j.

est formé en combinant deux intervalles contigus d'ordre 1. Dans cette figure, l'intervalle a_{12} correspond à une fusion des deux intervalles contigus a_{11} et a_{21} d'ordre 1. De la même manière, pour former les intervalles d'ordre 3, trois intervalles contigus d'ordre 1 sont combinés.

Remarquons qu'un point de départ correspond à plusieurs intervalles de granularité différente. Par exemple, les intervalles a_{21}, a_{22}, a_{23} et a_{24} (figure 5.1) sont des intervalles ayant des ordres de granularité différents mais le même point de départ (50K instructions). Ainsi la figure 5.1.B montre la matrice de phases avec multi-granularité pour l'application a. Chaque colonne correspond à un ordre de granularité. Dans cet exemple, quatre ordres de granularité sont présentés, il y a donc quatre colonnes dans la matrice. Au niveau d'un même ordre de granularité, les intervalles considérés comme étant similaires (même phase) ont le même identificateur de phase dans la colonne correspondante dans la matrice.

L'élément de la matrice à l'intersection de la ligne i et de la colonne j, correspond à une phase de j*t instructions (ordre de granularité j) dont le point de départ est à la (i-1)*t ième instructions. Ici, t représente la taille d'un intervalle d'ordre 1. Un exemple avec quatre ordres de granularité est donné dans la figure 5.1.A et 5.1.B. Chaque intervalle de la figure 5.1.A est représenté dans la figure 5.1.B par un identificateur de phases. Cet identificateur correspond à l'intervalle qui est le représentant de la phase en question. Par exemple, les deux intervalles a_{21} et a_{41} ont une granularité d'ordre 1 et commencent respectivement au premier et au troisième point de discrétisation d'ordre 1. Ces deux intervalles sont considérés similaires et ont par conséquent le même identificateur de phase a_{21} (figure 5.1.B). Leur phase correspondante a_{21} se trouve respectivement dans la deuxième et la quatrième ligne de la première colonne de la matrice.

Cette première étape est faite une seule fois pour tous les ordres de granularité pouvant

être rencontrés durant la simulation détaillée. A chaque fois qu'un ordre de granularité est détecté pour la première fois durant la simulation, la trace de phases correspondante est chargée en mémoire et la lecture dans la trace est effectuée à la phase correspondante. Grâce à une simulation fonctionnelle rapide, l'obtention de cette matrice de phases est faite en quelques minutes. Dans la sous-section suivante, nous montrerons comment les différentes matrices de phases des différentes applications sont utilisées pendant la simulation.

5.2.2 Deuxième étape : Génération et utilisation des grappes de phases MPCs

Durant cette étape, les matrices de phases générées dans la première étape sont utilisées. Comme nous l'avons déjà expliqué, les phases exécutées en parallèle par les processeurs sont combinées entre elles afin de composer un MPC. Un MPC contient p phases qui s'exécutent en parallèles, p étant le nombre de processeurs sur la plateforme MPSoC. Chaque nouveau MPC est associé à une entrée dans une table nommée MPCT pour *"Multi-Phase Cluster Table"*. Nous considérons que les MPCs contenant les mêmes phases parallèles ont le même comportement sur la plateforme architecturale et donnent les mêmes performances. Comme pour la méthode AS, les MPCs contenant les mêmes phases sont simulés une seule et unique fois.

5.2.3 Génération des MPCs

Dans la méthode MGS, pour discrétiser le recouvrement de phases exécutées en parallèle nous adoptons la méthode utilisée dans AS. Cette discrétisation est faite en utilisant la notion de barrière de simulation. La seule différence de cette méthode avec AS est qu'une seule phase est déterminée pour chaque processeur. Comme les processeurs ont des vitesses différentes, les tailles des phases sont aussi différentes. Comme les IPCs de chacune des applications concurrentes sont différents, le nombre d'instructions exécutées dans le MPC diffère d'un processeur à l'autre. Néanmoins, ce nombre d'instructions exécutées par chaque processeur est un multiple de t, la taille d'intervalle de la granularité de base, granularité d'ordre 1.

5.2.4 Utilisation des MPCs dans l'accélération de la simulation

L'identification d'une phase pour un processeur donné dans la matrice dépend du MPC et du nombre d'instructions exécutées par le processeur correspondant. Dans la figure 5.1.C, au niveau du processeur P0, le cinquième MPC commence au huitième point de discrétisation, après l'exécution de quatre phases d'ordre 2, et le nombre d'instructions exécutées par P0 dans ce MPC est égal à t (la taille d'intervalle d'ordre 1). La phase de P0 dans le cinquième MPC est alors a_{12} appartenant à la neuvième ligne. Dans la MPCT, la combinaison des identificateurs de phases parallèles représente un identificateur unique pour le MPC. Une fois qu'un nouveau MPC est simulé, une nouvelle entrée lui est allouée dans la MPCT. Les statistiques du nouveau MPC, le nombre d'instructions exécutées ou l'ordre de granularité de chaque phase ainsi que le nombre de cycles et l'énergie consommée sont sauvegardés dans la MPCT. Il est possible d'ajouter d'autres statistiques : le trafic sur le bus, le taux de défauts cache, etc. La table 5.1 montre une MPCT qui contient trois MPCs avec leurs statistiques. Ces trois MPCs sont générés par les trois applications (a, b et c) de la figure 5.1.C.

MPC	g_0	g_1	g_2	K-Cycles	Energy(mJ)	Repetition
a_{12}-b_{11}-c_{12}	100	50	100	280	104	3
a_{52}-b_{11}-c_{53}	100	50	150	487	190	1
a_{21}-b_{11}-c_{12}	100	50	150	277	150	1

TAB. 5.1 – Une table des clusters de multi-phases (MPCT) contenant 4 MPCs avec leurs statistiques. La variable "g_i" correspond à la granularité de phase (en K inst) pour le processeur i.

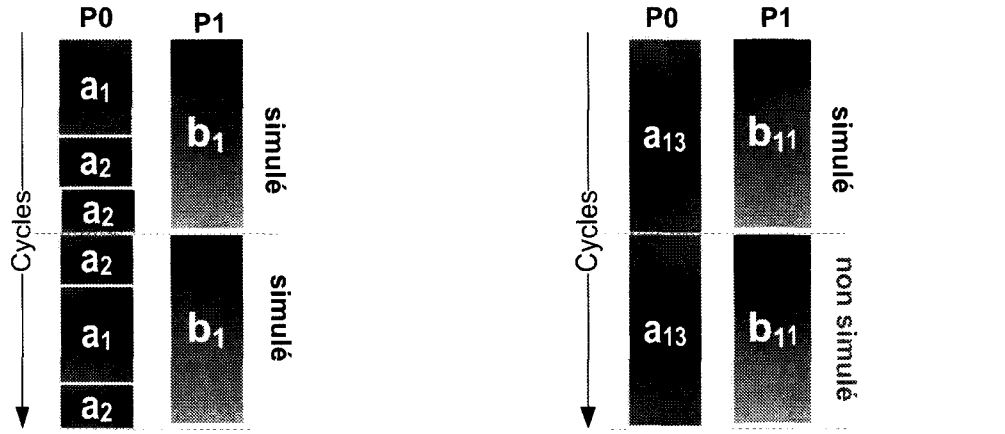
Après chaque MPC sauté ou simulé, toutes les entrées dans la MPCT sont testées afin de détecter une similarité avec les prochaines phases dans les différentes matrices des applications. Si une similarité est détectée entre un MPC déjà simulé et les prochaines phases dans les matrices, la recherche dans ces matrices est arrêtée. Les phases seront sautées de telle sorte que chaque processeur avance de la phase correspondante, et ainsi la fréquence dans l'entrée correspondante dans la MPCT (voir table 5.1) est incrémentée. Dans le cas contraire où il n'y pas de similarité, une simulation détaillée sera réalisée jusqu'à la génération d'un nouveau MPC.

La table 5.1 représente la MPCT de l'exemple de la figure 5.1. La simulation s'effectue jusqu'à la génération dynamique du premier MPC (a_{12}-b_{11}-c_{12}). Une entrée est allouée pour ce MPC dans la MPCT (voir la table 5.1). Ensuite, une similarité est recherchée entre (a_{12}-b_{11}-c_{12}) et une combinaison des prochaines phases dans les matrices. Dans son premier MPC, P0 a exécuté une phase d'ordre 2. Donc la prochaine phase du P0 se trouve sur la ligne correspondant au point de départ 3 dans la matrice (voir figure 5.1.B). La même procédure est appliquée pour P1 et P2. Nous remarquons qu'il existe une combinaison de phases similaire à (a_{12}-b_{11}-c_{12}). Puisque ce MPC existe déjà dans la MPCT, alors la fréquence associée à son entrée est simplement incrémentée et ainsi ce MPC ne subit pas de simulation détaillée. En revanche, les trois processeurs P0, P1 et P2 seront respectivement avancés de 100K, 50K et 100K instructions. Après la réalisation du saut d'un certain nombre d'instructions dans la simulation détaillée, nous recherchons de nouveau une similarité entre les entrées de la MPCT et les matrices. A ce point, au niveau du P0, la prochaine phase qui succède aux deux phases d'ordre 2 se trouve sur la ligne correspondant au point de départ 5. La phase située sur cette ligne et sur la colonne 2 est a_{52}, (voir figure 5.1.B), elle est différente de la phase a_{12} déjà exécutée par P0. Ainsi, il n'y a pas de similarité entre une combinaison des prochaines phases dans les matrices et les MPCs déjà simulés. Cependant, une simulation détaillée est donc lancée pour générer dynamiquement le MPC (a_{52}-b_{11}-c_{53}). De nouveau, la similarité n'existe pas et un nouveau MPC (a_{21}-b_{11}-c_{12}) est encore généré. Une entrée dans la MPCT est allouée pour chacun de ces deux MPCs (voir la table 5.1). Après la génération de ces deux MPCs, (a_{12}-b_{11}-c_{12}) réapparaît une troisième fois. La fréquence associée à l'entrée correspondante est maintenant incrémentée et elle est égale à trois.

Comme la méthode AS, la méthode MGS est compatible avec les deux techniques de saut de simulation : simulation fonctionnelle et checkpointing. Ces deux techniques ont pour but de récupérer le contexte du programme à la fin du MPC sauté. Ainsi les sauts des MPCs qui se répètent sont réalisés en utilisant l'une ou l'autre de ces deux techniques.

5.3 Gain en accélération par MGS

La méthode AS, décrite dans le chapitre 3, est conservatrice dans la détection des similarités des CSs. Les CSs doivent contenir les mêmes phases avec le même ordre pour être



A) Génération par AS de 2 CSs a_1,a_2,a_2-b_1,b_1 et a_2,a_1,a_2-b_1,b_1 pour 2 applications a et b en parallèle.

B) Génération par MGS de 2 MPCs a_{12}-b_{11} et a_{12}-b_{11} pour 2 applications a et b en parallèle.

FIG. 5.2 – Méthodes AS et MGS sont appliquées sur la même partie du code des 2 applications en parallèle a et b. Dans le cas de AS, le deuxième CS est simulé tandis que le deuxième MPC a été sauté par MGS.

considérés similaires. Ce critère réduit l'accélération en particulier quand les séquences de phases sont longues. C'est le cas des applications parallèles qui ont des comportements extrêmement différents sur la plateforme architecturale. Autrement dit, les combinaisons de phases contenant les mêmes phases avec différents ordres sont considérées comme CS différents au niveau de la méthode AS. Tandis que ces mêmes combinaisons sont considérées comme des MPCs similaires au niveau de la méthode MGS. D'où le gain en accélération obtenu par MGS. La figure 5.2.A montre l'inconvénient de la méthode AS. Dans cette figure, nous avons pour le processeur P0, deux CSs. Ces deux CSs contiennent exactement les mêmes phases avec un ordre différent. Ces deux CSs sont considérés différents. Ainsi, ces deux CSs vont subir la simulation détaillée. Dans les deux séquences de phases, les blocs de base exécutés sont les mêmes et les nombres d'instructions exécutées sont identiques. La figure 5.2.A montre qu'en définitive, ces deux séquences vont avoir les mêmes BBs (blocs de base) et les même fréquences correspondantes. Comme SimPoint prend en compte les BBs et leur fréquences et néglige l'ordre d'exécution de ceux-ci, alors ces deux séquences forment deux intervalles similaires. Cependant dans le cas de MGS, les deux séquences de phases correspondent à deux intervalles de même granularité. En détectant la similarité au niveau de chaque granularité, Simpoint va considérer que ces deux intervalles sont similaires. La figure 5.2.B montre que la phase a_{13} qui a une granularité d'ordre-3 représente les deux séquences de phases de la figure 5.2.A. Ainsi, le deuxième MPC généré par MGS dans la figure 5.2.B est similaire au premier et en conséquent il va être sauté. Cet exemple montre le gain en accélération de simulation obtenu par la méthode MGS.

I-cache	8KB accès direct, latence 1 cycle
D-cache	4KB associative par ensembles à 4 voies, latence 1 cycle
Mémoire	latence 64 cycles
Core	Arm version 7

TAB. 5.2 – Configuration du processeur MPARM.

App/version \ Ordre-Gran	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
adpcm.dec	5	4	8	5	5	5	5	8	3	4	3	4	5	5	6	6	6	5	7	4
adpcm.enc	8	7	5	3	6	6	4	5	6	8	3	5	5	6	5	5	6	5	5	4
blf.dec	4	4	4	4	4	4	3	4	4	4	4	3	4	3	3	4	4	4	2	3
blf.enc	3	4	3	3	3	4	3	4	4	3	4	4	4	3	4	4	3	4	3	3
fft	9	9	8	4	7	5	6	9	7	7	9	9	8	8	9	4	7	6	5	5
ifft	9	9	7	7	7	7	8	7	8	9	6	9	9	8	7	7	6	7	8	8
gsm.dec	7	6	3	5	6	3	5	5	3	4	4	3	3	4	3	3	3	3	3	3
gsm.enc	8	8	5	6	7	6	2	6	6	3	5	6	5	3	5	5	3	4	6	4
h264.dec	8	7	8	7	7	9	8	8	9	8	8	8	8	8	7	8	7	7	7	8
rjind.dec	3	4	3	3	4	4	4	4	4	4	3	3	3	3	3	3	3	3	2	3
rjind.enc	4	3	4	3	3	4	3	4	4	3	3	3	3	3	3	3	3	4	3	3

TAB. 5.3 – Nombre de phases détectées pour chaque application/version pour les différents ordres de granularité.

5.4 Résultats expérimentaux de la méthode MGS

Dans cette section, nous évaluons la précision et les performances de la méthode MGS en utilisant six applications : cinq appartenant à la suite de Mibench plus le benchmark H264. Ce sont les mêmes qui ont été utilisées pour évaluer la performance de la méthode AS et les deux techniques complémentaires à AS (voir chapitre précédent). Le nombre de processeurs du MPSoC que nous avons simulé varie entre 2 et 12. Quand une des cinq premières applications est exécutée, chacune de ses deux versions "encode" et "decode" est exécutée par un des processeurs. Tandis que pour H264, on exécute que la version "decode". La configuration de chaque processeur du MPARM est la même que celle utilisée précédemment (voir tableau 5.2). Rappelons que le facteur d'accélération utilisé correspond au ratio entre le nombre total d'instructions de toutes les applications et le nombre d'instructions simulées par tous les processeurs en mode détaillé.

5.4.1 Génération des matrices de phases des applications

La génération des traces de phases des différents ordres de granularité des applications est faite une fois pour toute durant l'opération du DSE. Les paramètres (taille des intervalles, le nombre maximum des phases) utilisés pour MGS sont les mêmes que ceux utilisés précédemment pour AS. La taille t des intervalles de granularité d'ordre 1, est égale à 50K instructions. Pour SimPoint, le **Max_K** vaut 10. Les raisons de ce choix ont été expliquées dans la section 3.5.3. Notons qu'il y a une relation entre l'ordre de granularité maximale détecté durant la simulation et TWSB. A une faible valeur du TWSB correspond une importante granularité maximale et vice versa. Durant nos expériences, nous avons mesuré que pour un TWSB de 5% l'ordre maximal est de 64 alors que pour un TWSB de 20% l'ordre maximal diminue à 20 (voir figure 5.8(a)). La génération de chaque trace n'est pas coûteuse, elle nécessite quelques minutes pour sa réalisation. Pour des questions d'espace, nous nous limitons

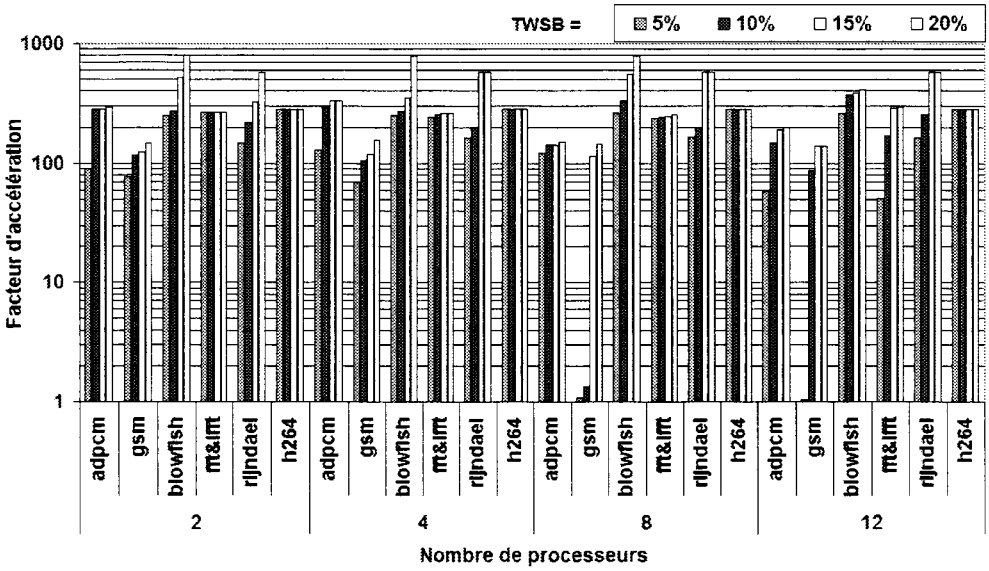


FIG. 5.3 – Variation du facteur d'accélération (en log) pour quatre valeurs du TWSB. La même application est exécutée sur 2, 4, 8 et 12 processeurs.

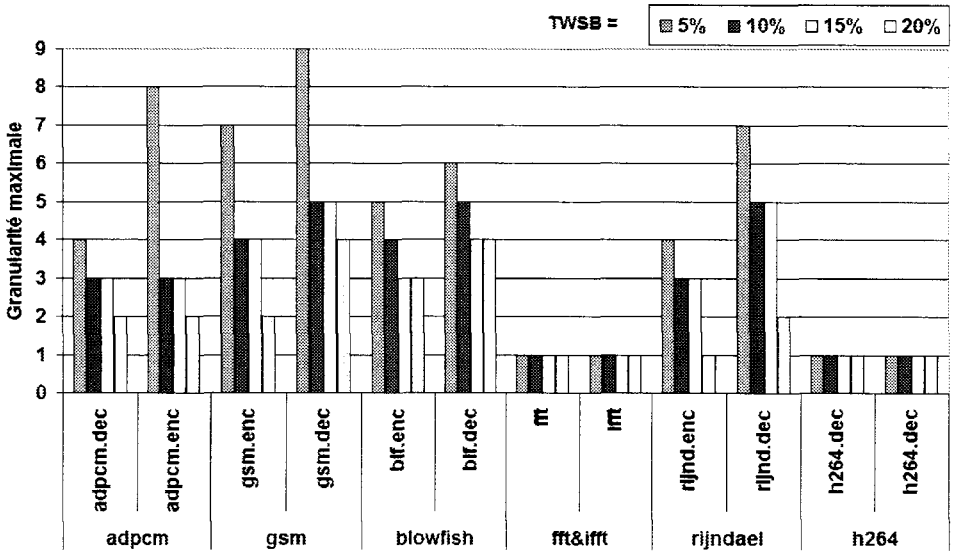
à montrer dans le tableau 5.3 le nombre de phases détectées dans les traces de phases pour 20 ordres de granularité pour chacune des versions des applications.

