NOTE TO USERS

This reproduction is the best copy available.

UNIVERSITÉ DE MONTRÉAL

CONCEPTION ET RÉALISATION D'UN OUTIL D'EXPLORATION ARCHITECTURALE DE LA HIÉRARCHIE DE MÉMOIRE D'UN SYSTÈME SUR PUCE AFIN D'OPTIMISER LA PERFORMANCE DE LA PLATEFORME LOGICIELLE

SÉBASTIEN FONTAINE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2008

Your file Votre référence
ISBN: 978-0-494-57246-7
Our file Notre référence
ISBN: 978-0-494-57246-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION ET RÉALISATION D'UN OUTIL D'EXPLORATION ARCHITECTURALE DE LA HIÉRARCHIE DE MÉMOIRE D'UN SYSTÈME SUR PUCE AFIN D'OPTIMISER LA PERFORMANCE DE LA PLATEFORME LOGICIELLE

présenté par : FONTAINE Sébastien

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. LANGLOIS J.M. Pierre, Ph.D., président

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. DAVID Jean-Pierre, Ph.D., membre

DÉDICACE

640 KB ought to be enough for everybody – citation attribuée à Bill Gates

REMERCIEMENTS

Je tiens d'abord à remercier M. Bois, mon directeur de maîtrise pour ses idées, ses conseils et sa disponibilité qui m'ont permis de mener à bien mes travaux de recherche.

Je tiens également à remercier les membres du groupe de recherche CIRCUS de l'École Polytechnique : Benoît Pilote, Laurent Moss, Luc Filion, Marc-André Cantin, Maxime De Nanclas et Sylvain Goyette pour leur précieuse collaboration et leur soutien.

Je tiens plus particulièrement à remercier Carole Vézina, ma mère, pour m'avoir aidé à corriger le texte et Louis-Mathieu Houle pour ses commentaires constructifs.

En dernier lieu, je tiens à remercier mes parents et amis qui m'ont soutenu moralement durant mes 6 années et demie à l'École Polytechnique.

RÉSUMÉ

Avec la croissance continue du nombre de transistors par puce grâce aux récents progrès en lithographie, il devient maintenant possible d'intégrer un grand nombre de composants complexes sur un même circuit intégré. Afin de tirer avantage de ce nombre de transistors, les concepteurs se tournent vers les systèmes multiprocesseurs sur puce (MPSoC) où plusieurs processeurs coexistent sur la même puce et travaillent de pair afin de produire un système performant et efficace. Ce nombre croissant de processeurs sur une même puce pose alors de nouveaux défis aux concepteurs pour l'utilisation de la mémoire intégrée à ces MPSoC.

Un de ces défis est l'utilisation efficace de la mémoire intégrée disponible sur la puce afin de maximiser la performance des logiciels s'exécutants sur les processeurs. Dans le cas de petits programmes, il est possible de stocker ceux-ci totalement sur cette mémoire intégrée (mémoire locale au processeur) afin de maximiser leur performance. Par contre, avec la complexification des applications et la monté en popularité des systèmes multiprocesseurs, la taille limitée de la mémoire intégrée devient un problème. Il est alors nécessaire de stocker le programme dans une mémoire plus volumineuse mais aussi plus lente et externe à la puce. Deux solutions se présentent alors : le partitionnement de mémoire qui vise à morceler le programme et à le répartir entre la mémoire intégrée et la mémoire externe et l'utilisation de la mémoire intégrée en tant que mémoire cache qui vise alors à accélérer la latence moyenne d'accès à la mémoire externe.

Plusieurs travaux se sont penchés sur l'amélioration de la performance de tels systèmes en tentant d'optimiser la hiérarchie de la mémoire. Certains travaux se concentrent uniquement sur l'utilisation de la mémoire locale, d'autres sur la configuration optimale de la mémoire cache. Par contre, peu de travaux se sont penchés sur l'utilisation et l'optimisation commune de la mémoire locale et de la mémoire cache.

Un outil, MemoryOptimizer, a donc été conçu dans l'optique d'effectuer l'exploration architecturale de la hiérarchie mémoire d'un système sur puce en utilisant une combinaison de mémoire locale et de mémoire cache. De plus, afin d'assurer des résultats précis en simulation, un modèle du MicroBlaze a été développé pour l'environnement de conception de systèmes sur puce SPACE. Finalement, un outil de recueil de la trace d'exécution, permettant à MemoryOptimizer d'amasser les métriques nécessaires à l'analyse du comportement du programme à optimiser, a été conçu et intégré au modèle de simulation.

À l'aide de MemoryOptimizer, il est maintenant possible de déterminer automatiquement le contenu et la taille de la mémoire locale ainsi que de déterminer les tailles optimales des mémoires cache. Il en résulte un système où la minorité du programme, demandant le plus grand nombre d'accès à la mémoire est contenu en mémoire locale et le reste, stocké en mémoire externe, est accéléré par l'utilisation des mémoires cache. De plus, l'outil permet de suggérer une taille minimale pour la pile et le tas du programme, ce qui permet de réduire davantage l'utilisation de la mémoire.

Les expériences menées démontrent que MemoryOptimizer permet d'améliorer grandement les performances d'un système basé sur la mémoire externe en plus de diminuer l'utilisation de la mémoire intégrée. Il est ainsi possible de réduire par un facteur de 6 la taille de la mémoire intégrée tout en maintenant la performance d'un programme situé uniquement en mémoire locale. L'intégration d'un grand nombre de processeurs sur une même puce est donc aisément réalisable sans qu'il soit nécessaire de faire des compromis en termes de performance. De plus, la méthodologie présentée n'est pas limitée seulement au MicroBlaze et à SPACE mais peut être adaptée à d'autres processeurs ainsi qu'à d'autres plateformes de simulation.

ABSTRACT

As the quantity of transistors per single chip steadily increases due to recent advances in lithography, it becomes possible to integrate a number of complex components on an integrated circuit. In order to maximize the use of these transistors, designers now concentrate on multi-processor systems on chip (MPSoC) on which many processors coexist and work together to produce a fast and efficient system. This increasing number of processors on a single chip thus poses new challenges in term of integrated memory utilization.

In order to perform optimally, a program executed by a processor is generally contained in a small memory integrated on the chip and directly connected to this processor. This type of memory is called local memory. Because this memory is limited in a given chip, the presence of multiple processors, every one of them striving to provide the best performance, severely limits the amount of local memory available to each processor. Using the integrated memory to store all these programs becomes impossible and mandates the use of an external memory which is much larger but also much slower than the integrated memory.

Previous works dealing with the performance enhancement of such systems by optimizing the memory hierarchy are numerous. Some works focus on effectively using the local memory while others focus on finding the optimal cache configuration. On the other hand, few works focus on the use and optimization of both the local memory and cache.

A tool, MemoryOptimizer, was thus designed to resolve this gap, by presenting a method of design space exploration of the memory hierarchy of a SoC, by using a combination of local memory and cache memory. In order to provide accurate simulation results, a model of the MicroBlaze processor has been implemented in the

SPACE simulation platform. Lastly, an execution-tracing tool has been designed and integrated into the MicroBlaze model to provide the metrics necessary for MemoryOptimizer to analyze the behavior of the program executed on the processor.

With MemoryOptimizer, it is now possible to automatically establish the content and the minimal size of the local memory but also the optimal configurations of the caches. The result of this optimization is a system where a small but memory intensive part of the program is relocated in local memory and where the remaining of the program is stored in external memory and access to this memory is accelerated by cache memories. Moreover, the tool can evaluate the minimal size of the stack and the heap to further reduce the memory footprint.