5.5 Performances de MGS pour les applications homogènes concurrentes

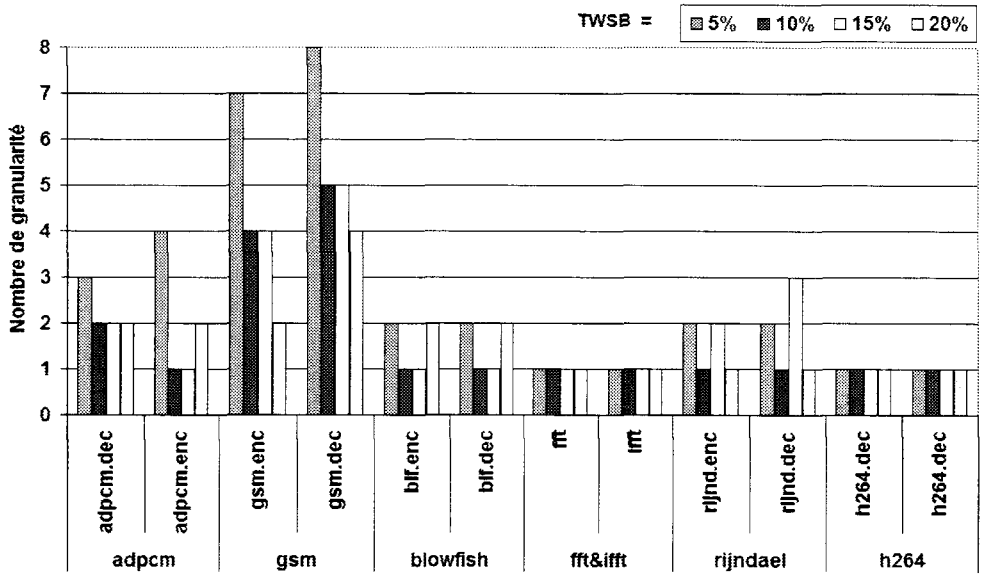
Dans cette section, nous étudions le facteur d'accélération et l'erreur dans l'estimation des performances obtenus par l'utilisateur de MGS dans le cas des applications homogènes concurrentes. Par homogénéité nous désignons la similarité dans le comportement des applications en termes de besoins en ressources. Nous comparons aussi les résultats de MGS avec ceux de AS dans les mêmes conditions.

5.5.1 Relation entre TWSB et l'accélération de la simulation

La figure 5.3 montre l'effet de l'augmentation de la valeur du TWSB sur l'accélération de la simulation obtenue par MGS. Quand la valeur du TWSB est importante, des temps d'attente importants sont autorisés au niveau des barrières de simulation. Ainsi, le nombre de MPCs est important. Dans ce cas aussi le nombre d'instructions exécutées dans les MPCs est réduit, d'où un plus grand nombre de MPCs de tailles réduites. La figure 5.4(a) montre la granularité maximale de chaque application/version pour chaque valeur du TWSB lorsque 2 processeurs sont utilisés. Il est évident que la granularité maximale diminue avec l'augmentation de la valeur du TWSB. En définitive, avec l'augmentation du TWSB, le nombre de MPCs représentatifs différents diminue, d'où une augmentation du facteur d'accélération. La figure 5.3 montre l'augmentation du facteur d'accélération avec l'augmentation de la valeur du TWSB pour 2, 4, 8 et 12 processeurs. Comme dans le cas de AS, ce sont les



(a) Granularité maximale avec 4 valeurs du TWSB pour des applications sur 2 processeurs.



(b) Nombre de granularités avec 4 valeurs du TWSB pour des applications sur 2 processeurs.

FIG. 5.4 – Variation de la granularité maximale et le nombre de granularités avec 4 valeurs du TWSB pour des applications sur 2 processeurs.

trois applications blowfish, rijndael et h264 qui donnent des facteurs d'accélération les plus importants. Les meilleurs facteurs d'accélération sont approximativement 801 et 574 respectivement pour blowfish et rijndael. Cela est dû au fait que ces applications ont un nombre de phases réduit (voir tableau 5.3). La figure 5.4(b) montre le nombre d'ordres de granularités rencontrés durant la simulation pour chaque application/version sur 2 processeurs. La régularité dans le comportement de blowfish et rijndael apparaît dans leur nombre réduit de

granularités détectées durant la simulation.

Pour l'application h264, la même version "decode" est exécutée sur tous les processeurs. Ainsi les processeurs ont le même comportement. Les MPCs générés comprennent une seule phase de granularité d'ordre 1 par processeur pour toutes les valeurs du TWSB (voir les deux figures 5.4(a) et 5.4(b)). Le nombre total de MPCs générés et le nombre de MPCs représentatifs se stabilisent quelque soit la valeur du TWSB. Par conséquent, le facteur d'accélération sur les 4 configurations d'architecture est quasiment constant (proche de 300). Dans les mêmes conditions, le facteur d'accélération de la simulation, pour les applications fft&ifft, est quasiment constant sur 2 processeurs et sur 4 processeurs quand le TWSB varie entre 10% et 20%.

Le nombre important de granularités respectivement pour gsm.encode et gsm.decode dans la figure 5.4(b) montre l'irrégularité dans le comportement de gsm. Cette irrégularité ainsi que le nombre élevé de phases de deux versions de gsm (voir tableau 5.3) provoquent un faible facteur d'accélération surtout pour 8 et 12 processeurs.

La plupart des applications ont le même facteur d'accélération sur les 4 configurations de processeurs. Cela est dû au fait que les applications correspondantes ont les mêmes granularités quelque soit la configuration des processeurs.

5.5.2 Relation entre TWSB et l'erreur d'estimation

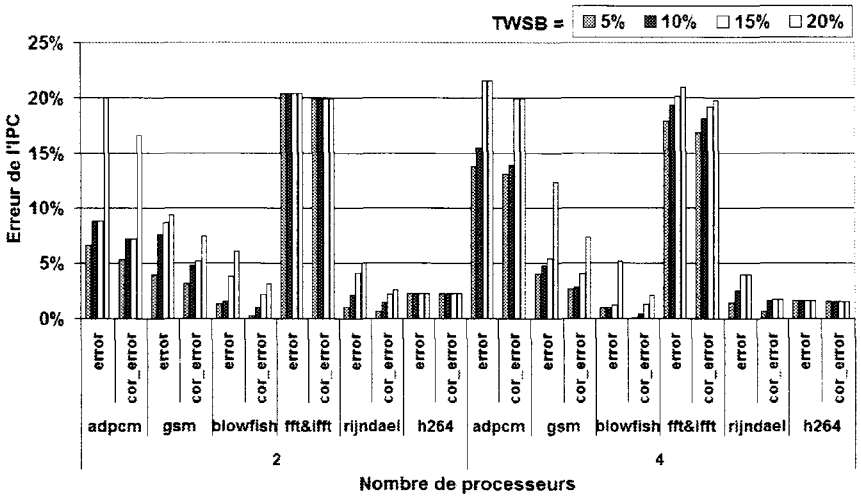
D'après ce qui a été expliqué au chapitre 3, deux facteurs influent sur l'erreur dans l'estimation de l'IPC. Le premier est l'approximation de l'IPC des différents MPCs avec l'IPC du MPC représentatif correspondant et le deuxième est dû à l'insertion de barrière de simulation. En effet, comme aucune instruction n'est exécutée durant une attente devant une barrière de simulation, l'IPC est sous estimé. La figure 5.5 montre l'erreur de l'IPC estimé en fonction de quatre valeurs du TWSB pour 2, 4, 8 et 12 processeurs notée "error" dans la figure. Les applications sont les mêmes que celles de la figure 5.3. Comme on peut le voir, l'erreur de l'IPC est pour certaines applications assez importante. Elle dépasse les 15% par exemple pour adpcm et fft&ifft. Afin de corriger l'IPC nous avons appliqué la formule (5.1) qui a été expliquée dans le paragraphe 4.2.2. Cette formule élimine la moyenne de cycles d'attente (notée $Avr_{wait-cyc}$) du nombre total de cycles. La moyenne de cycles d'attente est estimée par la formule (5.2).

$$Cor_{IPC} = \frac{Total_{instr}}{(Tot_{cycles} - Avr_{wait-cyc})} \tag{5.1}$$

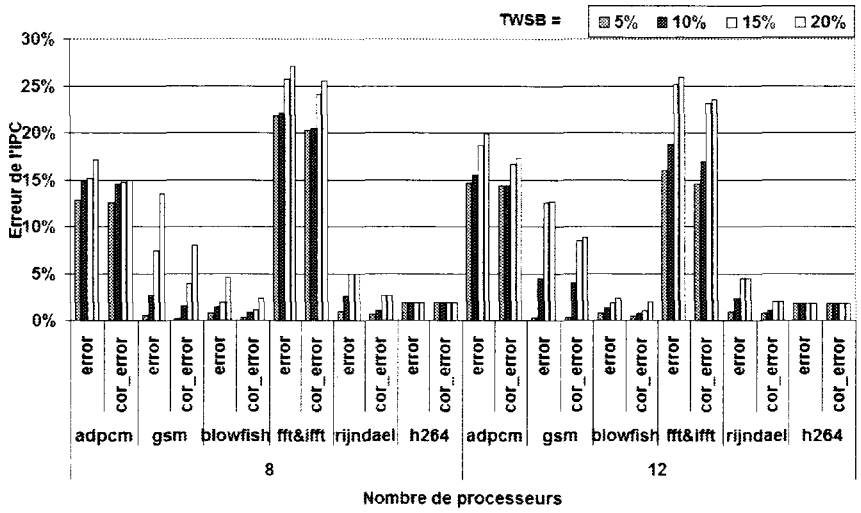
$$Avr_{wait-cyc} = \frac{(\sum_{i=1}^{Nb_{CS}} Est_{wait-cyc_i} * Freq_i)}{Nb_{proc}} \tag{5.2}$$

La figure 5.5 montre aussi l'erreur de l'IPC corrigé (cor_error). Nous pouvons constater que cette erreur est inférieure à l'erreur sans correction. Nous observons aussi que pour chaque application, l'erreur sans correction et l'erreur avec correction augmentent avec la valeur de TWSB. En fait, quand TWSB augmente le nombre de barrières de simulation injectées augmente, ce qui accroît les valeurs des deux erreurs "error" et "cor_error". De même, on peut constater que dans certains cas l'erreur est constante avec l'augmentation du TWSB, cela est dû au fait que les MPCs générés sont quasiment identiques, exemple h264 et fft&ifft exécutées sur 2 processeurs.

Comme la formule de correction de l'IPC (5.1) élimine la moyenne des cycles injectés par



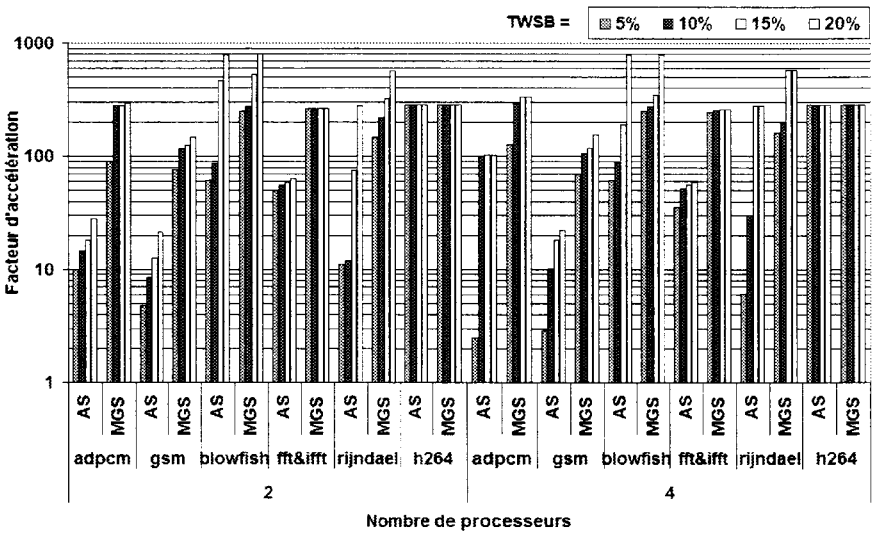
(a) Variation de l'erreur de l'IPC avec 4 valeurs du TWSB pour des applications sur 2 et 4 processeurs.



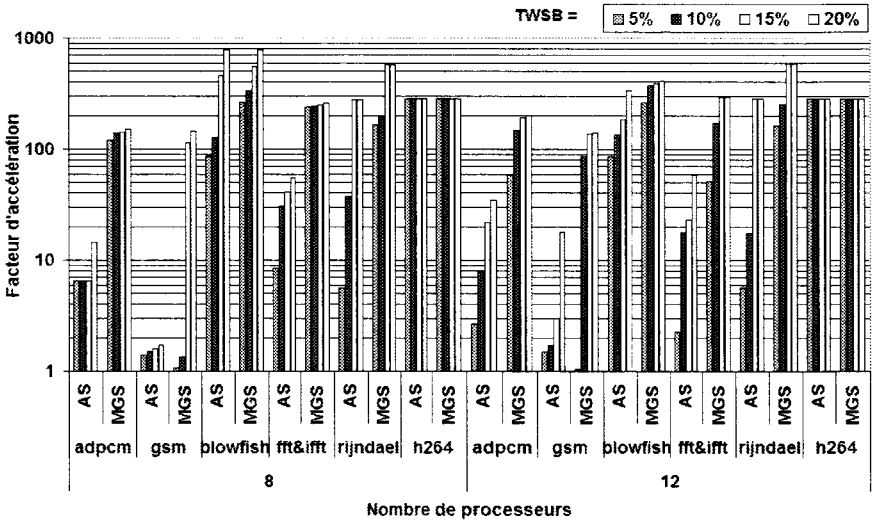
(b) Variation de l'erreur de l'IPC avec 4 valeurs du TWSB pour des applications sur 8 et 12 processeurs.

FIG. 5.5 – Variation de l'erreur de l'IPC avec 4 valeurs du TWSB pour des applications sur 2, 4, 8 et 12 processeurs.

la barrière de simulation, cette erreur corrigée est généralement liée au premier facteur qui est l'approximation de l'IPC des différents MPCs avec l'IPC du MPC représentatif. Quand l'erreur avec correction est proche de l'erreur sans correction, cela signifie que l'erreur due à la classification des phases est largement plus grande que celle due à l'insertion de barrières de simulation. Ceci est le cas pour les applications fft&ifft et adpcm.



(a) Le facteur d'accélération donné par AS et par MGS pour des applications sur 2 et 4 processeurs.



(b) Le facteur d'accélération donné par AS et par MGS pour des applications sur 8 et 12 processeurs.

FIG. 5.6 – Facteur d'accélération donné par AS et MGS.

5.5.3 Comparaison de la méthode MGS à la méthode AS dans le cas des applications homogènes

Dans cette section, nous comparons les performances de AS et de MGS du point de vue du facteur d'accélération et précision dans l'estimation. La figure 5.6 montre les facteurs d'accélération obtenus respectivement par AS et par MGS pour 2, 4, 8 et 12 processeurs en fonction de quatre valeurs du TWSB. Il apparaît clairement que le facteur d'accélération de MGS est plus important que celui de AS. Le gain en accélération obtenu par MGS, est de

100% à 8000% plus important que AS. La cause de ce gain est expliquée dans la section 5.3. Pour h264, le facteur d'accélération est le même dans le cas de AS, et de MGS. Dans le cas de AS les CSs de h264 contiennent une seule phase par processeur. Ainsi chaque processeur exécute un seul intervalle de 50K instructions. Cette taille d'intervalle correspond au niveau de MGS à la taille des intervalles de granularité d'ordre 1. Ainsi dans les MPCs de h264, chaque processeur exécute un seul intervalle de granularité d'ordre 1. Comme les intervalles executés en parallélle sont les mêmes pour AS et MGS, la disposition des phases entre les processeurs ne change pas entre AS et MGS. Les parties du code qui subissent la simulation détaillée sont les mêmes, d'où l'égalité des facteurs d'accélération de AS et de MGS.

La figure 5.7 montre l'erreur de l'IPC de AS et de MGS pour les mêmes applications que ceux de la figure 5.6 pour 4 valeurs du TWSB. Comme on peut le voir, l'erreur de MGS est plus importante que celle de AS. Elle dépasse les 10% pour adpcm, fft&ifft et gsm quand le TWSB est égal à 20%. Cette faible précision de MGS est mentionnée dans la section 5.3. En effet, la méthode AS est trés conservative dans la détection des similarités des séquences de phases. D'où l'estimation de la performance est précise au dépens de l'accélération. Dans le cas de MGS, une séquence de phases est représentée par une seule phase. Ainsi des séquences de phases, dont l'ordre des phases est différent, sont considerées comme des séquences différentes dans le cas de AS tandis que dans le cas de MGS, elles sont considerées comme similaires.

En effet, le temps d'exécution d'une phase dépend de celle exécutée précédemment. Cette dépendance est en particulier en termes de défauts cache. Considérons les deux séquences de phases "*a,b,c*" et "*a,c,b*" et supposons que chacune de deux phases *a* et *b* contienne une itération pour la même boucle et *c* contienne les instructions qui succèdent la boucle. Il est évident que les temps d'exécution de ces deux séquences ne sont pas égaux, la phase *b* dans la deuxième séquence a un nombre de défauts cache plus important que dans la première séquence. Ainsi la comparaison qui est faite par AS au niveau des séquences de phases est précise. Le manque de précision de MGS n'influe pas sur les applications aux comportements réguliers, l'erreur de rijndael et de blowfish reste inférieure d'environ 5%, mais est importante pour les applications ayant des comportements irréguliers. Nous déduisons que AS est généralement plus précise que MGS dans le cas des applications homogènes.

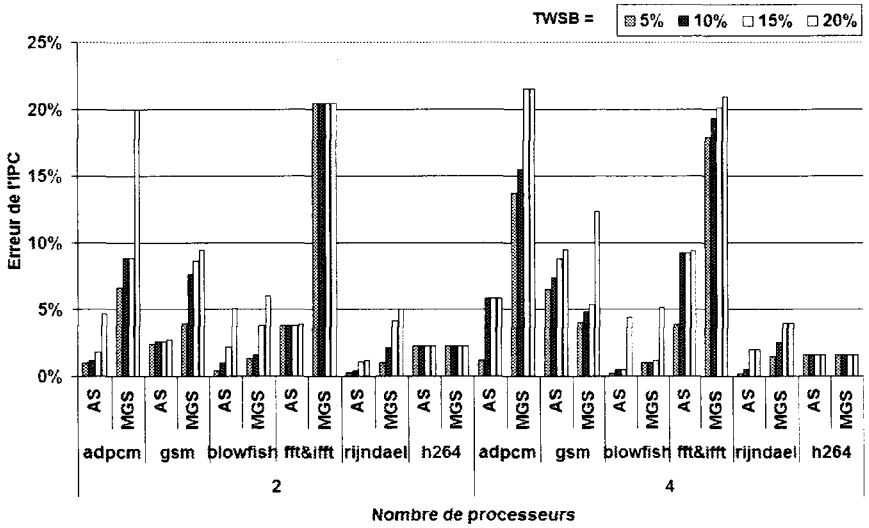
5.6 Performances de MGS pour les applications hétérogènes concurrentes

Dans cette section, nous nous intéressons au facteur d'accélération et au niveau de précision de l'estimation de MGS pour les applications hétérogènes concurrentes. Puis, nous comparons MGS respectivement à AS et à la méthode Cophase, proposée dans [21, 18], dans le cas des applications hétérogènes.

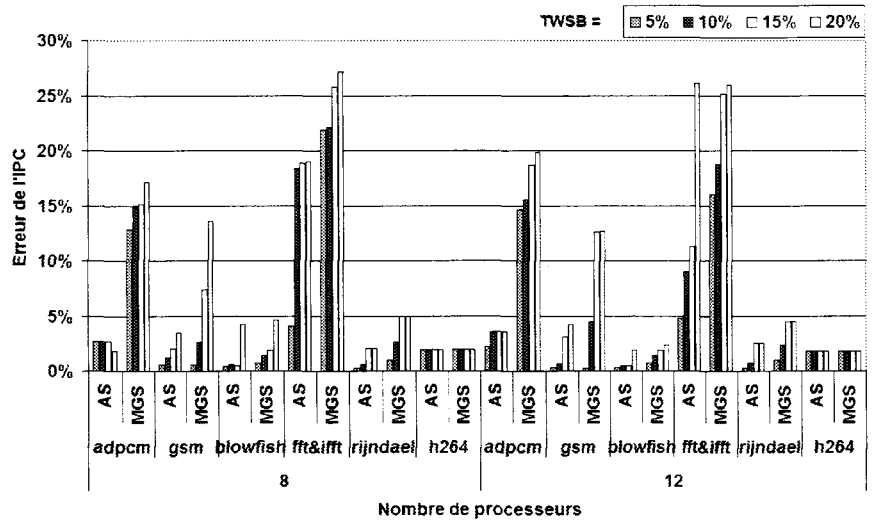
5.6.1 Relation entre TWSB et l'accélération de la simulation

La figure 5.8(a) montre, pour quatre valeurs du TWSB, la granularité maximale dans les MPCs de chaque application/version sur 4 processeurs. On peut distinguer deux facteurs dans la figure 5.8(a)

1. La diversité des granularités maximales montre l'hétérogénéité des applications qui s'exécutent en parallèle. L'écart le plus important apparaît en particulier pour adpcm



(a) Erreurs de l'IPC données par AS et par MGS pour des applications sur 2 et 4 processeurs.

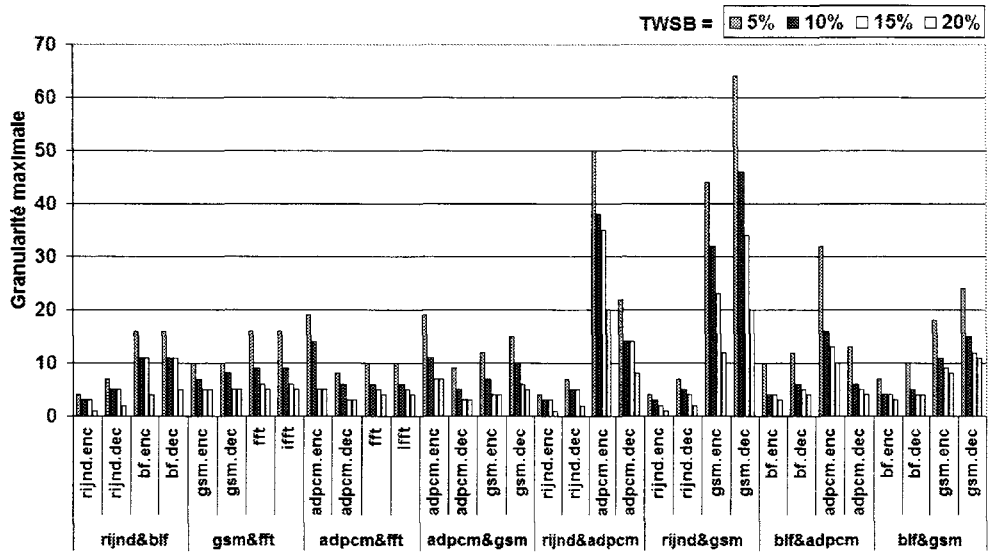


(b) Erreurs de l'IPC données par AS et par MGS pour des applications sur 8 et 12 processeurs.

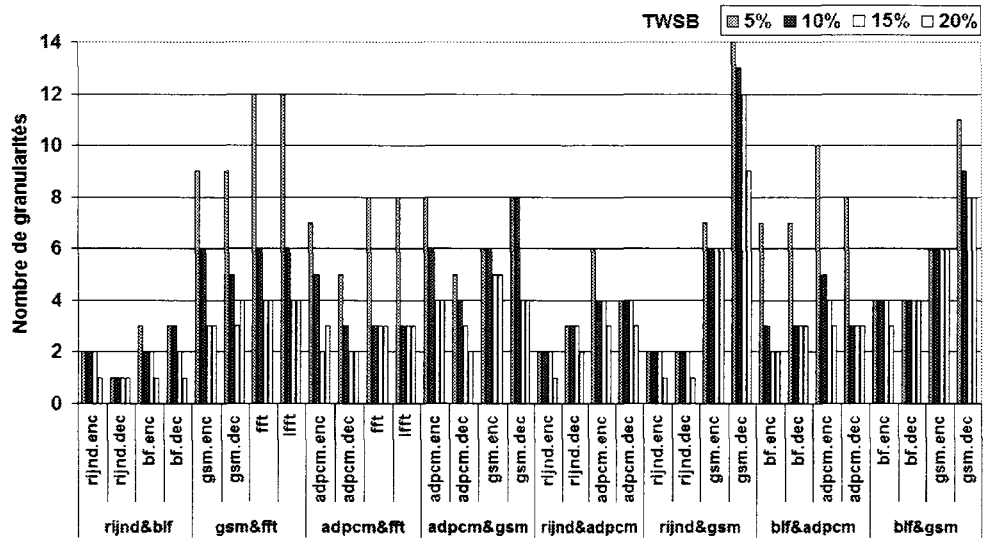
FIG. 5.7 – Erreurs de l'IPC données par AS et MGS.

et gsm quand l'une de ces applications est exécutée en parallèle avec rijndael. Dans le cas de rijnd&gsm et un TWSB de 5%, le nombre maximum d'instructions exécutées par gsm dans les MPCs est de 3,2 millions (64*50000) d'instructions, alors que pour rijndael le maximum est de 250K (5*50000) instructions.

2. La granularité maximale pour chacune des applications/versions diminue avec l'augmentation du TWSB. Dans le cas de rijnd&gsm le nombre maximum d'instructions du gsm atteint 1 million (20*50000) quand la valeur du TWSB vaut 20%. Cette diminution du nombre des instructions exécutées dans les MPCs correspond également à une aug-



(a) Granularité maximale avec 4 valeurs du TWSB pour des applications sur 4 processeurs.



(b) Nombre de granularité avec 4 valeurs du TWSB pour des applications sur 4 processeurs.

FIG. 5.8 – Variation de la granularité maximale et la variation du nombre de granularités avec 4 valeurs du TWSB pour des applications hétérogènes sur 4 processeurs.

mentation du nombre de MPCs. Ce qui explique l'augmentation de l'accélération avec l'augmentation du TWSB (voir figure 5.9).

La figure 5.8(b) montre le nombre de différentes granularités des MPCs présentes dans les applications/versions. Les applications de la figure 5.8(b) sont identiques à celles de la figure 5.8(a). Ce nombre reflète le comportement de l'application/version correspondante. Autrement dit, le degré d'irrégularité du comportement de l'application/version est pro-

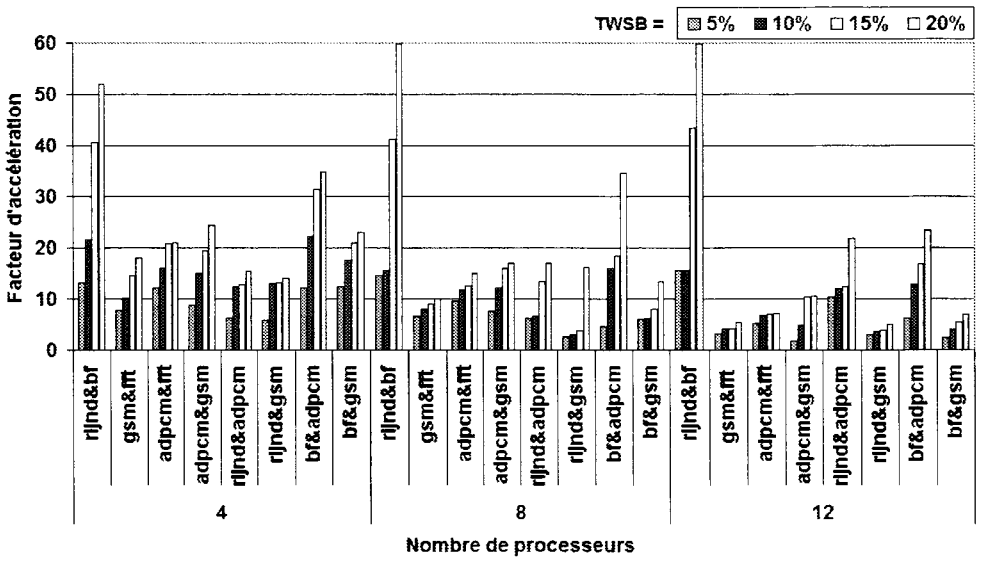


FIG. 5.9 – Variation du facteur d'accélération pour quatre valeurs du TWSB. Les applications sont exécutées sur 4, 8 et 12 processeurs.

portionnel au nombre des granularités. Ainsi plus ce nombre est important plus l'application/version correspondante est irrégulière. Les applications rijnd&blf sont régulières car le nombre de granularités correspondant est relativement faible. Gsm est l'application la plus irrégulière car elle a le nombre de granularités le plus important et elle change fréquemment d'un ordre de granulartité à un autre à cause de son irrégularité. La figure 5.8(b) montre que les deux versions "encode" et "decode" du gsm ont au minimum six ordres de granularités différents. De même le comportement de gsm influe sur le comportement de l'application s'exécutant en parallèle en augmentant son nombre de granularité. A titre d'exemple, le nombre de granularité de blowfish quand elle est exécutée en parallèle avec gsm est plus important que lorsqu'elle est exécutée avec rijndael (voir la figure 5.8(b)).

La figure 5.9 montre l'effet de l'augmentation de la valeur du TWSB sur le facteur d'accélération obtenu par MGS. Les applications parallèles hétérogènes sont exécutées sur 4, 8 et 12 processeurs. La notation "&" signifie que les applications correspondantes sont exécutées en parallèle de manière à ce que la moitié des processeurs exécutent les versions "encode" et l'autre moitié les versions "decode" des applications 3.5.2. Les combinaisons des applications s'exécutant en parallèle sont choisies afin d'avoir une grande variation de comportements entre elles. L'effet du TWSB sur l'accélération dans le cas des applications homogènes, que nous avons expliqué précédemment, est identique pour les applications hétérogènes. Ainsi, quand la valeur du TWSB augmente, le nombre de barrières de simulation augmente. Dans ce cas le nombre d'instructions exécutées dans les MPCs diminue et le nombre total de MPCs générés augmente. Par conséquent, la probabilité de générer des MPCs répétitifs augmente. Ce qui explique l'augmentation du facteur d'accélération avec la valeur du TWSB dans la figure 5.9.

Dans le cas de 4 processeurs et un TWSB de 20%, le facteur d'accélération donné par MGS varie entre 14 pour rijnd&gsm et 52 pour rijnd&blf. L'accélération moyenne des applications étudiées dans la figure 5.9 est un facteur de 22.

Les groupes de "rijnd&blf" ont le plus important facteur d'accélération. Ce résultat est dû en particulier à la régularité de chacune des deux applications rijndael et blowfish. Cette régularité apparaît dans les ordres faibles de granularités des phases qui forment les MPCs ainsi que dans le nombre faible de granularités rencontrées (voir figure 5.8). Généralement, les groupes d'applications contenant gsm ont le plus faible facteur d'accélération et surtout les groupes "rijnd&gsm". Le nombre faible des facteurs des "rijnd&gsm" s'explique par deux raisons. D'une part, rijndael et gsm ont deux comportements extrêmement differents. Rijndael a un taux de défaut cache plus élevé que gsm. La diversité dans le taux de défaut cache entre gsm et rijndael apparaît dans la diversité dans leurs nombres d'instructions exécutées dans leur MPCs (voir figure 5.8(a)). Pour un TWSB de 5%, les phases de gsm ont au maximum un ordre de granulartié de 64 tandis que celles de rijndael ont au maximum un ordre de 7. Ce qui réduit le nombre total des MPCs générés diminuant ainsi la probabilité d'avoir des MPCs qui se repétent. D'autre part, gsm a un nombre important d'ordres de granulartés augmentant le nombre des MPCs représentatifs et diminuant ainsi le facteur d'accélération.

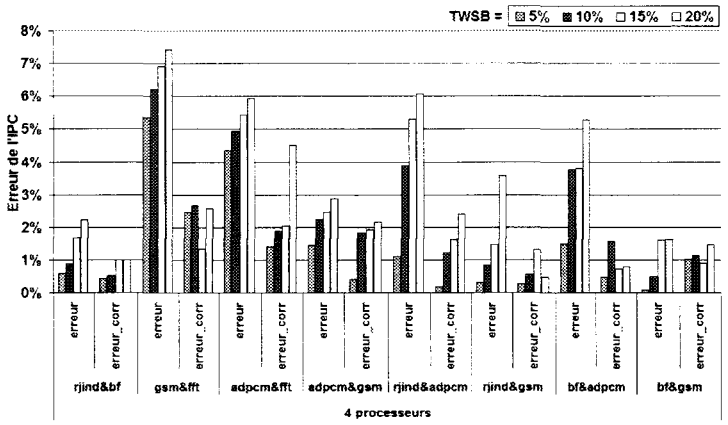
Les groupes "blf&adpcm" ont des facteurs d'accélération relativement élevés malgré l'hétérogénéité de blowfish et adpcm. Le nombre de défauts cache de blowfish est plus important que celui de adpcm. Ainsi, pour un TWSB de 20%, un maximum de 200k instructions du blowfish couvre un maximum de 500K instructions de adpcm (voir figure 5.8(a)). Cette hétérogeneité n'influe pas sur l'accélération de la simulation car chacune des deux applications a un comportement périodique, ce qui diminue le nombre de MPCs représentatifs. En testant les IPCs , à chaque fois que le processeur P0 exécute exactement 1 million d'instructions, nous avons remarqué pour blf&adpcm exécutée sur 4 processeurs que la différence entre le minimum et le maximum IPC est de 2%. Tandis que pour rijnd&gsm la différence est de 12%. Ce qui nous permet de déduire que blf&adpcm est plus régulière que rijnd&gsm. Par conséquent, les facteurs d'accélération des groupes "blf&adpcm" sont plus importants que ceux des groupes "rijnd&gsm".

Notons que le facteur d'accélération diminue avec l'augmentation du nombre de processeurs (voir la figure 5.9). En effet, l'augmentation du nombre de processeurs réduit l'accélération de la simulation de manière significative pour certaines applications telles que gsm&fft et blf&gsm. Pour un TWSB de 20%, l'accélération de gsm&fft diminue de 18 pour 4 processeurs, à 5 pour 12 processeurs.

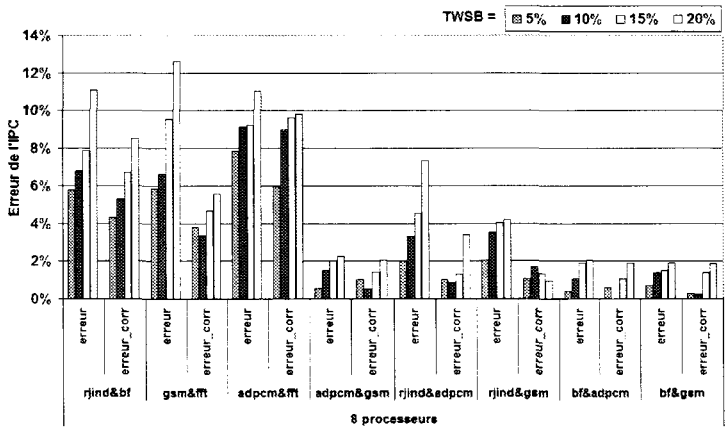
5.6.2 Relation entre TWSB et l'erreur de l'estimation

Là aussi nous avons deux sources d'erreurs dans l'estimation de l'IPC et de la consommation de puissance. La figure 5.10 montre l'erreur dans l'estimation de l'IPC pour quatre valeurs du TWSB pour des groupes d'applications exécutés sur 4, 8 et 12 processeurs. Ces groupes sont ceux de la figure 5.9. L'effet du TWSB que nous avons montré précédemment sur l'erreur est aussi valable dans le cadre du MGS pour les applications hétérogènes. On peut voir aussi dans la figure 5.10 que l'erreur de l'IPC augmente aussi avec l'augmentation du nombre de processeurs. Cette erreur reste raisonnable et inférieure à 13% pour un TWSB de 20%. La valeur moyenne est de 6%.

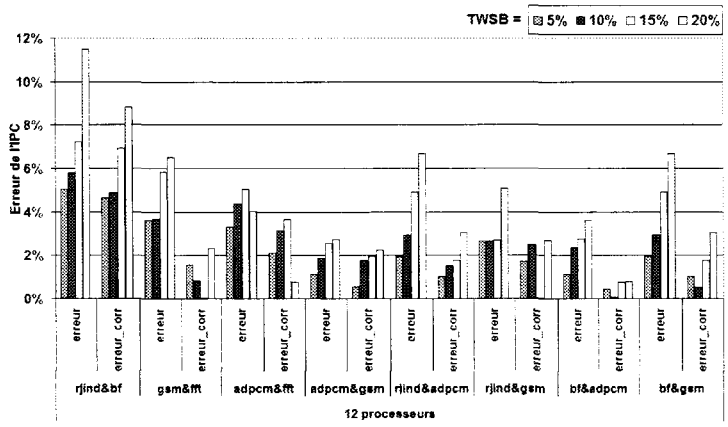
Comme l'injection des cycles d'attente sous-estime l'IPC obtenu par MGS pour les ap-



(a) Erreur de l'IPC donnée par MGS pour 4 processeurs.



(b) L'erreur de l'IPC donnée par MGS pour 8 processeurs.



(c) Erreur de l'IPC donnée par MGS pour 12 processeurs.

FIG. 5.10 – Variation de l'erreur de l'IPC sans correction et de l'erreur de l'IPC avec correction pour 4 valeurs du TWSB. Les différentes applications sont exécutées sur 4, 8 et 12 processeurs.

plications hétérogènes, nous pouvons adopter la formule (5.1) pour corriger l'estimation de l'IPC. Comme on peut le voir sur la figure 5.10, l'erreur corrigée est plus petite que l'erreur sans correction. Pour un TWSB de 20% l'erreur corrigée est inférieure à 10%. La valeur moyenne est de 3%. Pour certains groupes, par exemple blf&adpcm, la réduction de l'erreur est de 90%.

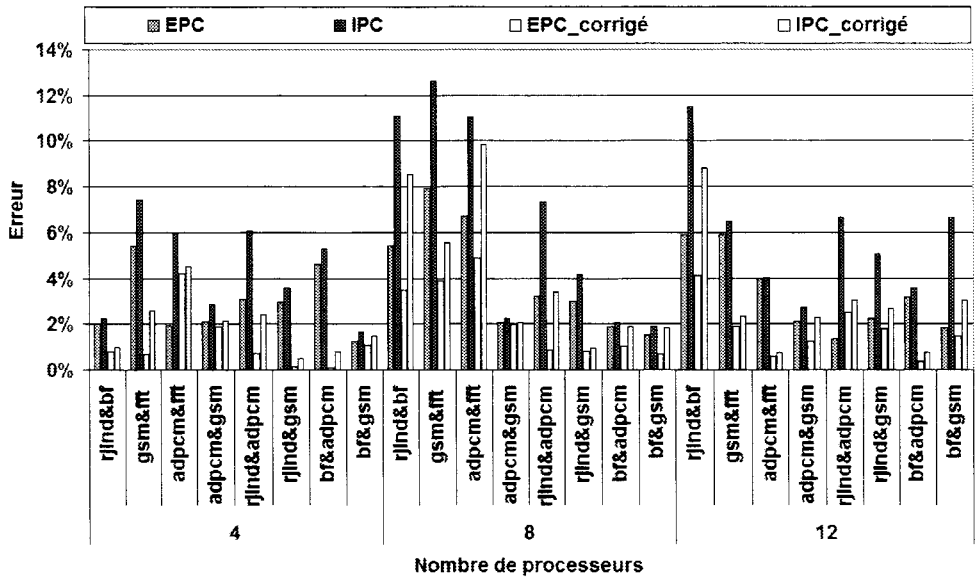


FIG. 5.11 – Comparaison de l'erreur de l'EPC avec celle de l'IPC et l'erreur de l'EPC corrigé avec celle de l'IPC corrigé pour un TWSB de 20%.

La figure 5.11 montre les erreurs dans l'estimation de l'EPC et de l'IPC obtenues par MGS pour les mêmes groupes d'applications que ceux testés précédemment. Comme nous l'avons indiqué, l'erreur dans l'estimation de l'EPC est plus faible que celle pour l'IPC. Celle-ci peut être trois fois plus grande, c'est le cas par exemple de rijnd&adpcm et bf&gsm sur 12 processeurs. L'erreur de l'IPC corrigé obtenue par la formule de correction (5.1) est aussi présentée dans la figure 5.11. Pour comparer l'erreur de l'IPC corrigé avec celle de l'EPC corrigé, nous appliquons la formule (5.1) de correction à l'EPC en remplaçant, dans cette formule, le nombre d'instructions exécutées par l'énergie estimée. L'erreur de l'EPC corrigée est montrée dans la figure 5.11. En effet, l'erreur corrigée dans l'estimation de l'EPC est plus faible que celle de l'IPC corrigé.

5.6.3 Comparaison de MGS à AS dans le cas des applications hétérogènes

Dans cette section, nous comparons MGS à AS dans le cas des applications parallèle hétérogènes. La figure 5.12 montre le facteur d'accélération donné par AS et par MGS pour des groupes d'applications exécutés sur 4, 8 et 12 processeurs. Le TWSB est égale à 20% pour les deux méthodes. Comme on peut le voir, MGS offre un facteur d'accélération plus important quand les applications qui s'exécutent en parallèle sont hétérogènes. En adoptant MGS, le facteur d'accélération peut être 25 fois plus grand que celui de AS (blf&adpcm sur 8 processeurs).