MemoryOptimizer can greatly enhance the performance of a system based on an external memory while also reducing the size of the required integrated memory. Thus, it is possible to reduce the size of the integrated memory by a factor of 6 while maintaining the same performance level than the same program completely stored in local memory. More processors can now be integrated in a single chip without compromising the execution speed of the programs. Furthermore, the proposed methodology is not limited only to the MicroBlaze and SPACE but can also be adapted to other processors and simulation platforms.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ANNEXES

LISTE DES ACRONYMES

ALU	Arithmetic-Logic Unit
API	Application Programming Interface
APU	Auxiliary Processing Unit
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
CA	Cycle Accurate
CWF	Critical Word First
BRAM	Block RAM
DDR	Dual Data Rate
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
ELF	Executable and Linkable Format
EMC	External Memory Controller
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSL	Fast Serial Link
GDB	GNU Debugger
GUI	Graphical User Interface
HDL	Hardware Description Language
IDCT	Inverse Discrete Cosine Transform
IF	Interface
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator

LMB	Local Memory Bus
LUT	Look Up Table
MCH	Multi Channel
MIPS	Million of Instructions Per Second
MMU	Memory Management Unit
MPSoC	Multi-Processor System-on-Chip
MSR	Machine State Register
OCM	On-Chip Memory
OPB	On-chip Peripheral Bus
PC	Program Counter
PIC	Programmable Interrupt Controller
PLB	Processor Local Bus
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTOS	Real Time Operating System
SDRAM	Synchronous DRAM
SPACE	SystemC Partitioning Aspects of Codesign and Exploration
SRAM	Static RAM
SoC	System-on-chip
SWT	Standard Widget Toolkit (Eclipse)
TLM	Transaction-Level Modeling
TTM	Time-To-Market
TF	Timed Functional
UART	Universal Asynchronous Receiver/Transmitter
UTF	Un-Timed Functional
VHDL	Very high speed integrated circuit Hardware Description Language
XCL	Xilinx Cache Link

CHAPITRE 1

INTRODUCTION

Cette section introduit le concept de systèmes sur puce (SoC), concept central à ce mémoire. Elle traite des aspects de conception, méthodes de design, des hiérarchies de mémoire utilisées ainsi que des problèmes reliés à ces hiérarchies.

1.1. Les systèmes embarqués d'aujourd'hui

Au cours des dernières années, les systèmes embarqués sont devenus omniprésents. Ils sont utilisés dans une multitude de domaines d'application : télécommunication, divertissement, transport, équipement médical. Certains instruments de musique utilisent même des systèmes embarqués afin d'étendre leurs fonctionnalités [1].

Certains produits n'utilisent qu'un seul processeur et d'autres en utilisent plusieurs ce qui permet d'améliorer les performances ou encore la fiabilité du système. Un exemple extrême est l'automobile de la série 7 de BMW qui contient plus de 100 processeurs [2]. Vu la quantité de voitures et de cellulaires vendus chaque année, il n'est donc pas étonnant d'apprendre que le marché des processeurs embarqués représentait 98% des ventes de processeurs [3] en 2005.

Bien qu'il n'ait été question jusqu'ici strictement que de processeurs embarqués, un système embarqué ne se résume pas seulement à un processeur. Un système embarqué se définit également par les accélérateurs matériels, la mémoire et les entrées-sorties. Bref, un système embarqué est bien souvent similaire à un ordinateur de bureau avec une différence majeure : il n'est en général conçu que pour exécuter un nombre limité de tâches [4], il est souvent conçu de la manière la plus performante et la moins coûteuse possible et avec un budget de puissance limité. Par contre, avec la miniaturisation des

transistors, il est maintenant possible de concevoir des systèmes embarqués très complexes sur une seule puce (SoC) et capables d'effectuer une variété grandissante de tâches comme les téléphones cellulaires dits « intelligents ».

1.2. Problématique

La complexification continue des SoCs et des systèmes multiprocesseurs sur puce (MPSoC) est en partie possible grâce à la célèbre loi de Moore. Cette loi, formulée originellement en 1965 par Gordon Moore [5] et révisée en 1975 [6], stipule que le nombre de transistors sur les puces double chaque 24 mois. Cette complexification des systèmes engendre un écart de productivité entre les ressources disponibles sur les puces (tant sur les ASIC que les FPGA) et la difficulté de concevoir du matériel utilisant ces ressources. Ce problème de productivité peut être atténué par l'utilisation des processeurs embarqués dans les SoC. L'utilisation de ces processeurs permet de diminuer le temps de développement du circuit [7] puisque l'on utilise des blocs déjà conçus et permet de diminuer le temps de conception du système puisque le comportement du système peut maintenant être codé en langage de plus haut niveau (en C par exemple) par opposition au VHDL. Afin de simplifier davantage la conception des programmes, un système d'exploitation peut être utilisé afin de fournir des mécanismes de synchronisation, d'exécution de tâches multiple et d'abstraction de matériel.

L'environnement d'exploration architecturale et de conception SPACE [44], basé sur la bibliothèque SystemC, vise à simplifier davantage ce processus de développement en proposant entre autres un mécanisme d'abstraction des détails d'implémentation des SoC. À l'aide de SPACE, il est possible d'effectuer des partitionnements logiciel-matériel et de créer facilement des systèmes multiprocesseurs. Les systèmes peuvent ensuite être simulés et optimisés à l'aide d'une suite d'outils de monitoring. Ceci permet donc de réduire significativement les temps de conception et de validation.

En général, le concepteur tente de stocker le programme à exécuter dans une mémoire intégrée au SoC très rapide. Puisque la technologie des mémoires ne progresse pas au même rythme que la technologie des processeurs, la taille de ces mémoires est limitée. Dans le cas des FPGA de Xilinx, la mémoire intégrée BRAM est d'une taille de quelques centaines de kilo-octets. Bien que cette mémoire soit de petite taille, elle est très rapide et peut exécuter jusqu'à 2 transferts par cycle ce qui permet d'exécuter le programme à pleine vitesse [14] lorsqu'elle est connectée directement à un processeur (mémoire locale au processeur). Un problème survient lorsque la taille des programmes dépasse la taille de la mémoire intégrée. Le concepteur est alors forcé d'utiliser une mémoire externe, plus volumineuse, mais aussi beaucoup plus lente. Le temps d'accès à une donnée passe donc de 1 cycle à plus de 10 cycles. De plus, dans un système multi processeur, les processeurs doivent alors se faire compétition pour obtenir l'accès à la mémoire, ce qui augmente encore une fois le temps d'accès moyen. Ce problème ne se pose généralement pas avec des mémoires locales puisque l'on peut dédier une mémoire à chaque processeur.

Il existe quelques méthodes afin d'augmenter la performance du système dans ces situations. Ici, deux méthodes seront discutées. Premièrement, il est possible d'utiliser des mémoires cache [13] qui permettent alors de stocker dynamiquement les instructions et les données récemment utilisées dans une mémoire rapide. Cette approche est simple et permet d'obtenir de très bonnes performances. Par contre, il reste difficile de déterminer la taille optimale de la mémoire cache afin de maximiser la performance. Également, dans certains cas la taille optimale de la mémoire cache peut se révéler trop importante et donc difficile à implémenter.

La deuxième méthode consiste à stocker une petite partie du code et des données dans une mémoire locale au processeur et de stocker le reste dans la mémoire externe. Encore une fois, cette solution n'est pas parfaite puisqu'il reste difficile de déterminer les portions du programme à stocker dans cette mémoire locale. De plus, les outils

disponibles permettent rarement de morceler un programme et d'en stocker des portions dans des mémoires arbitraires. Finalement, il est toujours possible de tenter de diminuer la taille du programme en coupant dans le code ou en utilisant des librairies plus légères, mais cette approche n'est pas toujours réalisable en pratique.

1.3. Objectif

L'objectif de ce travail consiste à développer un outil qui permettra de tirer profit de la hiérarchie mémoire disponible sur un système sur puce afin d'améliorer la performance d'un programme s'exécutant sur un microprocesseur embarqué (ici le MicroBlaze). Une plateforme de simulation de système sur puce doit être utilisée afin de générer des métriques qui seront ensuite analysées par l'outil qui déterminera la meilleure combinaison de mémoire locale et de mémoire cache.