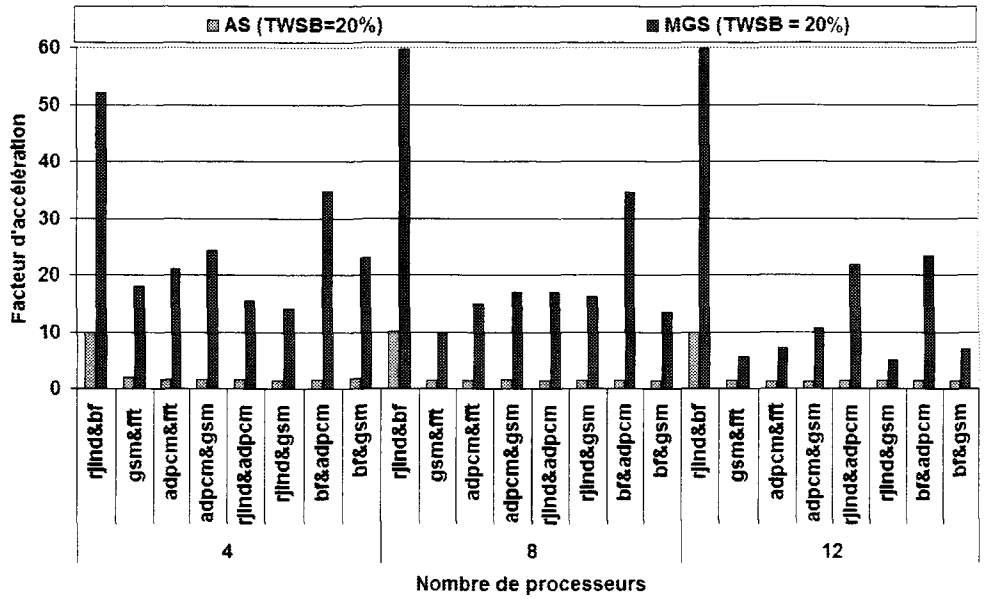


FIG. 5.12 – Comparaison du facteur d'accélération donné par AS (TWSB=20%) avec celui donné par MGS(TWSB=20%).Les applications sont exécutées sur 4, 8 et 12 processeurs.

Dans le cas de AS, le facteur d'accélération dépend du nombre de phases dans la séquence. Plus le nombre de phases dans la séquence est petit, plus grande sera la probabilité d'avoir des séquences qui se répètent. Ce qui va donner un facteur d'accélération plus grand. Par exemple, les combinaisons rijnd&blf ont le plus important facteur d'accélération, proche de 10. Ceci est dû au nombre de phases réduit pour rijndael et blowfish. Pour les autres applications, AS ne permet pas d'obtenir des résultats satisfaisants. Le facteur d'accélération est proche de 1.5. Cette faible accélération est causée par la difficulté de détecter la similarité entre les CSs avec de longues séquences de phases. La figure 3.15 qui est présentée dans la section 3.5.7 montre le nombre de phases dans les séquences des applications sur 4 processeurs. Ces applications correspondent à celles de la figure 5.12. Le nombre de phases dans les séquences atteint 20 pour adpcm et gsm dans le cas de rijnd&adpcm et rijn&gsm pour couvrir une seule phase de rijndael. La disparité entre les tailles des séquences de phases des applications parallèles réduit ainsi la répétition des CSs identiques. Les séquences contenant vingt phases avec AS sont représentées dans la méthode MGS par une seule phase avec une granularité de 20. Ceci facilite la détection des MPCs qui se répètent et permet d'obtenir un facteur d'accélération plus important. En conclusion, AS n'est pas efficace dans le cas des applications hétérogènes. La méthode MGS permet de résoudre ce problème en analysant différentes granularités.

5.6.4 Comparaison avec la méthode Cophase dans le cas des applications hétérogènes

Dans cette section, nous comparons les performances de MGS à celles de la méthode cophase proposée dans [21, 18] dans le cas des applications hétérogènes. Les paramètres de cophase utilisés sont les mêmes que ceux utilisés dans le cas de la comparaison avec AS

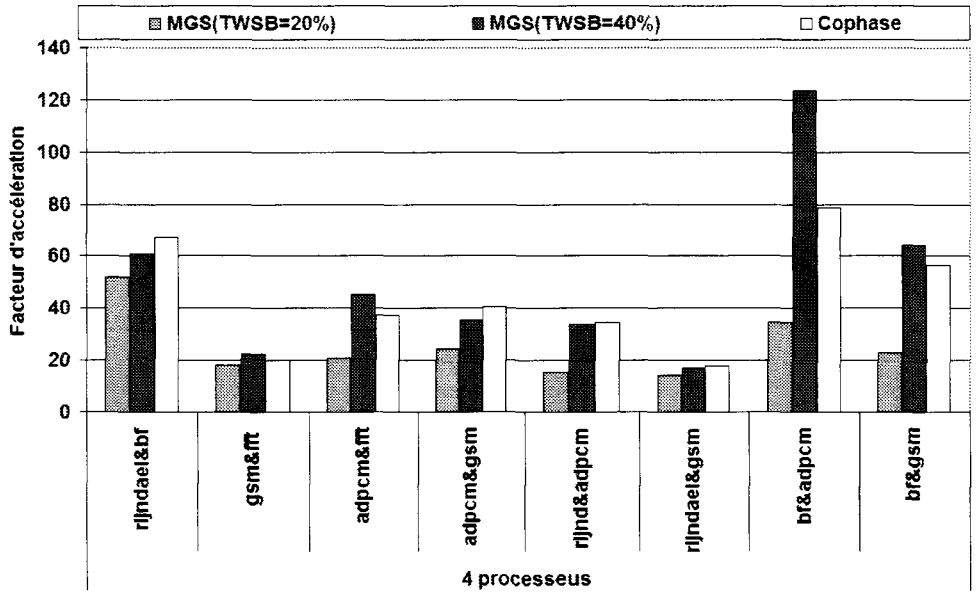
(voir la section 3.5.6). Dans chaque échantillon, chaque processeur exécute au moins 25K instructions pour des tailles d'intervalles de 50K instructions. Nous nous intéressons ici à l'accélération, donc pour cophase nous avons choisi la plus grande période (période = 100) pour re-simuler les mêmes cophases. Pour MGS, nous avons utilisé un TWSB de 20% et 40%.

La figure 5.13 donne le facteur d'accélération (en log) obtenu par MGS avec un TWSB=20% et 40% et par cophase sur 4, 8 et 12 processeurs. Comme on peut le voir, dans le cas de 4 processeurs, le facteur d'accélération de cophase est généralement plus important que celui de MGS (TWSB=20%). Tandis que pour 8 et 12 processeurs la méthode cophase n'accélère pas la simulation de façon sensible. Cette faible accélération de cophase est due au nombre important de combinaisons de phases générées et à la nécessité de resimuler les cophases.

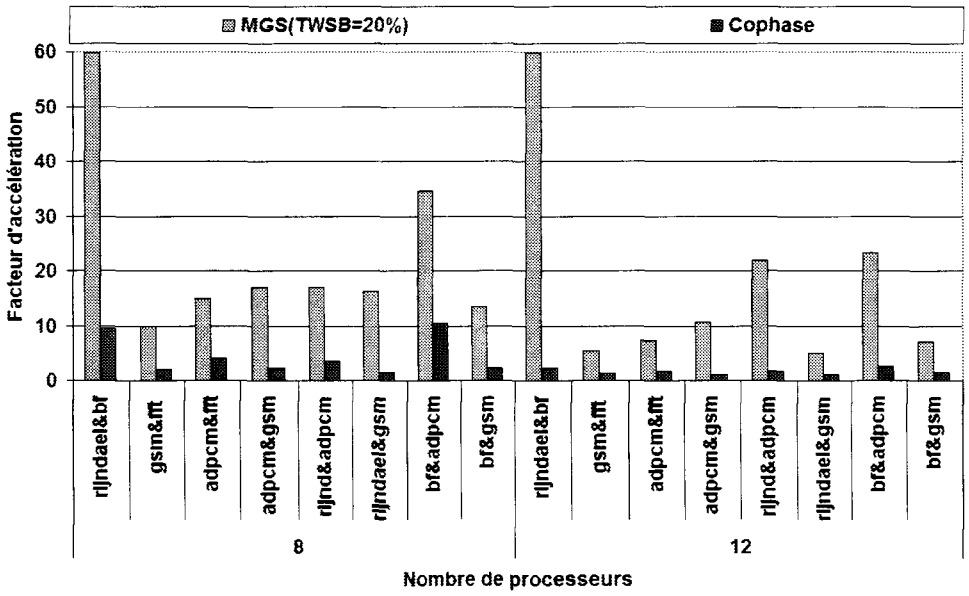
La figure 5.14 montre le nombre de combinaisons de phases donné par MGS et par cophase sur 4, 8 et 12 processeurs. Ce nombre correspond à celui de cophases représentatives (appelé aussi nombre de cophases observées) pour la méthode cophase et au nombre de MPCs représentatifs pour MGS. Rappelons que, dans le cas de cophase, le nombre de scénarios de phases conjoints est important. De même, ce nombre augmente avec l'accroissement du nombre de processeurs. Ce qui explique l'augmentation du nombre de combinaisons de phases avec celle du nombre de processeurs (voir figure 5.14). Dans le cas de MGS, le nombre de MPCs représentatifs est généralement constant avec l'augmentation du nombre de processeurs. Les barrières de simulation permettent la synchronisation entre les applications concurrentes. Ce qui réduit le nombre de scénarios de phases conjoints réduisant ainsi le nombre de MPCs représentatifs. Par conséquent, le nombre de cophases représentatives est plus important et dépasse 10000 tandis que le nombre de MPCs représentatifs est généralement inférieur à 100. Pour 4 processeurs, le nombre de cophases représentatives est légèrement plus grand que le nombre de MPCs représentatifs, mais le nombre d'instructions qui subissent la simulation détaillée dans le cas de MGS pour TWSB=20% est plus important que celui de cophase. Ceci explique l'importance de l'accélération de cophase par rapport à celle de MGS pour un TWSB de 20% sur 4 processeurs (voir figure 5.14).

La figure 5.15 montre les deux erreurs, erreur sur l'IPC sans correction et erreur sur IPC avec correction, obtenues par MGS (TWSB = 20% et TWSB = 40%) et l'erreur sur IPC obtenue par cophase. Les applications sont exécutées sur 4 processeurs. Les erreurs pour 8 et 12 processeurs ne sont pas montrées car cophase n'accélére plus pour ces deux configurations. En conclusion, de ces deux figures, nous pouvons affirmer que le facteur d'accélération de cophase est plus important que celui de MGS avec un TWSB de 20% sur 4 processeurs (voir figure 5.13(a)) mais l'erreur de cophase est également plus importante. En particulier quand on applique une correction. L'erreur de cophase dépasse généralement les 10% tandis que l'erreur corrigée dans MGS (TWSB=20%) est inférieure à 5%.

Si nous utilisons un TWSB à 40%, le facteur d'accélération de MGS augmente et dépasse celui obtenu par cophase (voir figure 5.13(a)). En terme de précision, l'erreur corrigée sur l'estimation de l'IPC avec un TWSB de 40% est généralement plus petite que l'erreur fournie par cophase. Comme on peut le voir dans la figure 5.15, l'erreur de l'IPC sans correction et l'erreur de l'IPC avec correction sont repectivement inférieures à 12% et à 6%, tandis que l'erreur de cophase atteint 16%.



(a) Facteur d'accélération obtenu par MGS (TWSB=20% et TWSB=40%) et par cophase. Les groupes d'applications sont exécutés sur 4 processeurs.



(b) Facteur d'accélération obtenu par MGS (TWSB=20%) et par cophase. Les groupes d'applications sont exécutés sur 8 et 12 processeurs.

FIG. 5.13 – Facteur d'accélération donné par MGS et par cophase.

5.7 Conclusion

Dans ce chapitre, nous avons présenté la méthode de simulation par intervalles de granularités variables MGS. Cette méthode permet une estimation plus précise de la performance

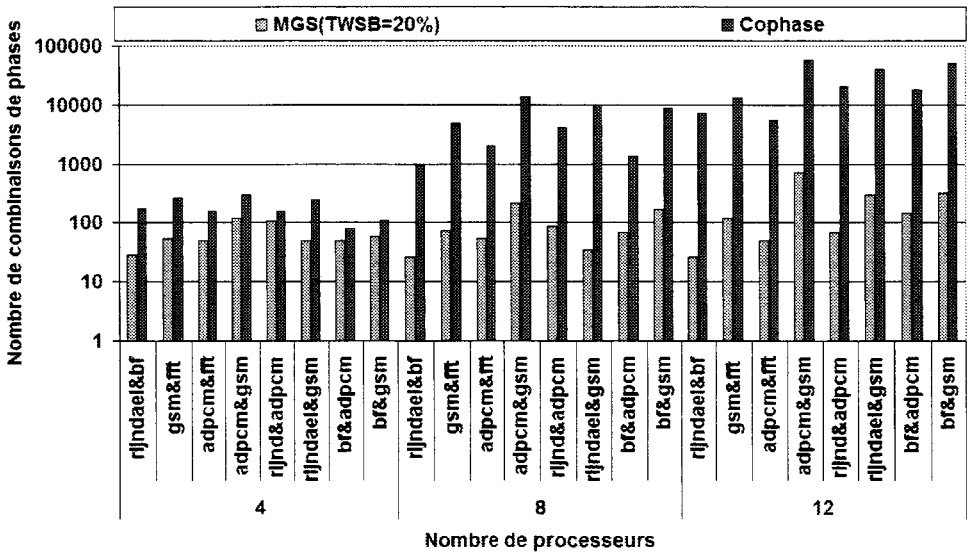


FIG. 5.14 – Nombres de combinaisons de phases (en log) donnés par MGS (TWSB=20%) et par cophase. Les groupes d'applications sont exécutés sur 4, 8 et 12 processeurs.

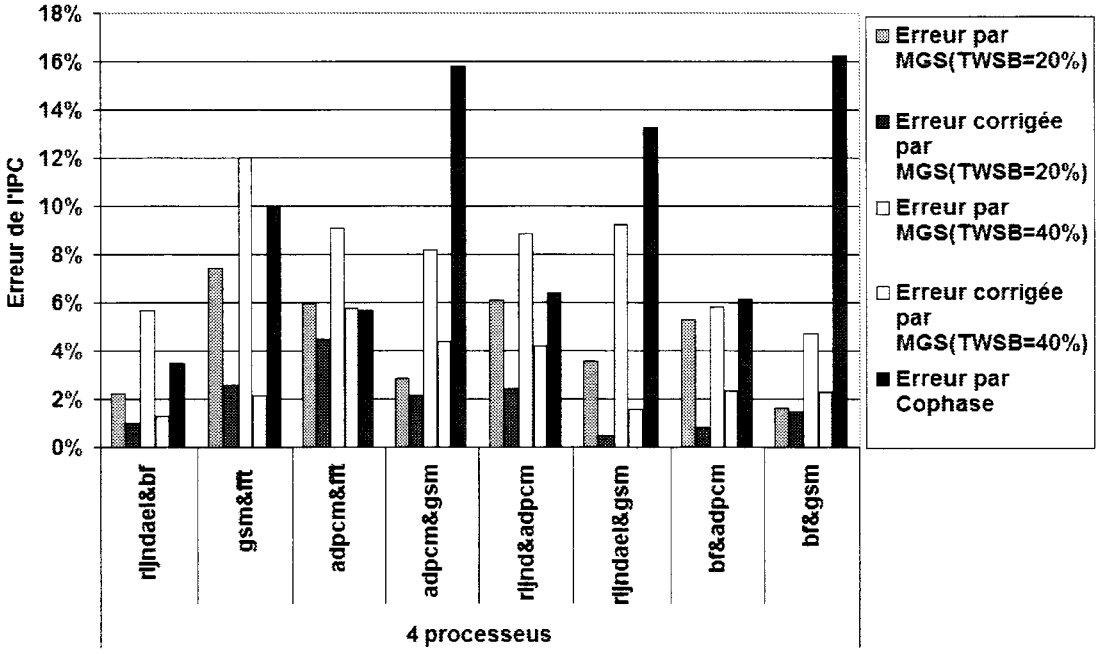


FIG. 5.15 – Les deux erreurs, erreur sur l'IPC sans correction et erreur de l'IPC avec correction, obtenues par MGS (TWSB = 20% et TWSB = 40%) et l'erreur sur l'IPC obtenue par cophase. Les applications sont exécutées sur 4 processeurs.

tout en accélérant la simulation des applications hétérogènes concurrentes. Elle permet de recouvrir les phases s'exécutant en parallèle dans un cluster de telle sorte que chaque clus-

ter contient une phase par application. La granularité de chaque phase est appropriée selon le comportement dynamique de l'application correspondante. Seuls les nouveaux recouvrements de phases qui n'apparaissent pas auparavant durant la simulation subissent la simulation détaillée. Cette réduction du nombre de phases par cluster simplifie la détection des recouvrements qui se répètent, ce qui permet d'augmenter l'accélération de la simulation.

Dans le cas des applications homogènes concurrentes, nous avons remarqué que MGS accélère la simulation plus que AS, alors que As est plus précise. Dans le cas des applications hétérogènes concurrentes, AS n'accélère pas la simulation.

L'accélération obtenue par Cophase est, elle aussi, relativement faible quand le nombre de processeurs simulé dépasse 4. A titre d'exemple, lorsque les applications choisies sont rijndael et blowfish sur 12 processeurs, MGS est 20 fois plus rapide que Cophase. En effet, la figure 5.13(b) indique que l'accélération par MGS est de 60 alors que celle obtenue par Cophase n'est que de 3.

L'erreur dans l'estimation des performances est en général acceptable. Elle est inférieure à 10% pour la plupart des applications étudiées. Une formule de correction de l'estimation de l'IPC est aussi proposée. La formule en question diminue l'erreur jusqu'à 90%. L'avantage le plus important de MGS demeure dans le fait que l'accélération de la simulation s'adapte dynamiquement selon l'interaction des applications hétérogènes sans intervention du concepteur du système. Cela est fait tout en maintenant la précision de la performance.

Grâce à la méthode MGS, nous avons obtenu une accélération de la simulation tout en gardant un bon niveau de précision pour les applications homogènes ou hétérogènes concurrentes. Dans le chapitre suivant, nous étudierons la construction de l'état du contexte du programme et l'état des composants architecturaux (exemple les mémoires cache) au début de la simulation détaillée de chaque recouvrement de phases après la réalisation du saut des recouvrements répétés de phases.

Chapitre 6

Utilisation du Checkpointing pour la simulation des MPSoC

6.1 Introduction

Comme il a été mentionné dans le chapitre 2, l'utilisation de l'échantillonnage d'application nécessite de résoudre deux problèmes avant de démarrer la simulation d'un intervalle. Il s'agit de trouver un moyen pour restaurer l'image exacte du contexte de l'application et un moyen pour restaurer l'état micro-architectural (mémoire cache, prédicteur de branchements). Ces deux tâches doivent être faites au démarrage de chaque intervalle simulé. L'image du système est construite soit par une simulation fonctionnelle, soit par chargement du checkpoint du point correspondant. Généralement, le checkpointing est plus rapide que la simulation fonctionnelle. Cette dernière limite en effet l'accélération de la simulation de la méthode par échantillonnage [62]. Néanmoins, l'inconvénient du checkpointing demeure le nombre important de checkpoints nécessitant un espace important de stockage sur le disque et un délai important de chargement du checkpoint pendant la simulation.
Dans les chapitres précédents, ces deux points n'ont pas été abordés. Aussi, nous avons supposé que :

1. Le coût de la restauration du contexte du programme et de l'état micro-architectural est nul.
2. La restauration de l'état micro-architectural est parfaite. Autrement dit, le contenu des mémoires, des structures du pipeline, ..., etc, est le même que si nous avons réalisé une simulation détaillée.

Par conséquent, les accélérations que nous avons présentées correspondent à un idéal.

Dans ce chapitre, nous prenons en compte ce défaut. Nous tentons de montrer aussi que la méthode du checkpointing est une méthode viable et efficace pour la restauration des contextes du programme et de l'état micro-architectural. Ainsi, nous proposons deux solutions originales pour contourner le principal problème de cette méthode relatif à la taille mémoire nécessaire. Nos deux méthodes visent simplement à réduire le nombre de checkpoints de façon sensible. Cette réduction permet un préchargement des checkpoints dans la mémoire centrale. Ce qui diminue le temps de chargement d'un checkpoint, d'où une amélioration de l'accélération de la simulation, offrant ainsi des facteurs d'accélération quasiment identiques à ceux des chapitres précédents.