L'outil développé doit automatiquement minimiser la taille de la mémoire locale utilisée de même que la taille de la mémoire cache tout en tentant de maximiser la performance du programme à exécuter. Cet outil doit être plus rapide que les méthodes manuelles et ne doit être minimalement dépendant de la plateforme de simulation. Seul le mécanisme de génération de métriques doit dépendre de la plateforme de simulation. L'outil doit ensuite analyser les métriques d'exécution récoltées afin de proposer une configuration à l'utilisateur. L'outil d'analyse doit présenter à l'utilisateur certaines options de configuration comme la taille maximale de mémoire locale à utiliser. Finalement, les résultats doivent être présentés dans une interface graphique à titre d'application autonome, mais l'outil doit également avoir la possibilité d'être intégré dans d'autres environnements et doit donc posséder des interfaces de programmation simples à utiliser.

1.4. Méthodologie

Afin de réaliser l'objectif décrit ci-haut, plusieurs étapes sont requises. Premièrement, une familiarisation avec SystemC et l'environnement SPACE est nécessaire. De plus, une étude des concepts de modélisation de haut niveau comme par exemple le Transaction Level Modeling (TLM) est de mise. Ceci permettra de réaliser les étapes suivantes qui reposent sur la plateforme SPACE.

Ensuite, le simulateur de processeur MicroBlaze, disponible sur la plateforme SPACE, devra être étendu et amélioré afin de le mettre à niveau avec la dernière mouture disponible en matériel, notamment la présence de mémoire cache. L'implémentation de ce simulateur doit également être modifiée afin d'améliorer l'exactitude de la simulation en termes de cycles d'exécution. De plus, afin de connecter le processeur directement avec la mémoire comme il est possible en matériel, un contrôleur de mémoire SDRAM supportant le protocole XCL sera implémenté.

L'étape subséquente sera la définition et l'implémentation du mécanisme de récolte des métriques. Ce mécanisme fournira à l'outil d'analyse toute l'information nécessaire afin d'optimiser la configuration de la hiérarchie mémoire sous la forme de fichiers de trace d'exécution.

Par la suite, une méthode d'optimisation doit être développée afin de produire les résultats désirés. Le problème sera abordé en deux volets : le partitionnement de mémoire et l'optimisation de la mémoire cache. Le partitionnement de mémoire cible la relocalisation des instructions et données les plus utilisées dans la mémoire locale du processeur. L'optimisation de la mémoire cache vise, quant à elle, à minimiser la taille des mémoires cache d'instructions et de données et d'obtenir une bonne accélération. Pour chacun de ces volets, la littérature reliée devra être étudiée afin de recenser les différentes approches possibles.

Le partitionnement de la mémoire utilisera un profil d'accès mémoire pour lequel chaque fonction et donnée se voit attribuer un pourcentage d'utilisation du temps processeur. Les fonctions et données présentant un grand pourcentage seront alors relocalisées en mémoire locale. Viendra ensuite l'optimisation de la mémoire cache qui se fera en simulant les différentes configurations de cache possibles. Pour chaque configuration simulée, la performance de cette configuration sera estimée, ce qui permettra de déterminer la configuration de mémoire cache la plus appropriée.

L'étape finale consistera du développement de l'interface graphique qui permettra de démarrer l'analyse et de visualiser les configurations et les résultats obtenus. Les résultats seront visibles sous une forme textuelle et sous forme de graphiques. L'interface graphique doit être simple d'utilisation tout en restant flexible.

1.5. Contributions

Ce travail apporte deux contributions. Premièrement, de nouveaux composants exhibant les fonctionnalités disponibles en matériel, de même qu'un comportement temporel plus précis sont maintenant disponibles dans la plateforme SPACE, une plateforme de conception haut niveau de systèmes sur puce. Avec ces nouveaux composants, il est désormais possible d'obtenir des métriques de performance plus précises pour le processeur, mais également pour l'ensemble du système. Il est également possible d'obtenir une trace d'exécution qui peut être utilisée à d'autres fins que celles prévues lors de la conception.

Finalement, un outil complet de partitionnement et d'optimisation de la mémoire a été développé. Cet outil permet au concepteur de connaître la configuration mémoire avec laquelle sa plateforme produira la meilleure performance tout en tentant de minimiser le coût en matériel. Le concepteur n'a plus à tenter de réduire la taille du programme afin de le placer entièrement dans la mémoire locale ou d'effectuer des dizaines de

simulations afin de connaître la taille optimale de la mémoire cache. Avec l'outil de partitionnement et d'optimisation, le concepteur n'a qu'à simuler une fois son système pour générer les traces d'exécution, démarrer l'outil d'analyse et reconfigurer son système avec la configuration suggérée. Le concepteur peut ensuite simuler le système et comparer la performance du système en simulation avec la performance estimée par l'outil. Comparativement aux méthodes existantes, la méthode proposée permet d'optimiser à la fois la mémoire locale et la mémoire cache en plus de générer le fichier script de l'éditeur de liens (*linker script*) à l'aide d'un seul outil. L'outil permet également de visualiser les résultats dans une interface graphique, ce qui n'est pas possible dans les autres travaux existants

1.6. Organisation du mémoire

Ce mémoire est organisé en cinq chapitres. Le présent chapitre sert d'introduction aux systèmes embarqués et aux méthodes de conception de ces systèmes. Le deuxième chapitre présente une revue des différents concepts d'architecture de mémoire et des méthodes développées dans le passé afin de tirer profit de ces architectures. Le troisième chapitre décrit le développement et l'intégration de composants additionnels à l'environnement SPACE qui permettront de générer les données nécessaires à l'outil de partitionnement de la hiérarchie mémoire développé. Le chapitre 4 présente les objectifs et la conception de l'outil de même que les différents mécanismes utilisés par l'outil. Le dernier chapitre présente les gains de performance obtenus avec deux applications et présente des comparaisons entre les performances estimées par l'outil et les performances en simulation. Finalement, la conclusion résume le travail de recherche qui a été accompli et suggère des améliorations et des travaux futurs.

CHAPITRE 2

SYSTÈMES SUR PUCE: HIÉRARCHIE MÉMOIRE, LOGICIEL ET OPTIMISATION

L'avancement constant des procédés lithographiques permet aujourd'hui d'intégrer près de deux milliards de transistors sur une seule puce [8]. Ce nombre impressionnant de transistors signifie qu'il est maintenant possible de concevoir des systèmes multiprocesseurs sur puce contenant jusqu'à 8 processeurs ou même plus. Cette nouvelle réalité permet aux SoC de suivre une évolution similaire à celle des logiciels d'ordinateurs de bureau. Le développement peut maintenant s'effectuer à un niveau d'abstraction plus élevé et ainsi diminuer le délai de développement et de commercialisation d'un produit (TTM). Le développement passe donc du design de matériel dédié complexe au design de logiciel s'exécutant sur microprocesseur.

Ce chapitre introduit tout d'abord l'architecture mémoire des systèmes sur puce centrés sur un processeur. Par la suite, un survol du processus de compilation de la plateforme logicielle est effectué. Les différentes approches utilisées pour améliorer la performance de la plateforme logicielle à l'aide de la hiérarchie de mémoire sont ensuite discutées. Ensuite, la librairie SPACE qui servira à la simulation est introduite et est suivie d'une brève comparaison de l'approche utilisée avec les travaux antérieurs est présentée. Finalement, un bref retour sur les concepts vus au cours de ce chapitre est effectué.

2.1. Hiérarchie mémoire

L'exécution d'un logiciel sur un processeur implique la présence de différents composants. Une mémoire doit ensuite être présente afin de stocker le logiciel qui sera exécuté. Lorsque le logiciel devient volumineux, il devient nécessaire d'utiliser une

mémoire externe à la puce. L'utilisation d'une hiérarchie de mémoire devient alors inévitable afin d'assurer la performance du système. En effet, si la mémoire ne peut fournir le processeur en données, ce dernier se retrouve sous-utilisé et sous-performant.

La Figure 2.1 illustre l'architecture simplifiée d'un SoC. Ce SoC pourrait par exemple être un système sur FPGA basé sur les cœurs de processeurs logiciels MicroBlaze (de l'anglais *softcore*). Dans cette architecture, chaque processeur est connecté directement à une mémoire locale ainsi qu'à une mémoire cache. De plus, les processeurs sont reliés par l'entremise du bus système à une mémoire principale externe à la puce.

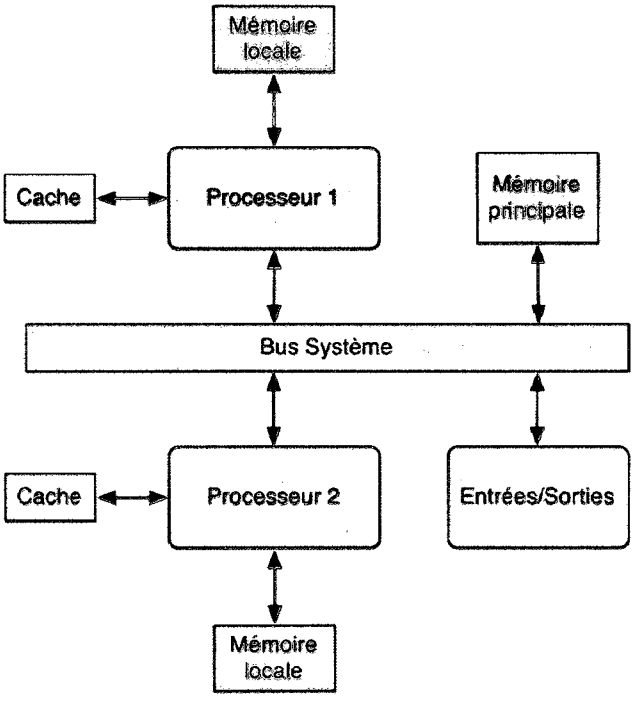

Figure 2.1 Architecture d'un SoC

Dans une telle situation, la hiérarchie de mémoire prend alors tout son sens. Si chaque processeur doit sans cesse utiliser le bus pour récupérer les instructions qu'il doit exécuter ainsi que les données à lire et à écrire, la bande passante disponible à chaque processeur sera divisée au mieux par deux. Lorsque les autres composants sur le bus entrent en jeu tel que les accélérateurs matériels, interfaces d'entrées/sorties ou encore

des modules DMA, la situation s'empire. Non seulement la mémoire ne peut suffire à la demande, mais le bus non plus. Une hiérarchie de mémoire est donc nécessaire afin d'alléger le trafic et de permettre aux processeurs de réaliser leur plein potentiel.

Une hiérarchie de mémoire se compose de mémoires petites et rapides en allant vers des mémoires de plus en plus volumineuses et lentes (Figure 2.2).

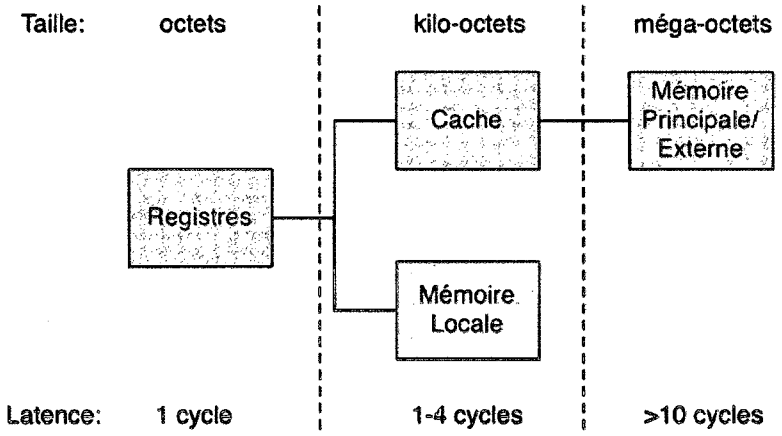

Figure 2.2 Hiérarchie de mémoire

Les registres sont internes au processeur et sont généralement de taille avoisinant les 100 octets. Ils sont généralement implémentés sous la forme de bascules et sont donc très rapides d'accès. La mémoire locale et la mémoire cache quant à elles sont plus volumineuses et peuvent aller jusqu'à plusieurs centaines de kilo-octets. Elles peuvent par contre être légèrement plus lentes que les registres avec une latence de quelques cycles. Finalement, la mémoire externe peut atteindre des tailles de quelques méga-octets à quelques giga-octets selon les besoins. Évidemment, ces tailles apportent un important coût en terme de performance. Par conséquent, le temps d'accès de ces mémoires peut être très grand, de l'ordre des dizaines de cycles et pouvant aller même à des centaines de cycles. Dans les sections suivantes, les mémoires principales, mémoires caches et mémoires locales seront discutées plus en détail.

2.1.1. Mémoire principale

La mémoire principale d'un SoC est généralement présente sous forme de mémoire DRAM et est externe au SoC. Cette technologie permet de produire de la mémoire volumineuse à un faible coût comparativement à la technologie utilisée pour les registres et la mémoire cache. Bien que cette technologie permette de stocker une grande quantité de mémoire sur une seule barrette (des barrettes de 4 Go sont maintenant disponibles sur le marché), le temps d'accès s'en trouve grandement affecté. En effet, il n'est pas rare de voir le processeur attendre 10 ou même 100 cycles pour accéder à une seule donnée.

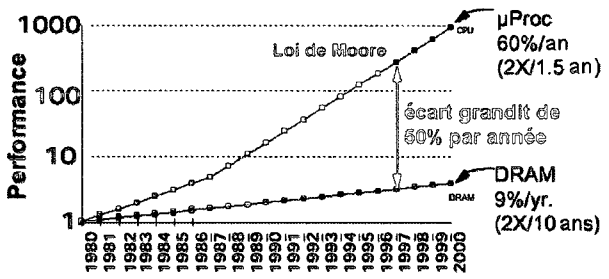

Figure 2.3 Comparaison de la performance des processeurs et de la mémoire DRAM

L'utilisation de la mémoire principale est souvent inévitable pour des systèmes relativement complexes mais a comme effet de ralentir considérablement le programme exécuté. Dans ce cas, il devient donc approprié d'ajouter au système une mémoire cache ou une mémoire locale afin d'accélérer la performance.

2.1.2. Mémoire locale

Certains systèmes, par exemple le MicroBlaze, disposent d'une connexion dédiée à de la mémoire intégrée sur le SoC. Sur d'autres systèmes, comme dans le cas d'un système basé sur le processeur PowerPC, cette mémoire est connectée au bus principal. Ce type de mémoire se nomme «mémoire locale» puisqu'elle est intégrée au SoC et connectée au

processeur. Son emplacement et sa technologie d'implémentation (souvent de la SRAM) lui permettent d'offrir des temps d'accès très rapides dans les environs de 1 à 2 cycles d'horloge.

La mémoire locale opère de façon identique à la mémoire principale, c'est-à-dire qu'elle doit être gérée de façon explicite. Il est donc du ressort du programmeur ou du compilateur de stocker les données dans cette mémoire. Son utilisation est donc bien différente de la mémoire cache qui est transparente au programme. L'avantage de cette approche est que le programmeur peut lui-même déterminer le contenu de cette mémoire et donc peut donner priorité à certaines données ou certaines fonctions. Ceci, couplé au temps d'accès constant, permet à la mémoire locale d'assurer le déterminisme. Cela contraste avec la mémoire cache qui ne peut offrir ce déterminisme puisque le cache tente de minimiser le temps d'accès moyen et non de garantir un temps d'accès minimal.