6.2 Construction de l'image du système par checkpointing

Afin d'obtenir une accélération importante de la simulation, la méthode par échantillonnage doit non seulement réduire le nombre d'instructions exécutées dans le mode détaillé mais doit aussi réduire le temps nécessaire pour faire avancer la simulation de la fin de l'échantillon courant jusqu'au prochain échantillon à simuler. Ainsi une technique pour construire l'état du système au démarrage de chaque échantillon dans le mode de simulation détaillée est nécessaire.

L'image du système associée à un échantillon contient par définition : l'état architectural nommé aussi *"sample starting image"* et l'état micro-architectural nommé aussi *"sample warming-up"*. L'état architectural est représenté par les valeurs des données se trouvant dans les registres et dans la mémoire partagée de notre architecture MPSoC. Ces données représentent le contexte de l'application et elles sont indépendantes de la configuration architecturale. L'état micro-architectural, quant à lui, correspond au contenu des mémoires caches,

de la table de prédiction de branchements, du pipeline, etc. [19, 62]. Comme les processeurs ARM7, que nous utilisons durant les expériences adoptent la politique de la prédiction de branchement statique *"non-pris"*, les états relatifs aux prédictions de branchement ne sont pas inclus dans les checkpoints et l'état micro-architectural est uniquement représenté par le contenu des mémoires cache. Néanmoins, notre méthode s'applique aussi aux architectures des processeurs plus évoluées (ARM9 et ARM11 par exemple).

Chacun de ces deux états est construit soit par une simulation fonctionnelle soit par checkpointing. La simulation fonctionnelle consiste à avancer rapidement la simulation vers un point donné tout en négligeant les détails architecturaux du système. Dans ce mode, les instructions du programme sont exécutées et le contexte du programme est mis à jour, mais aucun détail micro-architectural n'est pris en compte. Ce type de simulation permet de construire le contexte exact du programme tandis que l'état micro-architectural a besoin d'une technique de pré-chauffement supplémentaire. Généralement, la vitesse de la simulation fonctionnelle limite l'accélération potentielle de la méthode d'échantillonnage. A l'opposé, le checkpointing réduit le temps de simulation en éliminant le temps consommé par la simulation fonctionnelle [62] entre les échantillons simulés en détails. Le checkpointing consiste à sauvegarder une image, concernant les deux états cités précédemment, pour chaque point de démarrage de la simulation détaillée. Si durant la simulation, une image a été demandée pour un point de démarrage donné, l'image est restaurée en chargeant le checkpoint correspond. Les checkpoints sont collectés en simulant en détail l'application toute entière une seule fois pour toute l'opération du DSE. Traditionnellement, les points de démarrage de simulation des intervalles ne sont pas connus avant le lancement de la simulation. Ce problème existe dans le cadre de la méthode AS. Ainsi le checkpoint doit être généré pour chaque intervalle de l'application. Les checkpoints collectés sont sauvegardés sur le disque. Ainsi, plus grand est le nombre d'intervalles de ou des applications, plus grande sera la taille mémoire nécessaire pour stocker les checkpoints. Ce problème est d'autant plus grave quand la taille d'intervalle est réduite.

A l'opposé, notre méthode permet d'utiliser le checkpointing avec MGS et détecte à priori les points de démarrage de simulation détaillée. Ce qui réduit le nombre de checkpoints à collecter réduisant à l'occasion l'espace du stockage nécessaire. Cela permet un pré-chargement des checkpoints dans la mémoire diminuant ainsi le temps de chargement d'un checkpoint et améliorant l'accélération de la simulation. La section 6.2.2 explique en détail notre méthode qui applique le checkpointing avec MGS. Cette méthode dépend néanmoins des ordres de granularités analysées. Pour cela, nous proposons dans la section 6.3 une deuxième méthode qui traite le même problème indépendamment des granularités analysées et tout en permettant aux utilisateurs de contrôler à priori le nombre maximum de checkpoints nécessaires à collecter.

6.2.1 Techniques de construction des checkpoints

Dans cette section nous introduisons la technique de checkpointing que nous expérimentons dans ce chapitre. Comme il a été mentionné précédemment, un checkpoint doit permettre de rétablir à la fois le contexte du programme et l'état micro-architectural.

Le premier point concerne les valeurs des données en mémoire et les contenus des registres.

Afin de réduire la taille mémoire du checkpoint, nous avons choisi d'utiliser une technique qui ne prend pas en compte les régions de la mémoire non utilisées. Cette technique est

proche de la technique *"Touched Memory Image"* TMI [19] présentée dans la section 2.5.1. Ainsi, au lieu de sauvegarder l'adresse mémoire de chaque valeur de donnée, une seule adresse mémoire est utilisée afin de représenter une séquence de données consécutives en mémoire. De cette façon, chaque séquence de valeurs de données est représentée au niveau du checkpoint par :

1. L'adresse mémoire de la première valeur dans la séquence.
2. Le nombre de valeurs dans la séquence.
3. Les données qui composent la séquence.

Cette façon de représenter le checkpoint permet de réduire la taille mémoire et le temps nécessaire à la construction de l'image mémoire au point correspondant.

Pour le deuxième point, qui concerne l'état micro-architectural, la taille du checkpoint dépend de la configuration architecturale à simuler. Afin de collecter ce type de checkpoint une fois pour toute l'opération de DSE, nous avons choisi d'utiliser la technique *"Memory Hierarchy State"* MHS [19] expliquée dans la section 2.5.2. Ce type du checkpoint est collecté pour une seule taille de bloc de cache et pour une seule politique de remplacement de bloc. Le contenu de la mémoire cache est crée à partir d'une mémoire cache possédant une taille plus grande. Cette mémoire cache est appelée "cache de référence". Elle a une taille de 64 KOctets dans notre cas. Cette valeur représente un maximum par rapport à ce que nous pouvons trouver dans les processeurs embarqués actuels. MHS permet de construire l'état de n'importe quelle mémoire cache ayant une taille inférieure à celle du cache de référence. MHS diminue aussi l'espace de stockage ainsi que le temps nécessaire pour construire le cache demandé. Notre cache de 64 KOctet contient 4096 blocs de 16 Octets chacun. Chaque bloc est représenté au niveau du checkpoint par la partie étiquette de l'adresse (4 octets), un bit de validité, un compteur sur 4 octets (il y a 4096 blocs) afin de gérer la politique de remplacement et enfin les données sur 16 octets. Ainsi chaque bloc du cache de référence nécessite 25 octets. Dans notre cas, la taille du checkpoint de l'état micro-architectural correspondant au contenu d'un cache de référence nécessite 100 Koctets ($25x4096$).

6.2.2 Implémentation du Checkpointing présenté pour MGS

La méthode MGS que nous avons présentée dans le chapitre précédent repose sur l'utilisation de la matrice de phases. Une analyse de cette matrice révèle une similarité importante de ses lignes. Nous considérons que deux lignes sont similaires si elles ont des phases identiques au niveau de chaque ordre de granularité de la matrice. Ainsi les lignes similaires ont le même comportement au niveau de chaque granularité. Cette similarité des lignes est exploitée de façon à collecter un nombre réduit de checkpoints au lieu de collecter le checkpoint pour tous les intervalles. Ces checkpoints au nombre réduit sont appelés "checkpoints représentatifs" dans la suite de ce chapitre. Puisque les phases d'une même ligne ont un même point de démarrage et puisque leur comportement est le même quelque soit la granularité, le checkpointing est effectué au point du démarrage de la première occurrence de chaque groupe de lignes similaires. De cette façon, durant la simulation, quand un MPC est simulé en détail, le checkpoint associé à la première occurrence du groupe de lignes similaires est chargé. Ainsi les deux états, architectural et micro-architectural, sont restaurés à ce point de la simulation. Les instructions appartenant aux phases de la première occurrence de la ligne sont exécutées comme des représentatifs du comportement des lignes similaires.

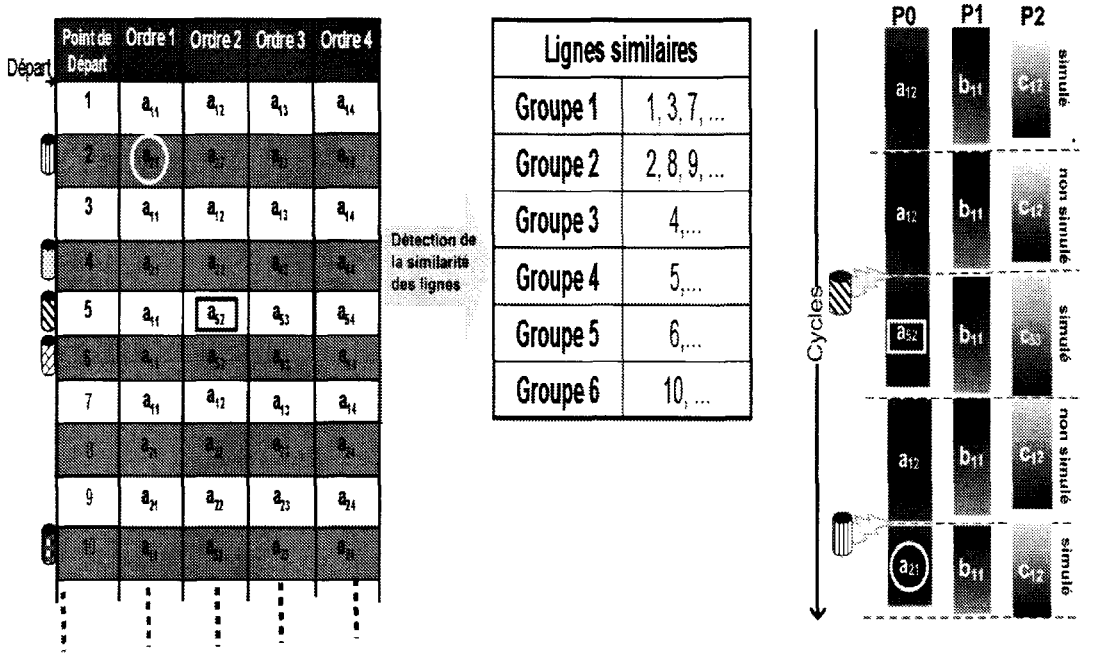


FIG. 6.1 – Détection des groupes de lignes similaires de la matrice générée avec MGS. Le nombre de checkpoints est réduit en prenant un checkpoint par groupe au lieu de prendre un checkpoint par intervalle.

La figure 6.1.B montre les groupes de lignes similaires détectés après l'analyse de la matrice de phases de l'application "a" de la figure 6.1.A. Le nombre de groupes détectés est 6. Ainsi 6 checkpoints sont collectés pour la première occurrence de chaque groupe. La figure 6.1.A représente par un cylindre les points de démarrage où le checkpoint sera formé. Cet exemple montre aussi la réduction en nombre de checkpoints en prenant un checkpoint par groupe au lieu de prendre un checkpoint par intervalle.
Dans le cadre de la méthode AS, il n'est pas possible de connaître à priori les points de démarrage de la simulation détaillée. Ainsi pour appliquer le checkpointing dans le cadre de AS, nous sommes obligés de collecter les checkpoints au niveau de chaque intervalle. Dans ce cas, le nombre de checkpoints qui doivent être collectés correspond au nombre d'intervalle d'ordre 1. Ainsi l'utilisation conjointe des deux techniques, MGS et checkpointing, parait intéressante puisque d'une part, il n'est pas nécessaire de disposer d'une zone mémoire importante pour stocker les checkpoints et d'autre part, ces deux méthodes permettent de réduire le temps de simulation de façon sensible. En effet avec MGS, seul un nombre réduit de phases est simulé (voir chapitre précédent) et avec notre méthode de démarrage de la simulation, il est possible de stocker les checkpoints au niveau de la RAM de la plateforme hôte.

La figure 6.1.C montre la génération des MPCs pour trois applications concurrentes : a, b et c. Après le deuxième MPC non simulé, P0 a besoin de construire l'état exact du système, le checkpoint qui correspond au point 5 de démarrage est chargé et la phase a_{52} est exécutée. De même après le quatrième MPC non simulé, P0 a besoin de construire l'état du système

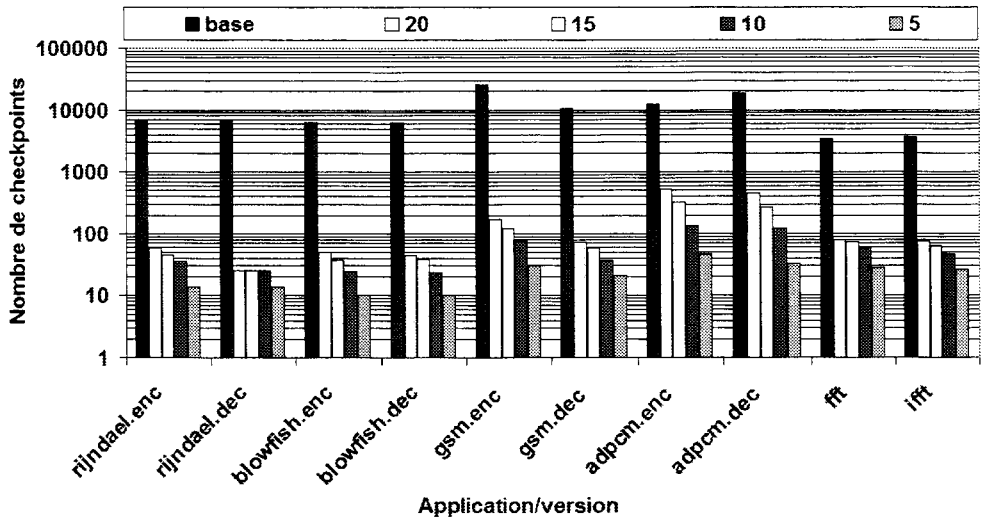


FIG. 6.2 – Variation du nombre de checkpoints avec l'ordre de granularité maximal autorisé. Le cas de base correspond au nombre de checkpoints qui va être sauvegardé dans le cas où la similarité des lignes n'est pas considérée.

au neuvième point de démarrage. Comme la ligne 9 et la ligne 2 sont considérées similaires, elles appartiennent au même groupe de lignes similaires (voir figure 6.1.B). La ligne 2 est la première occurrence du groupe. Ainsi le checkpoint représentant l'état du système au point 2 de démarrage est donc représentant de l'état du système au neuvième point de démarrage. Autrement dit le checkpoint qui correspond au point 2 de démarrage est chargé et les instructions de la ligne 2 sont exécutées. Dans l'exemple, c'est la phase a_{21}, de la ligne 2, qui est exécutée pour représenter le comportement de la phase a_{21} du point 9. Au niveau de la méthode MGS rien ne change, le système se comporte comme si la phase a_{21} du point de démarrage 9 est réellement exécutée. Ainsi les autres points sont traités de la même façon, d'où la réduction du nombre de checkpoints permettant ainsi un gain dans l'espace de stockage des checkpoints. Ce gain sera évalué dans la section suivante.

6.2.3 Gain en stockage mémoire du checkpointing avec MGS

Dans cette section, nous évaluons le gain obtenu par notre méthode du checkpointing pour la méthode MGS présentée dans le chapitre précédent. La quantité de stockage est calculée pour chacune des applications de Mibench.

6.2.3.1 Réduction du nombre de checkpoints

Dans le cadre de MGS, seuls les checkpoints représentatifs sont collectés au lieu de tous les checkpoints de tous les intervalles. Nous considérons que deux checkpoints pour deux points de démarrage du programme sont similaires si les phases au niveau de chacune des granularités dans la matrice des phases sont identiques (voir section 6.2.2).

La figure 6.2 montre la variation du nombre de checkpoints quand l'ordre de granularité maximal autorisé varie entre 5 et 20. Cette valeur 20 est la plus grande granularité observée

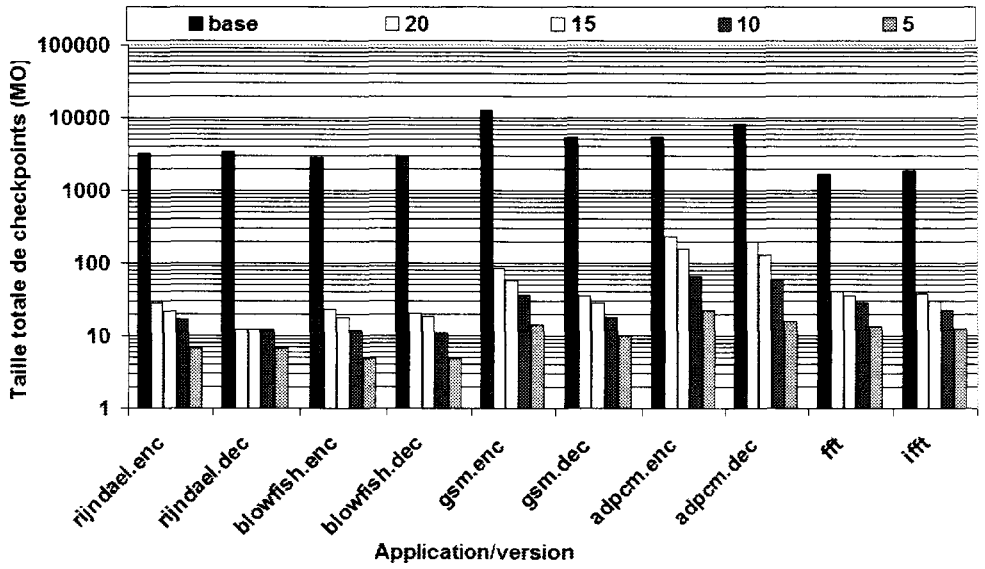


FIG. 6.3 – Variation de la taille totale en MO de checkpoints avec ordres de granularité. Les ordres de granularité varient entre 5 et 20. Le cas de base correspond au cas où la similarité des lignes n'est pas considérée.

pour les applications de Mibench et un TWSB de 20%. Cette figure donne les résultats expérimentaux pour 10 benchmarks. Nous avons aussi mesuré le nombre de checkpoints "sans prise en compte de la similarité des lignes" noté "base" dans la figure. En prenant compte de la similarité des lignes, le nombre de checkpoints depend de la granularité maximale autorisée dans la simulation.
Dans le mode "base", le nombre de checkpoints correspond simplement au nombre d'intervalles, il est donc indépendant de la granularité. Ainsi le mode "base" correspond au nombre de checkpoints avec la méthode AS.

Dans la figure 6.2, nous observons que le nombre de checkpoints s'accroît avec l'augmentation de la granularité dans la matrice. Cette valeur augmente dans le cas où les applications concurrentes sont hétérogènes. Néanmoins, malgré cette augmentation, le nombre de checkpoints nécessaires est plus petit que dans le cas "base". Le nombre de checkpoints est réduit de 99% quand l'ordre de granularité avec MGS est égal à 20.

6.2.3.2 Réduction de la taille mémoire de checkpoints

Les deux techniques utilisées pour construire les checkpoints avec l'état architectural et l'état micro-architectural ont été présentées dans la section 6.2.1. La taille d'un checkpoint correspond à la taille des données nécessaires pour construire l'image mémoire, les contenus des registres et des deux caches de données et d'instructions.

La figure 6.3 montre la taille nécessaire pour sauvegarder tous les checkpoints d'une application avec quatre valeurs différentes de granularité. Comme le nombre d'intervalles est très élevé, la taille mémoire pour stocker les checkpoints dans le mode "base" peut dépasser 10 GO pour une seule application. Tandis qu'avec MGS, le nombre de checkpoints est généralement inférieur à 100. Une taille mémoire de stockage de 90 MO est suffisante pour

sauvegarder tous les checkpoints. Ce qui donne une réduction supérieure à 98% de la taille mémoire pour le stockage des checkpoints avec MGS.

Un nombre de checkpoints relativement petit permet par ailleurs de sauvegarder les checkpoints en mémoire centrale RAM au lieu de les stocker sur disque. Ce qui va réduire aussi le temps nécessaire pour restaurer le checkpoint et construire l'état du système rapidement à un point donné de la simulation. Ceci réduit le temps de la simulation avec MGS. Ces deux avantages font de MGS une méthode viable et efficace pour l'accélération de la simulation des MPSoC.

6.2.3.3 Comparaison entre le checkpointing et la simulation fonctionnelle

Afin de comparer les deux méthodes de construction du contexte par la simulation fonctionnelle et par checkpointing, nous avons utilisé l'outil perfmon [34]. Grâce à perfmon nous avons mesuré le temps d'exécution sur le processeur hôte pour faire avancer la simulation d'un point à un autre. Ce temps est calculé en nombre de cycles sur la machine hôte. Intuitivement, le checkpointing est performant quand un grand nombre d'intervalles doivent être sautés. A l'opposé la simulation fonctionnelle peut être intéressante quand la taille de la partie non simulée (sautée) est réduite.

Notre but est de comparer le nombre de cycles sur la machine hôte de la simulation fonctionnelle avec celui du checkpointing. Les résultats expérimentaux montre qu'en moyenne 10 millions cycles sont consommés par la machine hôte pour restaurer un checkpoint de la mémoire RAM et construire les deux états architectural et micro-architectural d'un processeur à un point donné. En considérant que la simulation fonctionnelle est d'environ 25 fois plus rapide que la simulation détaillée [22], nous pouvons dire que la simulation fonctionnelle de 50K instructions nécessite 242 millions cycles sur la machine hôte. La taille de 50K instructions est choisie car elle correspond à la taille du plus petit intervalle à sauter. Ces résultats nous permettent de déduire que le saut d'intervalles, même avec de petites tailles (50K instructions), est plus rapide avec le checkpointing qu'avec la simulation fonctionnelle sur notre environnement de simulation. Le checkpointing est au moins 24 fois plus rapide que la simulation fonctionnelle.

6.3 Matrices des blocs de base pour le checkpointing

Comme cela a été mentionné précédemment, la réduction du nombre de checkpoints permet de stocker les checkpoints en mémoire centrale. Cet avantage réduit le temps de restauration des checkpoints et donne un temps de simulation plus court. La figure 6.2 montre que le nombre de checkpoints est de l'ordre de 100 checkpoints. Ce nombre, qui peut être coûteux en termes d'espace de stockage quand le nombre de processeurs simulés est élevé, dépend des granularités analysées avec MGS. En effet l'augmentation de la granularité analysée correspond à une augmentation du nombre de phases dans les lignes, ce qui diminue la probabilité de trouver des lignes similaires dans la matrice, d'où l'augmentation du nombre de checkpoints. Cette augmentation de granularité est nécessaire en particulier pour les systèmes embarqués car leurs applications concurrentes peuvent être hétérogènes. De plus les granularités qu'il faut analyser ne sont pas connues à priori. Dans ces conditions l'utilisateur sera obligé de les surestimer. Ce qui augmente le nombre de groupes de lignes similaires et augmente donc le nombre de checkpoints. Le risque dans ce cas est de stocker des

checkpoints qui ne seront jamais utilisés. Ainsi il est intéressant de réduire le nombre de checkpoints indépendamment des granularités analysées.

Nous avons proposé une deuxième méthode permettant de rendre le nombre de checkpoints indépendant des granularités analysées. Cette méthode consiste à regrouper les BBVs représentant des intervalles des différentes granularités dans une seule Matrice. Par la suite, la similarité entre les matrices générées est détectée, regroupant ainsi ces dernières dans des groupes ou phases. Finalement une seule matrice est sélectionnée pour représenter la phase toute entière. Le checkpoint sera collecté pour chacune des matrices sélectionnées collectant ainsi un seul checkpoint pour chaque phase. Nous avons utilisé la méthodologie de SimPoint pour détecter la similarité des matrices et pour sélectionner la matrice qui représente chaque phase. La méthode proposée a les avantages suivants :

1. Le nombre réduit de checkpoints est indépendant des granularités analysées. En effet l'utilisateur est capable de préciser à priori un nombre maximum de checkpoints nécessaires à collecter.

2. Les instructions correspondant aux matrices qui représentent les phases vont être simulées. Ceci réduit l'erreur provoquée par l'approximation des IPCs des MPCs avec ceux des MPCs représentatifs avec MGS.

En effet, Comme nous l'avons expliqué dans la section 2.3.2.4, Simpoint [57] détecte la similarité des BBV (Basic Block Vector), mais notre méthode nécessite la détection de la similarité des matrices. Ainsi une modification de la méthodologie de SimPoint est nécessaire pour que cette dernière soit utilisée. La méthode ainsi que l'utilisation de la méthodologie de SimPoint sont détaillées dans les sections suivantes.

6.3.1 Génération des matrices de blocs de base

Rappelons qu'avec la méthode MGS, les intervalles appartenant à une même ligne de la matrice de phases représentent différentes granularités. Il y a autant d'intervalles qu'il y a de granularités choisies à priori. L'idée est de combiner les BBVs des intervalles appartenant à une même ligne dans une matrice nommée BBM pour "Basic Block Matrix". La figure 6.4.A montre la construction des BBMs à partir des BBVs avec une granularité égale à 3. Il y a autant de BBMs que de lignes dans la matrice de phases. Toutes les BBMs construites forment une matrice nommée *"matrice de BBMs"* présentée dans la figure 6.4.B. La taille de chaque BBM est de gxn, g et n correspondent respectivement aux ordres de granularité analysées et au nombre total de blocs de base de l'application. Dans la figure 6.4, g vaut 3.

Pour détecter les phases de l'application les BBMs doivent être comparées entre elles afin de trouver la similarité. Dans notre cas cette comparaison est identique à la comparaison des BBVs utilisée par SimPoint. Ainsi la comparaison entre deux BBMs est effectuée en appliquant la distance euclidienne entre deux matrices. La formule (6.1) qui permet de calculer la distance euclidienne des deux BBMs, BBM_a et BBM_b chacune de taille gxn, est la suivante :

$$Dist\_Euclidienne(BBM_a, BBM_b) = \sqrt{\sum_{i=1}^{g}\sum_{j=1}^{n}(a_{ij} - b_{ij})^2} \tag{6.1}$$

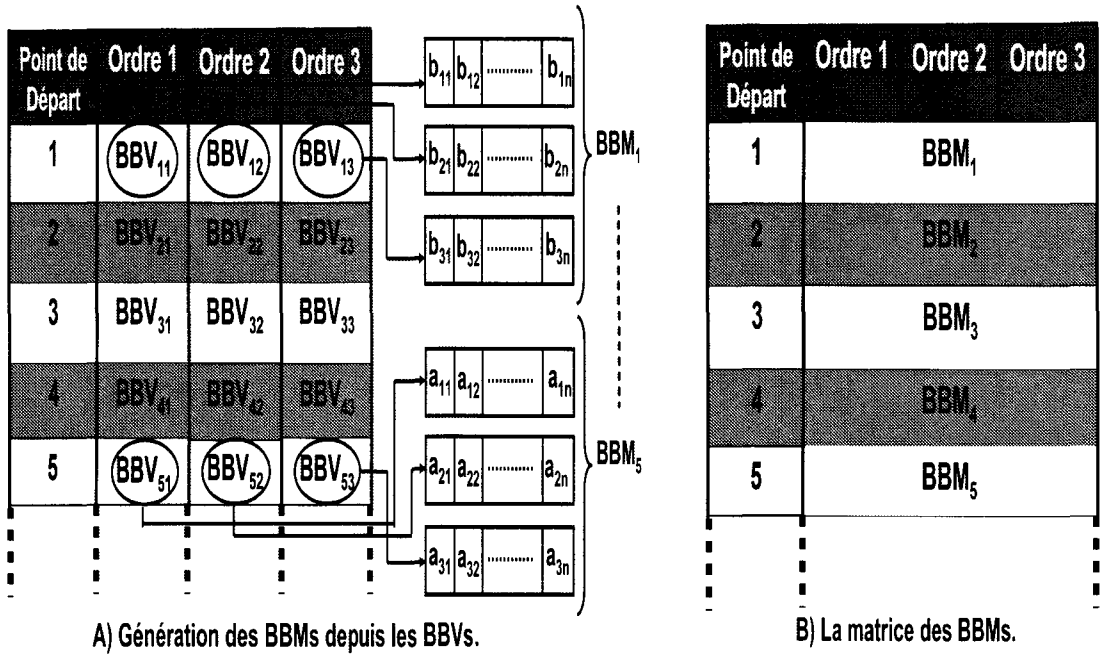


FIG. 6.4 – Génération des BBMS à partir des BBVs des intervalles appartenant aux mêmes lignes de la matrice.

6.3.2 Classification des BBMs

Une fois que le degré de similarité à été calculé, les BBMs similaires sont regroupées dans un seul groupe nommé dans notre cas "Mphase". Une seule BBM, nommée "BBM représentative", est choisie pour représenter chaque Mphase. Ceci nous permet de représenter précisément l'exécution de l'application toute entière. En effet, le but de la classification est de répartir les BBMs dans des Mphases de telle sorte que les BBMs appartenant à une même Mphase sont similaires, et les BBMs dans des Mphases différentes sont différentes l'une de l'autre.

En effet dans ce travail nous utilisons la méthodologie de SimPoint pour réaliser la classification des BBMs. Néanmoins, tout autre méthodologie de classification [37] peut être utilisée pour cette classification. La méthodologie de SimPoint expliquée dans la section 2.3.2.4, est basée sur l'algorithme de classification de k-means. C'est un algorithme qui exige que le nombre maximum de phases nommé "**Max_K**" soit choisi à priori par l'utilisateur. L'application est d'abord profilée afin de générer les BBMs. Par la suite ce sont les étapes suivantes qui sont appliquées :

1. Simpoint utilise la projection linéaire aléatoire pour réduire la dimension de chaque BBV à 15 dans le but de rendre la classification faite par k-means plus rapide. Cette projection est appliquée aux BBVs qui forment les BBMs avant que ces dernières subissent la classification.
2. Appliquer l'algorithme de classification k-means sur les BBMs de dimension réduit pour k allant de 1 à Max_K, k étant le nombre de Mphases. Chaque exécution de k-means réalise une distribution des BBMs dans k Mphases différentes. Pour choisir la bonne valeur de k, Simpoint utilise le *"Bayesian Information Criterion"* (BIC) [39] pour

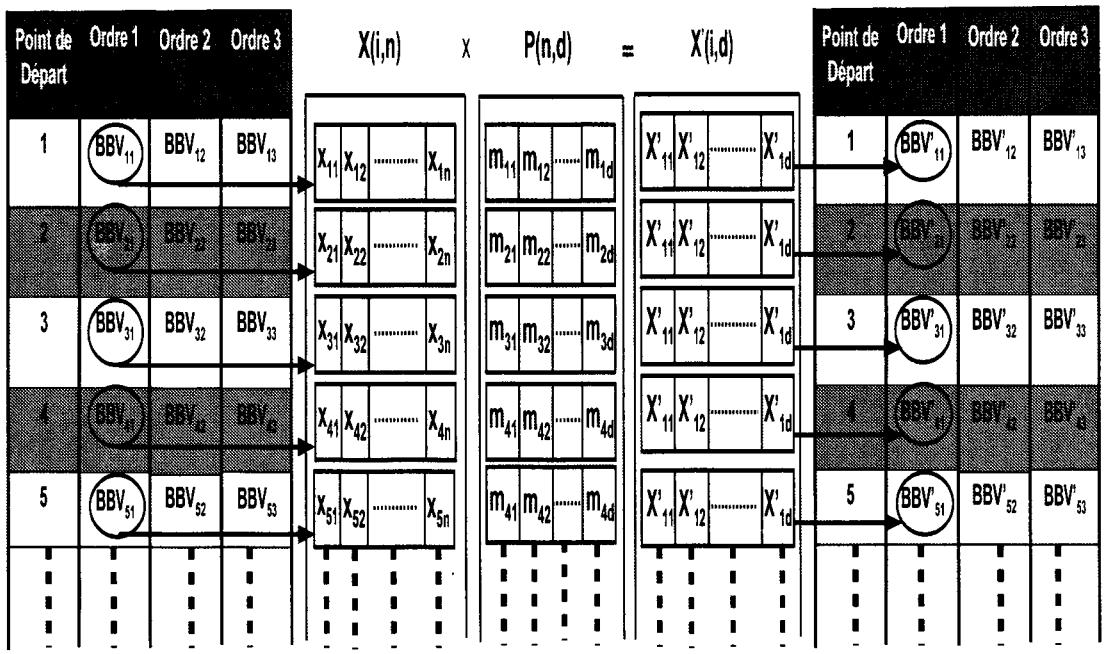


FIG. 6.5 – Projection aléatoire est appliquée à la matrice X contenant les BBVs d'ordre 1. Cette projection est faite en multipliant la matrice P ($-1 < m_{ij} < 1$) par X pour obtenir une nouvelle matrice X' de taille plus petite que X ($d < n$). X' contient les nouveaux BBVs réduits de l'ordre 1.

comparer la qualité de classification pour les différents nombres de Mphases. La classification choisie est celle ayant la plus petite valeur de k sachant que le score BIC doit être de 90% du meillieur score BIC obtenu.

Ces étapes sont expliquées en détail dans les sections suivantes.

6.3.2.1 Projection aléatoire pour les BBMs

Une projection des BBV est nécessaire afin de résoudre le problème de classification. En effet, la classification devient difficile avec l'augmentation de la dimension des BBVs. Cette dimension correspond au nombre de blocs de base exécutés dans l'application. Nous avons le même problème dans le cas des BBM, où la dimension est le nombre de colonnes (voir figure 6.5). Le fait que le temps d'exécution de l'algorithme de classification k-means dépend de la dimension. Ce temps devient important quand la dimension devient relativement large.

Dans ce travail nous utilisons la projection aléatoire de SimPoint pour réduire les dimensions de BBVs qui forment les BBMs. SimPoint garantit que cette projection est efficace en réduisant la dimension tout en gardant la conformité des données. Dans notre cas la réduction est faite jusqu'à une dimension 15 car dans Simpoint [57] cette dimension a été considérée comme suffisante pour différencier les phases de l'exécution. En partant d'une matrice X contenant les BBVs pour un ordre de granularté donnée de ixn, i et n étant le nombre d'intervalles et le nombre de blocs de base, nous obtenons une matrice de ixd, d étant la nouvelle dimension qui vaut 15 dans notre cas, en utilisant la projection linéaire aléatoire. La matrice

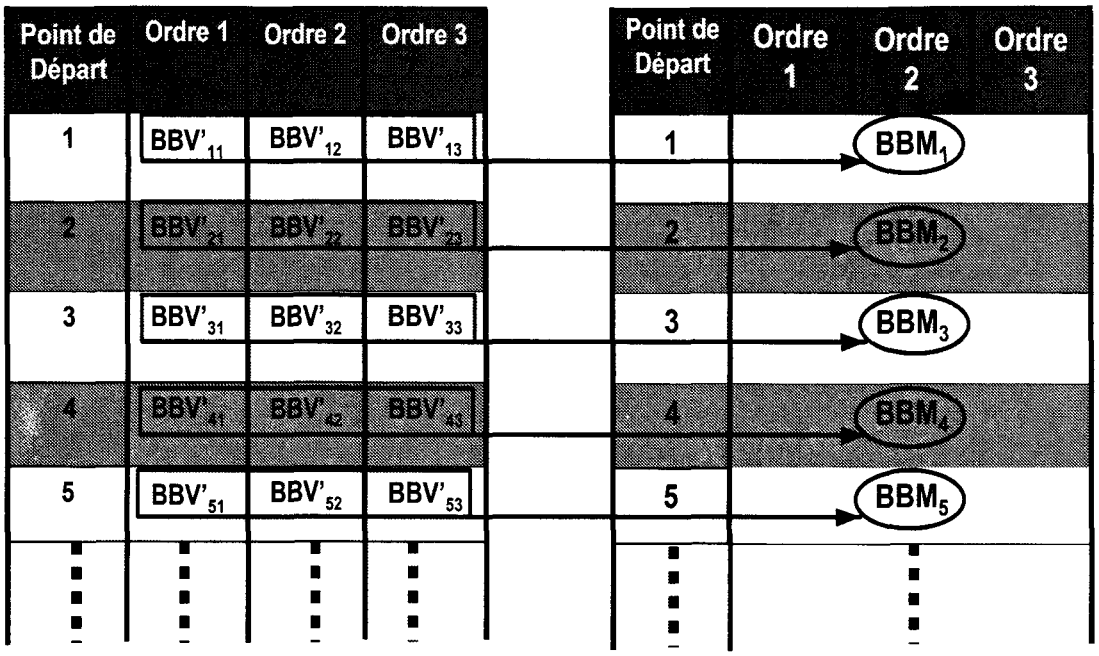


FIG. 6.6 – Les BBVs de dimension réduite de la figure 6.5 forment les BBMs qui vont subir la classification avec k-means.

X contient les BBVs pour un ordre de granularité donné. Les deux étapes suivantes sont réalisées pour chacune des g granularités :

1. Créer une matrice P de taille *nxd* en choisissant des valeurs aléatoires entre -1 et 1 pour chaque case de la matrice P. Les valeurs n et d sont respectivement le nombre de blocs de base et la nouvelle dimension choisie.
2. Multiplier la matrice X contenant les BBVs de la granularité par la matrice P afin d'obtenir une nouvelle matrice $X^{'}$ de taille *ixd* réduite.

La figure 6.5 montre l'application de la projection à la matrice X contenant les BBVs d'ordre 1 pour obtenir une matrice $X^{'}$ contenant les nouveaux BBVs de même ordre. De la même manière la matrice P est multipliée par les matrices contenant les BBVs des autres granularités pour obtenir des BBVs réduits. Ainsi tous les BBVs réduits vont ensuite former les BBMs qui vont subir la classification.

Les nouveaux BBVs forment les BBMs. La figure 6.6 montre la génération d'une BBM à partir des BBVs qui correspondent à la même ligne de la figure 6.5. Dans ce cas la taille de chaque BBM est réduite. Nous somme passés d'une dimension *gxn* à une dimension *gxd* pour les BBMs.

6.3.2.2 Algorithme K-means avec les BBMs

Le "k-means" est un algorithme d'optimisation itérative qui s'exécute en deux étapes. Ces deux étapes se répètent jusqu'à la convergence. Il commence par une assignation aléatoire des centres de k groupes ou Mphases dans notre cas puis effectue la procédure itérative. Les centres de Mphases définissent l'appartenance à la Mphase pour chaque BBM. De plus

suivant les BBMs qui composent la Mphase, le centre de cette dernière sera calculé. En effet le centre de chaque Mphase correspond à la moyenne des BBMs qui la composent. Chaque BBM appartient à une Mphase unique.

Pour les nouvelles BBMs (voir section 6.3.2.1), la distribution est réalisée par k-means pour chaque valeur de k, avec k allant de 1 jusqu'au Max_K. Les centres pour les k Mphases sont initialisés en choisissant aléatoirement k BBMs parmi les BBMs qui seront classifiées. Après l'initialisation, l'algorithme k-means est réalisé en deux étapes qui se répètent jusqu'à la convergence.

1. Pour chaque BBM, il faut comparer sa distance avec les k centres de Mphases en utilisant la formule de la distance euclidienne (6.1). La BBM en question est assignée à la Mphase ayant le centre le plus proche (distance la plus petite).
2. La position du centre de la Mphase sélectionnée à l'étape précédente est affectée au centroïde de tous les BBMs dans la Mphase. Le centroïde est calculé comme la moyenne de toutes les BBMs dans la Mphase.

Une fois que toutes les BBMs ont été distribuées, la deuxième itération est réalisée pour distribuer de nouveau les BBMs sur les phases ayant les centres obtenus dans l'itération précédente. Cette procédure est itérée jusqu' au moment où l'appartenance de BBMs cesse de changer entre les itérations. De même cette procédure d'itération est faite pour chaque valeur de k.

Pour choisir la meilleure valeur de k Simpoint affecte un score de BIC pour mesurer la qualité de classification de BBMs pour chaque valeur de k. Ainsi le score de BIC est utilisé pour comparer la qualité de classification des valeurs de k. On trouvera dans [57] plus de détails sur la façon avec laquelle les BIC sont calculés. Dans notre cas la valeur de k choisie est celle ayant la plus petite valeur sachant que le score du BIC est au moins de 90% du meilleur score obtenu. Ce seuil est utilisé par SimPoint [57]. Ainsi la classification est choisie et la Mphase de chaque BBM est obtenue.

6.3.2.3 Sélection des BBMs représentatives

Une fois que la classification des BBMs dans des Mphases a été réalisée, il faut trouver la BBM représentative de chaque Mphase. La BBM représentative représente le comportement de la Mphase tout entière. En effet, le centre de la Mphase ne correspond pas à une BBM qui représente un intervalle d'exécution mais correspond au centroïde de toutes les BBMs dans la Mphase. Le centroïde est calculé comme la moyenne de toutes les BBMs dans la Mphase. Dans ces condition, la BBM représentative est celle qui a la distance la plus proche du centre de Mphase. Rappelons que la distance euclidienne, définie dans la formule (6.1), est utilisée. Les BBMs représentatives des phases sont sélectionnées. Comme il y a autant de BBMs représentatives que de Mphases, k BBMs sont sélectionnées pour représenter le comportement de l'application tout entière. Ainsi les checkpoints sont collectés au point de démarrage des k BBMs représentatives.