Les mémoire locales sont déjà très utilisées dans les processeurs DSP [9] et gagnent en popularité dans les processeurs conventionnels. Un exemple de processeur commercial utilisant plusieurs mémoires locales, de façon à permettre aux différentes unités de fonctionner à leur performance maximale, est le processeur Cell [10] de Sony, Toshiba et IBM. Ce processeur contient 8 co-processeurs vectoriels contenant chacun 256 Ko de mémoire locale. Chaque mémoire locale peut être accédée par les autres co-processeurs via un DMA (et moyennant un coût en latence d'accès) ce qui permet, par exemple, de réaliser une architecture multi processeurs pipelinée en minimisant les accès à la mémoire principale externe.

2.1.3. Mémoire cache

La mémoire cache [13] a pour but de diminuer le temps d'accès moyen aux instructions et aux données. C'est une mémoire rapide dont le contenu est modifié dynamiquement selon le comportement du programme exécuté. La mémoire cache tire profit des

principes de localité : elle met à profit le principe de localité temporelle et de localité spatiale.

Le principe de localité temporelle est le principe selon lequel une donnée récemment utilisée a de bonnes chances d'être réutilisée dans le futur. Les mémoires caches permettent donc de stocker dans une mémoire rapide les instructions et les données récemment utilisées dans l'espoir qu'elles seront réutilisées dans le futur. Ceci permet donc d'éviter d'accéder à la mémoire principale.

Le principe de localité spatiale quant à lui est le principe selon lequel la donnée qui suit (dans la mémoire) la donnée accédée a de bonnes chances d'être accédée dans le futur. En règle générale, les mémoires caches ne transfèrent donc pas uniquement la donnée nécessaire mais aussi les quelques données qui la suivent. Ceci est particulièrement utile lors de l'exécution d'une boucle, par exemple, où la même séquence d'instructions est exécutée à répétition.

L'implémentation d'un cache simple peut être conceptualisée sous la forme de 2 tableaux (Figure 2.4). Le premier tableau agit à titre d'index et le deuxième tableau agit sous forme de conteneur des données (la mémoire cache proprement dite).

Une mémoire cache simple de type à accès direct (*direct-mapped*) est tout d'abord divisée en m lignes de cache. La ligne de cache représente l'unité élémentaire manipulée par le contrôleur de cache. La taille d'une ligne de cache (n) se compte généralement en mots. Pour un processeur 32 bits, un mot représente une donnée de 32-bits. À chaque ligne de cache est associée une entrée dans l'index. Cette entrée indique si les données dans la ligne de cache sont valides ou non et contient également l'étiquette (*tag*) qui est le complément de l'adresse. À l'aide de l'étiquette et du numéro de ligne de cache, il est possible de reconstruire l'adresse de la donnée stockée en cache et donc de déterminer si

l'adresse de l'accès correspond à l'adresse de la donnée contenue en cache (succès ou *hit*).

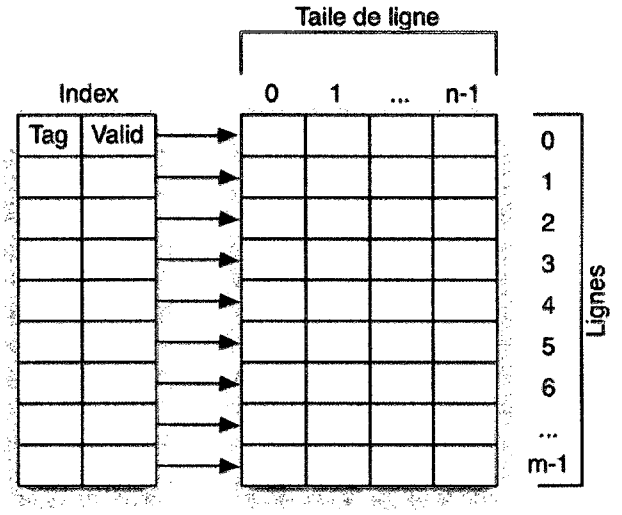

Figure 2.4 Mémoire cache à accès direct

Lorsque le processeur effectue une lecture, le contrôleur de cache doit déterminer si la donnée est présente ou non dans le cache. Si la donnée n'est pas présente, il doit lire la donnée de la mémoire principale et stocker cette donnée dans le cache. Dans le cas d'une mémoire cache *à accès direct* l'adresse d'une donnée est séparée en sections *étiquette, adresse de ligne* et *adresse de mot*. Selon la taille de l'espace d'adressage du processeur et la configuration du cache, chaque section englobera un nombre de bits différent. Considérons la situation suivante : un processeur avec un espace d'adressage de 32 bits, une taille de mots de 32 bits, un cache de 8 Ko et une taille de ligne de 8 mots. L'étiquette comprendra alors 19 bits, l'adresse de ligne comprendra 8 bits, l'adresse de mot 3 bits, les 2 bits les moins significatifs restants sont ignorés par le contrôleur de cache. Pour déterminer si une donnée est présente en cache, le contrôleur détermine d'abord à quelle ligne de cache correspond la requête. Ensuite, l'étiquette présente dans l'index est comparée avec celle de l'adresse de la requête et le bit de validité est vérifié. Si les étiquettes correspondent et que la ligne est valide, la ligne est récupérée dans la mémoire cache et le mot désiré est retourné. La Figure 2.5 (inspirée de [14]) illustre ce fonctionnement.

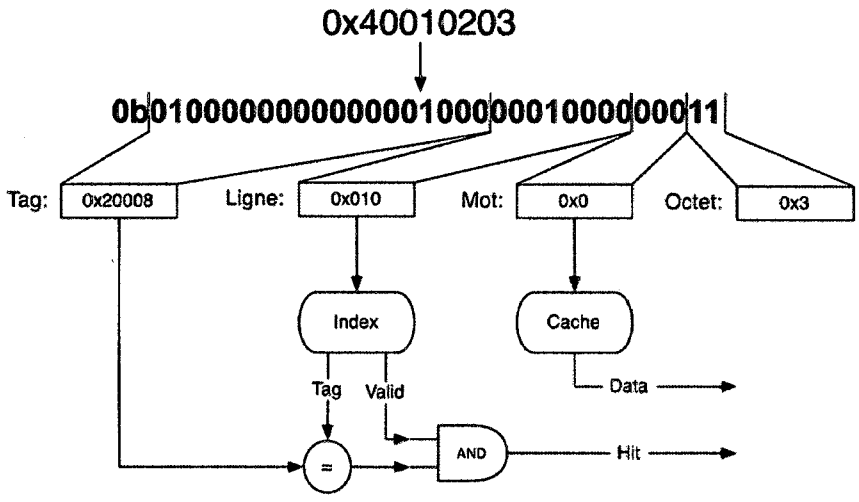

Figure 2.5 Fonctionnement d'une mémoire cache à accès direct

Lorsque la donnée demandée n'est pas disponible en cache, elle est alors récupérée et remplace la donnée déjà présente en cache. On parle alors d'échec (*cache miss*), dans le cas contraire, on parle de succès *(cache hit).* Ces concepts, de même que les métriques de performances des mémoires cache sont décrits dans [11] où l'auteur défini entre autres les trois types d'échecs: capacité, obligatoire et conflit.