La figure 6.7.B montre la classification des BBMs de l'application de la figure 6.7.A. Les BBMs similaires ont le même identificateur de Mphase. L'identificateur de Mphase correspond au nom de l'application et à l'indice de la BBM représentative de cette Mphase. A titre d'exemple la première et la quatrième BBM sont considérés similaires et ont le même identificateur de Mphase. La quatrième BBM nommée BBM_4 a été sélectionnée comme la BBM représentative de cette Mphase. Ainsi l'identificateur de cette Mphase est X_4 dans la

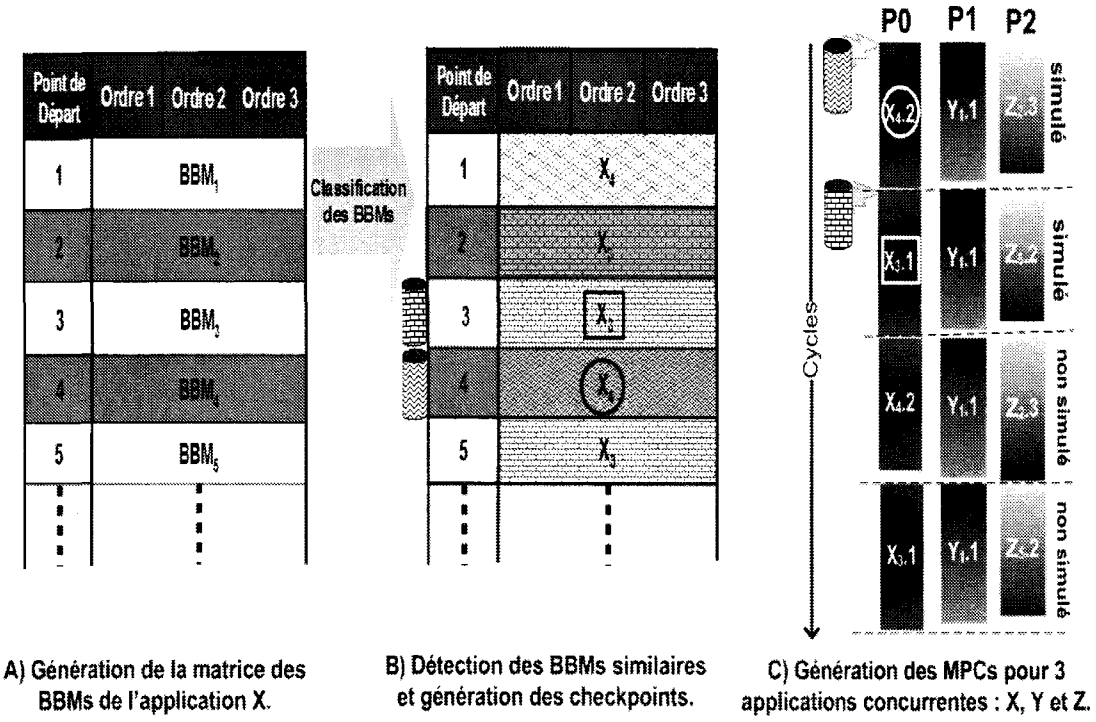


FIG. 6.7 – Les BBMs similaires dans la figure B ont le même identificateur. L'identificateur de Mphase est composé du nom de l'application et de l'indice de sa BBM représentative. Les cylindres représentent les checkpoints collectés. C) $X_i.j$ signifie que les instructions exécutées appartiennent à la Mphase X_i et à l'ordre de granularité j.

figure. La figure 6.7.B montre deux Mphases et leurs deux BBMs représentatives BBM_3 et BBM_4. Comme les checkpoints sont collectés aux points de démarrage des BBMs représentatives, les checkpoints dans la figure 6.7.B sont collectés au troisième et au quatrième point de démarrage. Ces checkpoints sont représentés par des cylindres distincts. L'intégration des BBMs avec la méthode MGS est expliquée dans la section suivante.

6.3.3 Utilisation des BBMs par MGS pour l'accélération de la simulation

Avec la méthode MGS, les BBVs des différents ordres de granularités forment des BBMs. Comme nous venons de le voir, à la fin de la classification des BBMs, les informations suivantes sont fournies au simulateur :

- Une trace unique de Mphases correspondant aux BBMs (voir figure 6.7.A) avec un même identificateur de Mphase pour les BBMs similaires.
- Le point du démarrage de chaque BBM représentative détectée. Le point de démarrage est le numéro d'instruction où commence la BBM représentative en question.

Les checkpoints sont collectés aux points de démarrage des BBMs représentatives. A titre d'exemple, dans la figure 6.7.B les checkpoints sont collectés aux points de démarrage 3 et 4. Il y a autant de checkpoints que de BBMs représentatives. Contrairement à la méthode de similarité des lignes de phases expliquée dans la section 6.2.2, ce nombre peut être fixé par l'utilisateur et ne dépend pas des ordres de granularité analysés avec MGS. L'utilisateur

est donc capable de fixer un nombre maximum de checkpoints à collecter de telle sorte que ces chechkpoints puissent par exemple être préchargés dans la mémoire RAM de la machine hôte.

Au niveau de chaque processeur, et au point de démarrage de la simulation du MPC, le checkpoint associé à la BBM représentative de BBM dont on va démarrer la simulation est chargé. Ainsi les deux états, architectural et micro-architectural sont restaurés au point correspondant. Les instructions de la BBM représentative sont alors exécutées représentant ainsi le comportement de la BBM en question.

Dans la figure 6.7.C, P0 simule premièrement la BBM qui représente la Mphase X_4. La BBM représentative de cette Mphase est la quatrième BBM dans la matrice (voir figure 6.7.B) et correspond au point de démarrage 4. Ainsi le checkpoint de cette BBM est chargée au niveau de P0 au début du premier MPC. Dans la figure le cylindre qui correspond à ce checkpoint est montré au début du premier MPC. Dans ce cas les instructions de la quatrième BBM sont exécutées et représentent ainsi le comportement de la première BBM. Comme la granularité des instructions exécutées dans un MPC est déterminée dynamiquement par MGS, nous trouvons dans le premier MPC que les instructions exécutées par P0 sont d'ordre 2. Dans la figure 6.7.C l'identificateur de Mphase est suivi par l'ordre de granularité des instructions.

Après l'exécution d'un intervalle d'ordre 2, le deuxième intervalle de P0 démarre au point 3. Ainsi pour le deuxième MPC, P0 charge le checkpoint de la BBM représentative démarrant au point 3. Dans cet exemple la BBM représentative de la troisième BBM est la même (notée X_3).

Pendant la simulation par MGS, un MPC est considéré similaire à un MPC déjà simulé si au niveau de chaque processeur la Mphase dans le MPC simulé et celle dans la trace sont similaires, ayant donc le même indentificateur (voir figure 6.7.B). Ainsi dans la figure 6.7.C, le troisième MPC est considéré similaire au premier MPC. De même le quatrième MPC est considéré similaire au deuxième. Dans chaque MPC sauté, le nombre d'instructions avancées par chaque processeur correspond au nombre d'instructions qu'il exécute dans le MPC simulé.

Ici nous pouvons noter que les avantages de la méthode des BBMs par rapport à la méthode de similarité des lignes sont les suivants :

1. Le nombre de checkpoints est indépendant de la granularité maximale analysée. En fonction de la précision désirée et de la capacité de mémoire RAM de la machine hôte, l'utilisateur fixe un nombre maximum de checkpoints.
2. La BBM représentative correspond au centre de Mphase des BBMs similaires. Cette BBM est plus représentative que la première occurrence rencontrée durant la simulation. Nous assurons ainsi que l'exécution des BBMs représentatives réduit l'erreur due à l'approximation des IPCs des MPC avec les IPCs des MPCs représentatifs.

Dans ce chapitre nous nous limitons à évaluer le gain du premier point, celui qui correspond à une réduction de l'espace de stockage. Les contraintes de temps n'ont pas permis de faire l'étude similaire pour le gain en précision (point 2). Concernant l'accélération de la simulation d'une application, celle ci dépend du nombre de Mphases détectées et de la valeur du TWSB. Comme nous le savons, ces deux nombre sont contrôlables par l'utilisateur. Ainsi dans les perspectives de cette thèse une étude a été proposée dans le but de paramétrer cette méthode afin de répondre d'une manière satisfaisante aux besoins de l'accélération et de précision.

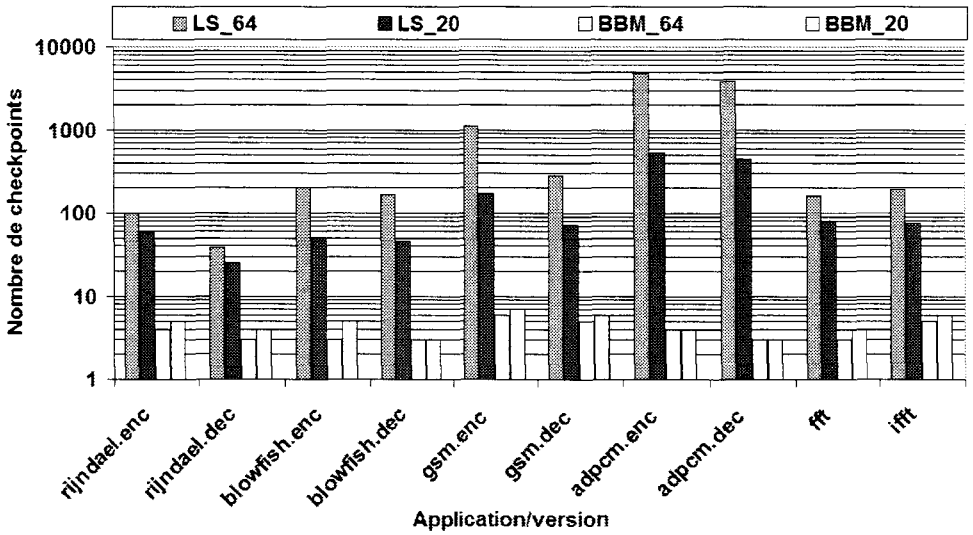


FIG. 6.8 – Variation du nombre de checkpoints pour la méthode de lignes similaires (LS) et pour la méthode BBM (BBM) pour 20 et 64 ordres de granularités.

6.4 Évaluation du gain en stockage du checkpointing avec les BBMs

Dans cette section nous évaluons le gain en stockage quand les checkpoints sont collectés en utilisant les BBM représentatives par rapport à la méthode de lignes similaires. Nous évaluons la performance de la méthode des BBMs pour 20 et 64 ordres de granularités avec des intervalles de 50K instructions pour la granularité d'ordre 1. Dans les expériences que nous avons réalisées nous avons fixé *Max_K* à 10. La quantité de stockage est calculée pour les deux version "decode" et "encode" de chaque application.

6.4.1 Évaluation du nombre de checkpoints

Avec MGS nous disposons de deux méthodes pour collecter des checkpoints représentatifs. Ces deux méthodes sont respectivement la méthode des lignes similaires (LS) et la méthode BBM (BBM) détaillées dans ce chapitre.

La figure 6.8 montre la variation du nombre de checkpoints pour les deux méthodes LS et BBM pour les ordres de granularités 20 et 64. Nous avons choisi ces deux ordres de granularité car dans nos expériences, le plus grand ordre de granularité qui a été trouvé avec un TWSB de 20% (resp 5%) est de 20 (resp 64). Comme il a été expliqué précédemment, la figure 6.8 montre que le nombre de checkpoints augmente avec les ordres de granularité dans le cas de LS. Par exemple dans le cas de l'application adpcm le nombre de checkpoints dépasse 3000 checkpoints pour un ordre de granularité 64. Il est évident que pour LS le nombre de checkpoints dépend des ordres de granularités analysées. Contrairement à LS, le nombre de checkpoints est indépendant des ordres de granularités analysées pour BBM. Ce nombre dépend de la valeur Max_K précisée à priori par le concepteur. Pour un Max_K de 10, la figure 6.8 montre le nombre de checkpoints pour 20 et 64 ordres de granularités.

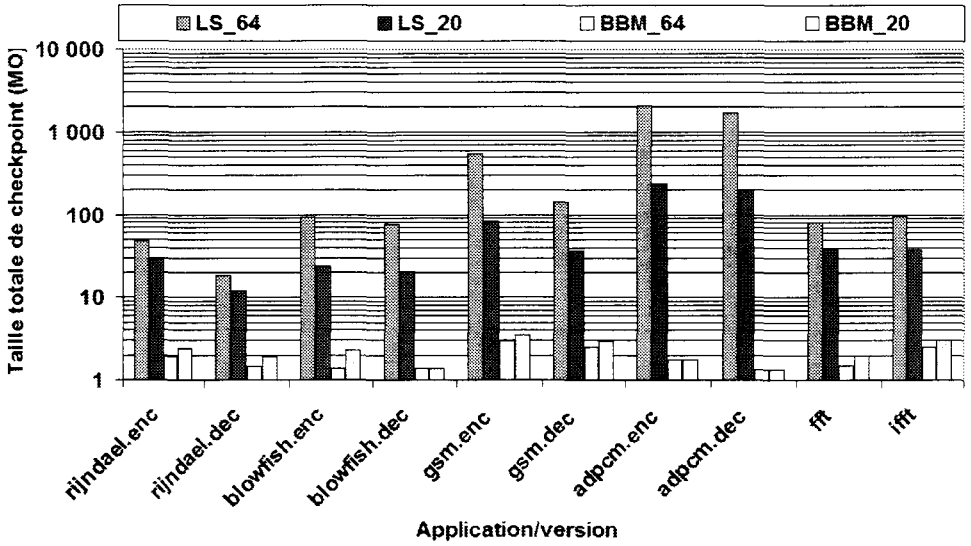


FIG. 6.9 – Variation de la taille totale en MO pour stocker les checkpoints avec LS et avec BBM pour des granularités 64 et 20.

Comme on peut le voir, ce nombre est inférieur à 10 quelque soit l'ordre de granularité analysée. De même nous constatons pour BBM que le nombre de checkpoints (nombre de Mphases) diminue généralement avec l'augmentation des ordres de granularité. En effet dans notre cas, les BBVs sont une sorte de combinaisons des BBVs de granularité d'ordre 1. Ainsi pour un même ordre de granularité les BBVs comprennent des BBs communs. Le nombre de ces BBs communs augmente avec la granularité. Ceci explique la diminuation du nombre de Mphases avec la granularité, d'où la diminuation du nombre de checkpoints dans la figure 6.8.

En effet, l'utilisation de BBM est très importante pour les systèmes embarqués où les ordres de granularité des applications concurrentes peuvent être importants à cause de l'augmentation du degré d'hétérogénéité entre ces applications. BBM permet le contrôle du nombre de checkpoints par l'utilisateur. Ce dernier choisit la valeur de Max_K en respectant par exemple la capacité mémoire de la machine hôte de simulation et la précision désirée.

6.4.2 Gain en taille mémoire des checkpoints

Rappelons que les deux techniques expliquées dans la section 6.2.1 sont utilisées pour construire les checkpoints de deux états architectural et micro-architectural. La taille d'un checkpoint correspond à la taille de données qui permettent de construire l'image de mémoire, des registres et des mémoire caches.

La figure 6.9 montre la taille en MO de la zone mémoire nécessaire pour sauvegarder tous les checkpoints collectés pour LS et pour BBM pour les ordres de granularités 20 et 64. La taille totale est au maximum de 3 MO dans le cas de BBM tandis qu'elle dépasse 1000 MO pour adpcm dans le cas de LS_64. A titre d'exemple, si le nombre de processeurs simulés est 12, alors nous avons besoin de 12 GO de RAM pour LS tandis que seul 16 MO de RAM suffisent pour BBM. Ainsi en utilisant BBM avec MGS, le préchargement des checkpoints dans

la mémoire centrale RAM est possible même pour une machine hôte de gamme moyenne. Ce préchargement réduit le temps nécessaire pour restaurer les checkpoints, ce qui provoque une réduction du temps de simulation pour la méthode MGS.

6.5 Conclusion

Dans ce chapitre nous nous sommes intéressés à étudier le checkpoiting avec la méthode MGS. L'analyse des matrices de phases générées nous a révélé la similarité des lignes des phases. Dans un premier temps, nous avons mis à profit de cette similarité pour collecter un seul checkpoint représentatif pour chaque groupe de lignes similaires au lieu de collecter un checkpoint par intervalle. Par conséquence, cette réduction, dans le nombre de checkpoints, permet de stocker ces checkpoints dans la mémoire RAM. Ce qui améliore le temps de chargememt des checkpoints durant la simulation améliorant ainsi le temps de simulation pour MGS. Néanmoins, nous avons remarqué que ce nombre de checkpoints dépend des granularités choisies par l'utilisateur.

Pour remédier à cet inconvénient, nous avons proposé une deuxième méthode BBM qui permet de réduire le nombre de checkpoints. La matrice de BBVs est donnée à un outil de classification. Dans notre cas nous avons modifié Simpoint pour réaliser cette tâche. Cet outil détecte la similarité des matrices des BBVs générées afin de les classer dans des phases. Un checkpoint est collecté pour chaque phase générée. Les résultats ont montré que le nombre de checkpoints par BBM est indépendant des granularités analysées. De plus BBM permet aux utilisateurs de choisir un nombre maximum de checkpoints à collecter en se basant sur la taille de la mémoire RAM de la machine hôte disponible. Ce qui permet le chargement des checkpoints dans la mémoire RAM. Une étude complémentaire est nécessaire pour évaluer la précision de la méthode BBM.

Chapitre 7

Conclusion et perspectives

7.1 Bilan

Les travaux qui ont été présentés dans ce mémoire s'inscrivent dans le cadre de la réduction de la phase de conception des systèmes sur puce multiprocesseurs (MPSoC). Nous avons d'abord exposé les difficultés de la conception des MPSoC liées principalement à l'augmentation de la complexité architecturale de ces systèmes. Notre contribution réside dans la mise au point de techniques visant à écourter la phase de simulation et d'exploration des alternatives architecturales. Il est important de souligner que dans le cadre de cette thèse nous avons prêté attention à la méthodologie par échantillonnage des applications afin d'accélérer la simulation des systèmes MPSoC. Cette méthodologie consiste à choisir des phases représentatives de toute l'application. Comme nous l'avons expliqué, l'échantillonnage a été utilisé largement dans le cas de la simulation des systèmes mono-processeurs. Les travaux dans le domaine de l'échantillonnage dans les architectures multi-processeurs sont rares. L'utilisation de l'échantillonnage sur ces dernières est problématique car il est difficile à déterminer à l'avance les phases exécutées simultanément par les différents processeurs.

Nos travaux ont commencé par la proposition d'une méthode d'accélération de la simulation par échantillonnage nommée *Adaptive Sampling* (AS). Cette méthode permet d'adapter dynamiquement les échantillons ainsi que le pourcentage d'échantillons prélevés de telle sorte que l'erreur sur l'estimation de la performance reste acceptable. Les échantillons générés par AS, nommés *Cluster of Strings* (CS), correspondent à des recouvrements de séquences de phases associées aux applications s'exécutant en parallèle. Une seule occurrence d'un groupe de CS subit la simulation détaillée.

Les expériences montrent que AS permet un échantillonnage adaptatif des applications en se basant sur leur comportement dynamique sur la platforme architecturale. Dans le cas des applications homogènes, l'accélération aboutit à un facteur de 800 tout en gardant une erreur inférieure à 10%. De même en comparant AS avec la méthode Cophase [21, 18] pour les applications homogènes, nous avons remarqués que AS est plus performante du point de vue de l'accélération et de la précision.

Cependant quand les applications parallèles sont hétérogènes (applications ayant différents comportements), les séquences de phases deviennent longues. La similarité entre deux CS contenant de longues séquences de phases se révèle difficile et peu probable. Ainsi il y a une faible probabilité de retrouver des CS qui se répètent et par conséquent un faible facteur d'accélération en résulte. Le facteur d'accélération obtenu pour ce type d'applications parallèles est proche de 1.5.

Pour résoudre ce problème, nous avons proposé deux techniques complémentaires à AS. La première technique, nommée *synthèse des CS*, améliore l'accélération de la simulation par la synthèse d'une séquence en plusieurs séquences semblables. L'accélération augmente alors jusqu'à un facteur 100 tout en gardant une erreur inférieure à 10%. Malheureusement, la synthèse de phases offre des performances limitées lorsque les séquences de phases sont longues. Dans ce cas, même en utilisant la synthèse de phases l'accélération ne dépasse pas la valeur 10. Pour remédier à cette nouvelle problématique, nous avons proposée la deuxième technique qui fixe la taille des intervalles des applications de façon intelligente. Ainsi au lieu d'avoir une taille identique d'intervalle pour toutes les applications, pour chaque application une taille est déterminée en vue d'obtenir des CS équilibrés en nombre de phases. Néanmoins, même si ces deux techniques améliorent le facteur d'accélération de la plupart de combinaisons d'applications, les performances obtenues dans certains cas ne sont pas excellentes.