Lorsqu'une donnée doit être écrite, deux approches sont fréquemment utilisées [12]. La première est d'écrire la valeur dans la mémoire principale peu importe si la donnée est présente en cache ou non, cette méthode porte le nom de politique d'écriture immédiate (*write-through policy*). La deuxième approche est de n'écrire la donnée que dans le cache et porte le nom de politique d'écriture différée (*write-back policy*). La première approche est simple à gérer. Lorsqu'une donnée doit prendre la place d'une autre dans le cache, la cohérence entre l'état du cache et la mémoire principale est assurée. Ceci n'est pas le cas pour la politique *write-back.* En effet, puisque la donnée n'est écrite que dans le cache, il y a discordance entre la donnée présente en cache et en mémoire. Le contrôleur doit donc utiliser un drapeau supplémentaire pour chaque ligne de cache afin de signifier qu'elle a été modifiée, le *dirty bit.* Si un remplacement de ligne doit être

effectué, le contrôleur vérifie si la ligne a été modifiée et si oui, l'écrit d'abord dans la mémoire principale pour ensuite aller récupérer la nouvelle ligne. Le *write-back* implique donc une logique supplémentaire par rapport au *write-through* mais elle permet de diminuer les accès à la mémoire principale.

Bien qu'un cache *à accès direct* soit simple à implémenter, il n'est pas adapté à tous les scénarios. Plusieurs concepts entrent donc en jeu afin d'améliorer la performance des caches dans ces conditions. L'associativité est un concept permettant d'éviter des remplacements inutiles de lignes. En relation avec l'associativité, la politique de remplacement permet de déterminer quelle ligne remplacer. L'utilisation de niveaux multiples de cache permet également d'améliorer la performance. Puisque la méthode présentée ne touche pas à ces concepts, le lecteur désireux d'en apprendre plus peut se référer à [13], [15] et [16].

2.2. Structure d'un logiciel compilé

Pour qu'un code source soit converti en un programme exécutable, une série d'opérations doit être effectuée sur ce code afin de le rendre compréhensible au processeur cible [18]. Les deux étapes majeures de ce processus sont la compilation du code et l'édition des liens. Lorsque ces étapes sont terminées, un fichier exécutable est généralement créé selon un format spécifique. Dans le cas du processeur MicroBlaze, ce fichier exécutable est au format ELF. Finalement, une dernière étape (souvent optionnelle) consiste à profiler le logiciel afin de déterminer les points chauds de celui-ci et donc de réduire l'impact de ces points chauds sur le temps d'exécution de ce dernier.

2.2.1. Compilation

Compiler un code source (ici du C) est un processus complexe qui peut se diviser en trois étapes [17] :

1. Frontale
2. Optimisation
3. Finale

L'étape frontale se charge de traduire le code source en une représentation intermédiaire plus facile à comprendre pour le compilateur : l'arbre syntaxique abstrait ou *Abstract Syntax Tree* (AST). Lorsque cette étape est terminée, l'étape d'optimisation prend la relève et optimise le programme en effectuant des simplifications et des optimisations de boucles. Finalement, à la troisième étape, la représentation intermédiaire est convertie en code machine spécifique à l'architecture ciblée.

Le processus de compilation ne fait pas que compiler du code, il sert également à séparer le programme en différentes sections. Ceci permet de compartimenter un programme et devient utile lors de la phase d'édition des liens. Un programme est normalement constitué de quelques sections de base [19] :

- .data
- .bss
- .text
- .stack
- .heap

La section `.data` contient des données initialisées, la section `.bss` contient des données non-initialisées et finalement, la section `.text` contient le code lui-même ainsi que des

données qui sont en lecture seulement. Finalement, les sections `.stack` et `.heap` contiennent respectivement les données locales aux fonctions (pile) et les données allouées dynamiquement (tas). Fait à noter : la pile croît dans le sens négatif des adresses en mémoire tandis que le tas croît dans le sens positif.

Il est par contre possible de spécifier des sections additionnelles en plus des sections prédéfinies. Ceci est fait à l'aide d'un attribut pouvant être spécifié à la fin de la déclaration d'une variable, classe, structure ou fonction [20]. La syntaxe pour spécifier une section est la suivante :

```
type_var nom_var __attribute__ ((section("nom_section")));
```

Une autre méthode proposée par le compilateur GCC est d'utiliser deux options lors de la compilation :

- `-ffunction-sections`
- `-fdata-sections`

Ces options de compilation permettent d'assigner une section à chaque fonction (`-ffunction-sections`) et à chaque variable globale (`-fdata-sections`). Le nom des sections ainsi créées dépend de la section dans laquelle la variable ou la fonction se trouve et de son nom. À titre d'exemple, pour la fonction :

```
void Func_1(char a, char b)
```

qui réside dans la section *.text* par défaut, le compilateur lui assignera la section :

```
.text._Z6Func_1cc
```

Cette nouvelle section peut ensuite être utilisée par l'éditeur de liens pour spécifier un emplacement mémoire pour cette fonction. Si l'option n'est pas utilisée lors de la compilation, il n'est pas possible de déplacer manuellement la fonction en mémoire et le compilateur s'occupe de déterminer l'emplacement le plus approprié.

2.2.2. Édition des liens

Suite à la compilation, le fichier objet créé ne contient aucune référence à des adresses en mémoire. Les références se font à l'aide des noms des fonctions et des variables (symboles). L'édition des liens est l'étape qui permet de remplacer ces symboles par leur adresse définitive dans la mémoire. Ce processus permet également d'intégrer le code provenant de librairies statiques au programme final ou de lier le programme à des librairies partagées [21]. Le résultat sera un exécutable ou une librairie, souvent sous le format ELF [24].

Une fonction importante de l'éditeur de liens est la possibilité de spécifier manuellement l'adresse désirée des différentes sections d'un programme [22]. Ceci peut être effectué à l'aide d'un script qui sera ensuite passé en paramètre à l'éditeur de liens. À l'aide de cette fonctionnalité, il est possible de placer des sections à n'importe quelle adresse dans n'importe quelle mémoire disponible sur le système [23].

2.2.3. Profilage

Le profilage d'un programme est le processus selon lequel de l'information sur le temps d'exécution et la fréquence d'exécution des fonctions d'un programme est recueillie. Profiler un programme suggère généralement que le développeur désire connaître les points chauds de son application afin de pouvoir accélérer ces points chauds. En effet, selon la loi la du 90/10, un programme passe environ 90% de son temps à exécuter 10% du code [13]. Il est donc payant d'optimiser ce 10%.

Le profilage implique généralement l'ajout d'instructions qui comptent le nombre de cycles passés dans une fonction. Cette approche est utile lorsque le programmeur ne peut que compiler et exécuter son programme sur un processeur existant. Un fichier peut alors être généré et peut servir à un outil d'interprétation graphique. L'inconvénient de cette approche est qu'elle peut avoir un impact non négligeable sur le temps d'exécution du programme et peut biaiser les résultats. Par contre, elle ne nécessite généralement aucune modification manuelle au programme de la part du développeur puisque les outils existants se chargent eux-mêmes d'instrumenter le code [25].

Une deuxième approche est d'utiliser un simulateur d'instructions (ISS) qui peut alors intercepter les instructions exécutées par le processeur simulé [26]. Dans ce cas, le code original peut être exécuté sans insertion de code d'instrumentation. Cette approche a l'avantage de n'avoir aucun impact sur le temps d'exécution simulé du programme et par conséquent ne biaise pas les résultats. Par contre, il est nécessaire de disposer d'une plateforme de simulation et le fait de simuler l'exécution ralentit sensiblement la vitesse d'exécution.

2.3. Optimisation de la hiérarchie

L'ajout d'une certaine hiérarchie mémoire implique qu'il devient nécessaire d'effectuer une certaine exploration architecturale afin d'obtenir les meilleures performances possibles. Bien entendu, il est possible d'effectuer ces explorations manuellement lorsque les configurations offertes sont limitées. Par contre, lorsque l'on a affaire à un SoC hautement configurable, l'utilisation d'une méthode automatique devient alors incontournable. Dans cette section, les méthodes de partitionnement de mémoire ainsi que les méthodes d'optimisation de la configuration de la mémoire cache sont présentées.

2.3.1. Mémoire locale

Avec l'utilisation croissante de la mémoire locale à l'intérieur de processeurs et des SoC, plusieurs techniques d'optimisation de l'utilisation de la mémoire locale ont été développées. Lorsqu'aucune technique d'optimisation n'est employée, l'utilisation de la mémoire locale se résume à:

- Contenir les vecteurs d'interruption du programme,
- Contenir le programme en entier, ou
- À titre de mémoire tampon[28].

Lorsque la mémoire locale ne contient que les vecteurs d'interruption, le programme est contenu dans une mémoire externe de taille plus importante. Ceci permet d'avoir des programmes de grande taille mais induit un délai significatif.

Si le programme est de taille inférieure à la taille de la mémoire locale, il est possible de le placer entièrement dans la mémoire locale. Ceci permet une rapidité d'accès optimale aux instructions et aux données, mais impose une contrainte de taille significative à l'application développée. En effet, il suffit d'exécuter une application basée sur le RTOS µC pour observer que même ce système d'exploitation simple dépasse la taille maximale de 64 Ko pour un contrôleur de mémoire locale. Il va sans dire qu'il est donc très facile d'être limité par la taille pour des systèmes complexes.

Finalement, la mémoire locale peut être utilisée comme tampon (*scratchpad)*. Une mémoire tampon est une mémoire qui sert à stocker temporairement des données ou des instructions lorsque le programmeur désire accéder rapidement ou de façon déterministe à ces données. Ceci est un avantage de la mémoire tampon par rapport au cache: le temps d'accès est déterministe. Par contre, c'est au programmeur de s'assurer que les bonnes données seront présentes lorsqu'elles seront nécessaires.

Afin de remédier aux limites et à la difficulté pour le programmeur de manipuler manuellement le contenu de la mémoire locale, plusieurs techniques ont été développées. Ces techniques se divisent en deux catégories : statiques ou dynamiques. Pour les méthodes d'optimisations statiques, le contenu de la mémoire locale est déterminé à l'aide d'analyse de traces [27] ou d'analyse de code. Le contenu de la mémoire locale restera alors fixe durant l'exécution du programme. Dans le cas des méthodes d'optimisation dynamiques, un sous-système, semblable à de la mémoire virtuelle, charge les données et les instructions dynamiquement dans la mémoire locale durant l'exécution du programme. L'avantage de cette approche est la plus grande flexibilité du contenu de la mémoire locale mais elle impose des coûts de gestion de la mémoire.

Parmi les méthodes statiques notons tout d'abord la solution proposée par [29] qui présente une méthode d'optimisation de la performance d'un système embarqué utilisant une combinaison de mémoire tampon et de cache. Bien que la méthode proposée permette de déterminer automatiquement ce qui devrait être placé dans la mémoire tampon, elle ne propose pas de méthode pour effectuer l'allocation. De plus, les tailles des caches ne sont pas déterminées automatiquement. Par contre, la conclusion des auteurs est qu'il est important d'utiliser judicieusement la mémoire locale et la mémoire cache afin d'obtenir la meilleure performance possible, ce qui valide le travail présenté dans ce mémoire.

Dans [30], la méthode présentée, plus complète, utilise tout d'abord une trace d'exécution afin de recueillir la fréquence d'utilisation de chaque adresse. Ensuite, un algorithme analyse les entrées dans le profil créé précédemment et détermine la solution optimale selon la taille fixée de la mémoire locale, le gain de performance engendré pour chaque entrée et le coût de partitionnement de cette entrée. Finalement, un outil modifie directement le code exécutable afin de relocaliser les blocs dans la mémoire locale. Cette méthode permet d'améliorer la performance de 20% par rapport à la performance obtenue lorsque le système utilise uniquement la mémoire cache.

Plusieurs travaux [31] [32] et [33] ont abordés les méthodes de partitionnement dynamiques. La méthode proposée par [33] est particulièrement intéressante puisqu'elle utilise le MMU du processeur afin de permettre la gestion du contenu de la mémoire locale. Pour utiliser cette fonctionnalité du processeur, les auteurs ont développé un algorithme d'analyse ainsi qu'un moteur d'exécution (*runtime*). Afin de permettre le fonctionnement de ce moteur d'exécution, un profil de l'application est tout d'abord généré par l'exécution du programme à optimiser. Ensuite, un outil analyse ce profil et détermine les dépendances entre les blocs de base [34]. L'outil modifie ensuite le programme afin d'inclure ces données d'analyse et d'intégrer le moteur d'exécution dans le fichier binaire. L'approche permet donc d'utiliser la mémoire locale et la mémoire externe comme mécanisme de mémoire virtuelle. Ce mécanisme est ensuite combiné à de la mémoire cache pour améliorer davantage les performances. Des gains de performance de 12% sont atteints par rapport à un système utilisant uniquement de la mémoire cache.

Il existe nombre de systèmes développés utilisant des mémoires locales afin d'améliorer la performance. Un exemple d'un tel système mettant à contribution des mémoires tampons ainsi que des mémoires cache est proposé dans [35]. Cette approche, combinée à une architecture utilisant trois processeurs configurables permet d'obtenir un gain de performance de 22,3x par rapport à une architecture monoprocesseur conventionnelle.

2.3.2. Mémoire cache

L'optimisation de la configuration des caches a suscité beaucoup d'intérêt dans les dernières années et les outils d'exploration architecturale sont nombreux. Les méthodes sont toutes centrées sur l'utilisation d'une trace d'exécution et d'un simulateur de cache. Par contre, les approches diffèrent sur la méthode utilisée pour simuler et également sur la méthode utilisée pour recueillir la trace d'exécution.

L'avantage d'utiliser une trace d'exécution est qu'il devient possible d'effectuer l'optimisation en différé mais surtout de diminuer le temps de simulation du cache en compactant la trace. En effet, une trace d'exécution peut devenir rapidement très volumineuse. Par exemple, la trace d'une seule seconde d'exécution d'un processeur cadencé à 100MHz peut occuper jusqu'à 500 Mo d'espace disque.

Dans [37], les auteurs présentent une méthode d'évaluation de la performance d'un cache en utilisant une approche basée sur l'échantillonnage de la trace d'exécution. Puisqu'un échantillonnage est utilisé, l'ensemble de la trace d'exécution n'a pas à être gardé en mémoire et un gain significatif de vitesse de simulation peut être observé.

La méthode proposée dans [38] est une méthode itérative qui réduit la taille de la trace à simuler à chaque itération. Ceci permet de simuler rapidement un grand nombre de configurations de cache. Par contre, ce modèle ne peut garantir l'exactitude des résultats.

W.-H. Wang et J.-L. Baer [39] proposent quant à eux de réduire dès le départ la trace en utilisant un algorithme de compression. Leur méthode simule un cache associatif mais, afin de compresser la trace, ils simulent d'abord un cache à accès direct et ne gardent que les instructions qui causent un échec ou une écriture. Ceci leur permet d'obtenir une trace réduite et d'utiliser cette dernière pour simuler l'ensemble des configurations de cache associatif qui sont plus coûteuses en termes de temps à simuler.

Il ne suffit pas uniquement de réduire la taille de la trace pour obtenir des performances satisfaisantes de simulation de cache. Il importe aussi d'améliorer l'algorithme de simulation. Dans [40] les auteurs présentent un outil permettant de simuler plusieurs configurations de cache en une seule itération. Cet outil s'appuie sur le principe d'inclusion [36] qui stipule que si une donnée provoque un succès pour une certaine taille de cache, les caches de plus grande taille sont assurés de générer également un

succès pour cette donnée. De plus, les auteurs utilisent une quantité d'améliorations à l'algorithme leur permettant d'accroître significativement l'efficacité de la méthode.

Plusieurs autres outils de simulation de cache ont été développés au fil des ans. Les outils Dinero IV [41] et CACTI [42] sont probablement les outils les plus connus, mais d'autres outils comme Mcsim [43] sont également disponibles. Par contre, ces outils ne se concentrent pas sur l'optimisation de mémoires caches mais surtout sur la simulation pure de ces caches.