Après ces deux techniques complémentaires à AS, nous avons présenté notre deuxième méthode, nommée *Multi-Granularity Sampling* (MGS), qui consiste à considérer dans la simulation des intervalles de tailles variables. Dans notre cas, les tailles des intervalles sont nommées granularités. Cette méthode permet une estimation plus précise de la performance tout en accélérant la simulation des applications hétérogènes concurrentes. Elle permet de regrouper les phases s'exécutant en parallèle dans un cluster de telle sorte que chaque cluster contienne une seule phase par application formant ainsi des *Multi-Phase Cluster* (MPC). La granularité de chaque phase est fixée selon le comportement dynamique de l'application correspondante. Là aussi seuls les nouveaux MPC qui n'ont pas été générés auparavant subissent la simulation détaillée. Cette réduction du nombre de phases par cluster simplifie la détection des phases qui se répètent, ce qui permet d'augmenter l'accélération de la simulation.

Nous avons comparé les performances de MGS à celles de AS et de Cophase. Dans le cas des applications homogènes nous avons remarqué que MGS accélère la simulation d'une façon plus importante que AS, alors que AS est plus précise. Dans le cas des applications hétérogènes, l'accélération obtenue par MGS est acceptable tandis que celle obtenue par *Cophase* est relativement faible en particulier quand le nombre de processeurs simulé est supérieur à 4. Par ailleurs notre méthode MGS est meilleure que Cophase. L'accélération par MGS atteint le facteur 60 alors que celle par Cophase n'est que de 3 pour les mêmes applications. De plus, l'erreur dans l'estimation des performances est en général acceptable. Elle est inférieure à 10% pour la plupart des applications étudiées. Une formule de correction de l'estimation de l'IPC est aussi proposée. La formule en question diminue l'erreur de 90%. L'avantage le plus important de MGS demeure le fait que l'accélération de la simulation s'adapte dynamiquement selon le comportement des applications sans intervention du concepteur du système. Ainsi grâce à la méthode MGS, nous avons obtenu une accélération de la simulation tout en gardant un bon niveau de précision pour les applications homogènes ou hétérogènes concurrentes.

Dans la dernière partie de cette thèse, nous avons étudié l'utilisation de la technique des points de contrôle (ou *checkpoints*). Cette technique permet de construire l'état du contexte du programme et l'état des composants architecturaux (les mémoires caches, etc.) au début de la simulation détaillée d'un intervalle. L'analyse des matrices de phases générées par MGS nous a révélé l'existence d'une similarité importante entre les lignes des phases. Nous avons pris en compte cette similarité pour construire les checkpoints représentatifs de groupes de lignes similaires au lieu d'un checkpoint pour chaque intervalle. Cette réduction, dans le nombre de checkpoints générés, permet de les stocker dans la mémoire RAM. Ceci améliore le temps de chargement des checkpoints durant la simulation améliorant ainsi le temps de simulation par MGS.

Nous avons remarqué aussi que le nombre de checkpoints représentatifs dépend des granularités analysées. Pour cela nous avons proposé une deuxième méthode basée sur la matrice des blocs de base. Cette méthode permet à l'utilisateur de choisir le nombre de checkpoints représentatifs selon la taille de la mémoire RAM disponible sur la machine hôte. Cette caractéristique permet aussi le préchargement des checkpoints dans la mémoire RAM, ce qui doit réduire le temps de simulation. La matrice BBM comprend les *Basic Block Vector* (BBV) représentant des intervalles de tailles différentes.

Nous avons aussi modifié SimPoint pour réaliser une classification sur les BBM. Un checkpoint est ensuite collecté pour chaque BBM représentative. Les résultats ont montré que le nombre de checkpoints par la méthode BBM est indépendant des granularités de

phases analysées par MGS.

7.2 Perspectives

Le travail effectué dans le cadre de cette thèse peut être poursuivit dans de nombreuses directions. Nous présentons ici celles qui nous semblent les plus intéressantes.

1 Choix dynamique du seuil du temps d'attente devant les barrières de simulation (ou TWSB)

Les expériences ont montré que l'effet du TWSB sur l'accélération et sur l'erreur d'estimation varie d'une application à l'autre et d'une configuration architecturale à l'autre. Rappelons que TWSB permet de contrôler l'équilibre entre l'accélération et la précision. Pour une même valeur de TWSB, l'erreur sur l'estimation des performances peut être plus ou moins importante. Par ailleurs nous avons remarqué que l'accélération de la simulation pour certaines applications n'augmente pas quand le TWSB dépasse une certaine limite tandis que pour d'autres applications l'accélération augmente continuellement avec l'augmentation du TWSB. Il est donc intéressant de prévoir un outil permettant de choisir dynamiquement le TWSB en se basant sur des critères associés aux applications. Ceci permet d'atteindre l'accélération maximum spécifique à chaque application tout en gardant un niveau acceptable de l'erreur.

2 Développement et évaluation des performances de la méthode BBM

Dans le chapitre 6, nous avons montré le gain du point de vue nombre de checkpoints et espace de stockage avec la méthode BBM. Il est intéressant d'implémenter complètement la méthode BBM afin d'évaluer les performances du point de vue accélération de la simulation et précision. En effet dans le cadre des BBM, les MPC représentatifs sont composés de BBM représentatives des phases, ce qui réduit l'erreur sur l'estimation.

3 Accélération de la simulation pour les applications communicantes

Dans cette thèse, des techniques pour accélérer la simulation des applications concurrentes indépendantes ou non communicantes ont été proposées. Il serait intéressant d'étudier l'accélération des applications parallèles communicantes avec les méthodes proposées AS et/ou MGS. En effet, ces dernières contiennent des primitives de synchronisation (lock, barrière de synchronisation, etc.). Ceci nécessite de concevoir une technique qui utilise d'une manière efficace les barrières de simulation adoptées par AS et MGS avec les primitives de synchronisation des applications de telles sortes à ne pas bloquer le système. Cette proposition nécessite d'étudier les conditions supplémentaires lors de la génération de la barrière de simulation. Il faut par exemple veiller à ce qu'un processeur se trouvant dans une section critique ne génère pas une barrière de simulation.

4 Conception d'une méthode hybride

Comme il a été noté dans le chapitre sur l'état de l'art, l'émulation sur FPGA est plus rapide et plus précise que la simulation réalisée par un ISS au niveau CABA. L'émulation

sur FPGA demande en contre partie des connaissances détaillées sur la structure de chaque composant du système, ainsi qu'un temps de développement relativement coûteux. Ainsi le développement sur un FPGA exige un effort important comparativement à l'utilisation d'un simulateur. A l'opposé la simulation au niveau CABA est plus lente.

Il est donc intéressant de concevoir une méthode d'accélération hybride utilisant les circuits reconfigurables (FPGA) et un simulateur conjuguant ainsi les avantages des deux plateformes. Nous pourrons utiliser ainsi nos méthodes d'échantillonnage AS, MGS pour accélérer la simulation et exploiter les barrières de simulation pour synchroniser le simulateur avec le FPGA au démarrage de l'émulation de chaque intervalle et vice versa. Généralement, seul un sous ensemble des composants du système contribue largement dans l'exécution de l'application sur le simulateur. Une bonne partie des composants du système (les outils d'entrée/sorties, les RAM, etc.) sont rarement activés. Ainsi, cette proposition nécessite une analyse au niveau des phases des applications dans le but de réduire la complexité du développement en synthétisant sur FPGA uniquement les opérations fréquentes et simples tandis que les composants complexes et rares seront représentés sur le simulateur.

Bibliographie personnelle

Conférence internationale avec actes et comité de lecture:

[1] Melhem Tawk, Kaled Z. Ibrahim, and Smail Niar. Adaptive sampling for efficient mpsoc architecture simulation. In *MASCOTS '07: Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 186–192,Washington, DC, USA, 2007. IEEE Computer Society.

[6] Melhem Tawk, Khaled Z. Ibrahim, and Smail Niar. Multi-granularity sampling for simulating concurrent heterogeneous applications. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 217–226, New York, NY, USA, 2008. ACM.

Revue internationale avec comité de lecture:

[2] Melhem Tawk, Kaled Z. Ibrahim, and Smail Niar. Parallel application sampling in mpsoc simuation. ***Submited*** *to Design Automation for Embedded Systems journal*, Feb 2009.

Séminaire international avec actes sans comité de lecture:

[3] Melhem Tawk, Khaled Z. Ibrahim, and Smail Niar. Simulation acceleration for mpsoc performance and power consumption evaluation. In *Architectures and Compilers for Embedded Systems (ACES)*, Anvers, Belgium, Oct 2006.

[4] Melhem Tawk, Khaled Z. Ibrahim, and Smail Niar. Dynamic sample generation for rapid mpsoc simulation. In *Colloque du GDR SOC SIP*, Paris, France, Jun 2007.

[5] Melhem Tawk, Khaled Z. Ibrahim, and Smail Niar. Small sample clustering for mpsoc simulation. In *Advanced Computer Architecture and Compilation for Embedded Systems (ACACES)*, Aquila, Italy, Jul 2007.

Bibliographie

[7] Advanced encryption standard. *http ://www.nist.gov/aes.*

[8] Mendation g.726 (12/90) - 40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (adpcm). *http ://www.itu.int*, Dec. 1990.

[9] Counterpane internet security, inc. the blowfish encryption algorithm. *http ://www.counterpane.com/blowfish.html*, 1993.

[10] *International Telecommunication Union. Global Standard for Mobile Communication, http ://www.itu.int*, Feb 2000.

[11] *ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document JVTE022*, Sep. 2002.

[12] I. S. 1076-1987. *IEEE Standard VHDL Language Reference Manual.* IEEE Press, Dec. 1998.

[13] S. 3.1. Accellera's extensions to verilog. 2003.

[14] G. Ascia, V. Catania, A. G. D. Nuovo, M. Palesi, and D. Patti. Efficient Design Space Exploration for Application Specific Systems-on-a-Chip. *Journal of Systems Architecture*, 53, Jan. 2007.

[15] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. Unisim : An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, 2007.

[16] A. P. Batson and A. W. Madison. Measurements of major locality phases in symbolic reference strings. In *Proceedings of the ACM SIGMETRICS conference on Computer performance modeling measurement and evaluation*, pages 75–84, New York, NY, USA, 1976. ACM.

[17] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM : Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing*, 41(2) :169–182, 2005.

[18] M. V. Biesbrouck. Sampled simulation for multithreaded processors. *Thèse à l'université de California San Diego USA*, 2007.

[19] M. V. Biesbrouck, B. Calder, and L. Eeckhout. Efficient Sampling Startup for SimPoint. *IEEE Micro*, 26(4) :32–42, 2006.

[20] M. V. Biesbroucky, L. Eeckhout, and B. Calder. Considering All Starting Points for Simultaneous Multithreading Simulation. *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 143– 153, Mar. 2006.

[21] M. V. Biesbroucky, T. Sherwoodz, and B. Caldery. A co-phase matrix to guide simultaneous multithreading simulation. *The IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, Mar. 2004.

[22] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3) :13–25, 1997.

[23] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. Fpga-accelerated simulation technologies (fast) : Fast, full-system, cycle-accurate simulators. In *MICRO '07 : Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[24] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. Protoflex : Fpga-accelerated hybrid functional simulation. Technical report, Parallel and Distributed Processing Symposium, IEEE International, 2007.

[25] M. Dales. Swarm-software arm. *http ://www.cl.cam.ac.uk/mwd24/phd/swarm.html.*

[26] A. Donlin. Transaction level modeling : flows and use models. *CODES+ISSS '04 : Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004.

[27] L. Eeckhout, Y. Luo, K. D. Bosschere, and L. K. John. Blrl : Accurate and efficient warmup for sampled processor simulation. *Computer Journal*, 48(4) :451–459, 2005.

[28] Future of Embedded Systems Technology from BCC Research Group. `http://www.bccresearch.com/report/IFT016B.html`.

[29] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip. *The IEEE/ACM international conference on Computer-aided design (ICCAD)*, Nov. 2001.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench : A free, commercially representative embedded benchmark suite. *The 4th IEEE Annual Workshop on Workload Characterization (WWC)*, pages 3–14, Dec. 2001.

[31] G. Hamerly, E. Perelman, and B. Calder. How to use simpoint to pick simulation points. *SIGMETRICS Perform. Eval. Rev.*, Mar. 2004.

[32] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0 : Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[33] J. W. Haskins and J. K. Skadron. Memory reference reuse latency : Accelerated sampled microarchitecture simulation. In *In Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, 2003.

[34] HP. *http ://www.hpl.hp.com/research/linux/perfmon.*

[35] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. *IEEE Trans. Comput.*, 45(10) :1180–1194, 1996.

[36] J. John W. Haskins and K. Skadron. Accelerated warmup for sampled microarchitecture simulation. *ACM Trans. Archit. Code Optim.*, Mar. 2005.

[37] J. Johnston and G. Hamerly. Improving simpoint accuracy for small simulation budgets with edcm clustering. *In proceedings of the Second workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART '08)*, Jun. 2008.

[38] T. S. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors : an analytical approach. In *ISCA '07 : Proceedings of the 34th annual international symposium on Computer architecture*, pages 402–411. Association for Computing Machinery, Jun. 2007.

[39] R. E. Kass and L. Wasserman. A reference bayesian test for nested hypotheses and its relationship to the schwarz criterion. *Journal of the American Statistical Association*, 90 :928–934, 1995.

[40] J. L. Kihm and D. Connors. CoGS-Sim - CoPhase-Guided Small-Sample Simulation of Multithreaded and Multicore Architecutres. *The 3rd annual Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun. 2007.

[41] J. L. Kihm and D. A. Connors. Statistical simulation of multithreaded architectures. In *MASCOTS '05 : Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 67–74, Washington, DC, USA, 2005. IEEE Computer Society.

[42] J. L. Kihm, S. D. Strom, and D. A. Connors. Pgss-Sim : Phase-guided, small-sample simulation. *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2007.

[43] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. 2001.

[44] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *CGO '06 : Proceedings of the International Symposium on Code Generation and Optimization*, pages 135–146. IEEE Computer Society, 2006.

[45] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE International Symmposium on*, Mar. 2005.

[46] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *In Hawaii International Conference on System Sciences*, pages 205–210, Jan. 1994.

[47] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, January 1967.

[48] R. Mouhoub and O. Hammami. Multiprocessor on chip : beating the simulation wall through multiobjective design space exploration with direct execution. *Parallel and Distributed Processing Symposium, International*, 0 :366, 2006.

[49] J. Namkung, D. Kim, R. Gupta, I. K. J. Bouget, and C. Dulong. Phase Guided Sampling for Efficient Parallel Application Simulation. *The 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, Oct. 2006.

[50] S. Niar, N. Inglart, M. Chaker, S. Hanafi, and N. Benameur. FACSE : a Framework for Architecture and Compilation Space Exploration. *The IEEE International Conference on Design and Technology of Integrated Systems in nanoscale era (DTIS)*, Sep. 2007.

[51] S. Nussbaum and J. E. Smith. Statistical Simulation of Symmetric Multiprocessor Systems. *The 35th annual Simulation Symposium (Ss)*, Apr. 2002.

[52] Open SystemC Initiative. *SystemC v2.1 Language Reference Manual*, 2003. `http ://www.systemc.org/`.

[53] D. Patterson and J. Hennessy. *Computer Organization and Design : The Hardware/Software Interface, 3rd Edition*. Morgan Kaufmann, 2006.

[54] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT '03 : Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2003.

[55] S. project. An open platform for modelling and simulation of multi-processors system on chip. `http ://soclib.lip6.fr/Home.html`.

[56] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on smt processors. *Parallel Architectures and Compilation Techniques, International Conference on*, 2003.

[57] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *The 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[58] S. Sutherland. *A Guide to the New Features of the VERILOG Hardware Description Language*. Springer, Jan. 2002.

[59] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz. Simsnap : Fast-forwarding via native execution and application-level checkpointing. *Interaction between Compilers and Computer Architecture, Annual Workshop on*, Feb. 2004.

[60] P. G. D. Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini, and G. D. Micheli. Architectural exploration of mpsoc designs based on an fpga emulation framework. *The Conference on Design of Circuits and Integrated Systems (DCIS)*, Dec. 2006.

[61] E. Viaud, F. Pecheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on parallel discrete event principles. *Design, Automation and Test in Europe Conference and Exhibition*, 2006.

[62] T. Wenisch, R. Wunderlich, B. Falsafi, and J. Hoe. Simulation Sampling with Live-Points. *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, Mar. 2006.

[63] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Turbosmarts : Accurate microarchitecture simulation sampling in minutes. *In Proceedings of the ACM SIGMETRICS*, Jun. 2005.

[64] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts : Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *The 30th International Symposium on Computer Architecture (ISCA)*, Jun. 2003.

[65] J. J. Yi and D. J. Lilja. Simulation of computer architectures : simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on computers*, Mar. 2006.

Accélération de la Simulation par Échantillonnage dans les Architectures Multiprocesseurs Embarquées

Résumé : La conception de systèmes embarqués s'appuie fortement sur la simulation pour évaluer et valider des nouvelles configurations architecturales avant la réalisation physique. Néanmoins, comme ces systèmes deviennent de plus en plus complexes, la simulation de ces systèmes exige des temps importants. Ce problème est encore plus visible au niveau des architectures embarquées multiprocesseurs (ou MPSoC) qui offrent des performances certes intéressantes (en nombre d'instructions/Joule) mais qui exigent des simulateurs performants. Pour ces systèmes, il est impératif d'accélérer la simulation afin de réduire les délais de la phase d'évaluation des performances et obtenir ainsi des temps d'arrivée sur le marché (time-to-market) relativement courts. La thèse s'intéresse aux méthodes d'accélération de la simulation pour ce type d'architectures. Dans ce cadre, nous avons proposé une série de solutions visant à accélérer la simulation des MPSoC. L'ensemble des méthodes proposées sont basées sur l'échantillonnage des applications. Ainsi, les applications parallèles sont d'abords analysées afin de détecter les différentes phases qui les composent. Par la suite et pendant la simulation, les phases s'exécutant en parallèle se combinent et forment des clusters de phases. Nous avons mis au point des techniques qui permettent de former les clusters, de les détecter et de sauvegarder leurs statistiques de façon intéressante. Chaque cluster représente un échantillon d'intervalles d'exécution de l'application similaires. La détection de ces derniers nous évite de simuler plusieurs fois le même échantillon. Pour réduire le nombre de clusters dans les applications et augmenter le nombre d'occurrences des clusters simulés, une optimisation de la méthode a été proposée afin d'adapter dynamiquement la taille des phases des applications à simuler. Ceci permet de détecter facilement les scenarios des clusters exécutés lorsqu'une répétition dans le comportement des applications a lieu. Enfin, pour rendre notre méthodologie viable dans un environnement de conception de MPSoC, nous avons proposé des techniques performantes pour la construction de l'état exact du système au démarrage (checkpoint) de la simulation des clusters.

Mots clés : Système embarqué (MPSoC), accélération de la simulation, CABA, échantillonnage.

Simulation Acceleration by Sampling for Embedded Multiprocessor Architectures

Abstract : Embedded system design relies heavily on simulation to evaluate and validate new platforms before implementation. Nevertheless, as technological advances allow the realization of more complex circuits, simulation time of these systems is considerably increasing. This problem arises mostly in the case of embedded multiprocessor architectures (MPSoC) which offer high performances (in terms of instructions/Joule) but which require powerful simulators. For such systems, simultion should be accelerated in order to speed up their design flow thus reducing the time-to-market. In this thesis, we proposed a series of solutions aiming at accelerating the simulation of MPSoC. The proposed methods are based on application sampling. Thus, the parallel applications are first analyzed in order to detect the different phases which compose them. Thereafter and during the simulation, the phases executed in parallel are combined together in order to generate clusters of phases. We developed techniques that facilitate generating clusters, detecting repeated ones and recording their statistics in an efficient way. Each cluster represents a sample of similar execution intervals of the application. The detection of these similar intervals saves us simulating several times the same sample. To reduce the number of clusters in the applications and to increase the occurrence number of simulated clusters, an optimization of the method was proposed to dynamically adapt phase size of the applications. This makes it possible to easily detect the scenarios of the executed clusters when a repetition in the behavior of the applications takes place. Finally, to make our methodology viable in an MPSoC design environment, we proposed efficient techniques to construct the real system state at the simulation starting point (checkpoint) of the cluster.

Keywords : Embedded multiprocessor system (MPSoC), simulation acceleration, CABA, sampling.