2.4. Environnement de simulation SPACE

SPACE [44] est un environnement de conception de SoC qui repose sur SystemC. Cet environnement propose de multiples composantes basées sur un modèle TLM: bus, simulateurs d'instructions (ISS), UART mais permet également à l'utilisateur d'écrire ses propres «modules » qui communiquent avec le reste du système en suivant le protocole SPACE.

SPACE permet de concevoir et de simuler des systèmes complets et de passer facilement d'un niveau d'abstraction à un autre. De plus, l'environnement permet de faire un partitionnement logiciel/matériel. En d'autres mots, le concepteur peut, sans recoder son module, passer d'une exécution logicielle sur un processeur à un module matériel connecté sur un bus et vice-versa.

La plateforme SPACE rend ces fonctionnalités possibles grâce à l'utilisation d'une couche d'abstraction SystemC/RTOS qui permet d'exécuter un module SystemC en tant que tâche sur un RTOS et ce, tout en gardant les communications avec les autres modules matériels et logiciels intactes. De plus, SPACE utilise le concept d'adaptateurs qui permettent de connecter un module utilisateur à un bus TLM à haut niveau ou une

implémentation à bas niveau. Ces fonctionnalités permettent de réaliser et de simuler un SoC rapidement en tirant avantage des possibilités offertes par la librairie SystemC.

Cette dernière section conclut donc ce chapitre dans lequel les concepts de base des systèmes sur puce, des méthodes d'optimisation de la hiérarchie mémoire de tels systèmes ont été expliqués. Armé de ces outils, les mises à jour à effectuer à la plateforme de simulation SPACE afin de permettre le support d'un outil d'optimisation de la hiérarchie mémoire peuvent alors être décrites. Le chapitre suivant se concentre sur cette tâche qui ouvre la voie à l'implémentation de l'outil MemoryOptimizer.

CHAPITRE 3

MISE À JOUR DE LA PLATEFORME DE SIMULATION SPACE

Les travaux réalisés antérieurement (voir section 2.3) ont démontrés qu'il est nécessaire d'utiliser une plateforme de simulation afin de concevoir et d'implémenter un outil permettant d'optimiser la configuration de la hiérarchie mémoire. Cette plateforme permet de recueillir les traces d'exécution et de simuler la configuration finale. Dans le cas présent, l'environnement de simulation utilisé est la plateforme SPACE. Cette plateforme, décrite brièvement à la section 2.4, doit par contre être modifiée afin d'accommoder les fonctionnalités requises par l'outil développé.

Ce chapitre traite tout d'abord de la plateforme MicroBlaze de la compagnie Xilinx, plateforme matérielle qui sert de base au plus bas niveau d'abstraction de SPACE. L'état des composants dans la version courante de SPACE sera ensuite discuté. Par la suite, les modifications apportées à ces composants seront expliquées et finalement, l'outil permettant d'effectuer le traçage de l'exécution sera présenté.

3.1. Plateforme MicroBlaze

La plateforme MicroBlaze est la plateforme de choix de l'environnement de simulation SPACE. Afin de permettre la récolte de résultats significatifs cette plateforme a donc été choisie puisque la majorité des composants sont déjà implémentés et testés.

Dans un premier lieu, cette section présentera un survol de l'architecture de SoC proposée par Xilinx. Par la suite, le MicroBlaze sera présenté et finalement, le contrôleur MCH OPB SDRAM sera décrit brièvement.

3.1.1. Architecture Xilinx

La plateforme Xilinx spécifie une architecture de système sur puce complète : processeurs embarqués, bus, composants d'entrée/sortie, interfaces bus pour des composantes utilisateur, interfaces point-à-point, etc. Sur cette plateforme, les processeurs sont généralement le point central et servent à orchestrer le fonctionnement du système. Chaque processeur et périphérique étant relié à un bus, une hiérarchie de bus peut être implémentée afin de distribuer le trafic. De plus, les processeurs MicroBlaze et PowerPC implémentent chacun une interface point à point, FSL [46] et APU [47] respectivement, qui permettent de communiquer directement avec un co-processeur sans passer par le bus. La Figure 3.1 illustre un exemple d'un système sur puce comprenant un processeur PowerPC et un processeur MicroBlaze.

Dans cet exemple, le PowerPC est connecté à une mémoire locale OCM (On-Chip Memory) ainsi qu'au bus rapide PLB. À ce bus sont également connectés un contrôleur réseau Ethernet, une mémoire externe SDRAM et un contrôleur DMA permettant d'effectuer des transferts sans l'intervention du processeur. Ceci forme un sous-système capable d'effectuer dans calculs intensifs et d'interagir avec des périphériques rapides.

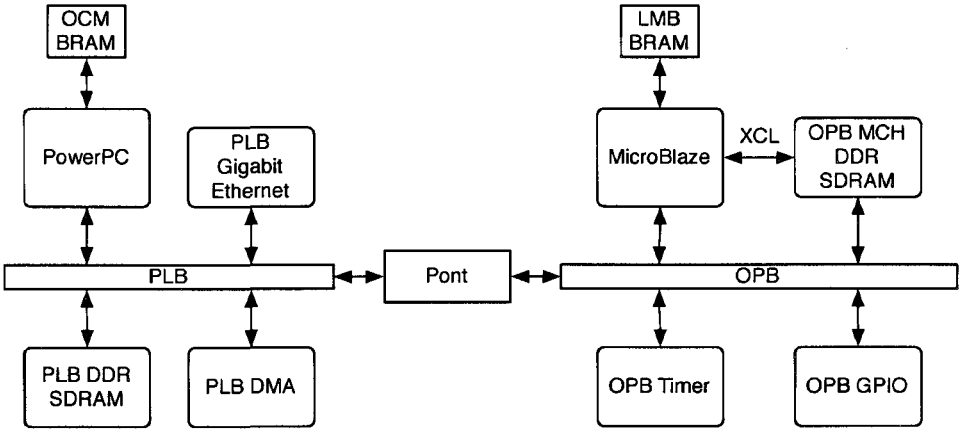

Figure 3.1 Exemple d'une plateforme Xilinx

Du côté du sous-système de droite, on retrouve, connecté au processeur MicroBlaze, une mémoire locale LMB, un lien direct à la mémoire externe XCL et une connexion au bus lent OPB. On retrouve également une minuterie, une mémoire externe SDRAM ainsi qu'une interface à utilisation générale GPIO. Ceci constitue un sous-système centré plus particulièrement sur le contrôle de périphériques lents.

L'utilisation des deux bus (PLB et OPB) permet de réduire la contention et permet également d'utiliser deux domaines d'horloges séparés ce qui a comme effet d'améliorer la performance du système. Par contre, afin d'assurer la communication entre les deux sous-systèmes, il est nécessaire d'ajouter un point. Ce pont permet de traduire les protocoles PLB vers OPB et OPB vers PLB. Il devient donc possible, par exemple, au sll PowerPC de communiquer avec le MicroBlaze afin de lancer une tâche d'entrée-sortie.

La plateforme Xilinx supporte donc principalement (selon le FPGA utilisé) deux types de processeurs : le processeur PowerPC [48] et le processeur MicroBlaze [49]. Un microcontrôleur 8 bits nommé PicoBlaze [50] est également disponible mais ne sera pas discuté ici puisqu'il n'est pas adapté aux types d'applications ciblées. Chacun de ces processeurs dispose également d'un bus adapté au processeur : le PowerPC utilise le bus PLB [51] tandis que le MicroBlaze dispose du bus OPB [52].

Sur les FPGA de Xilinx, le processeur PowerPC est intégré sur le circuit et ses options de configuration sont limitées. En particulier, les tailles des caches sont fixes et le processeur ne contient aucune instruction optionnelle. Le processeur MicroBlaze, quant à lui, est un cœur de processeur logiciel (*softcore*) c'est-à-dire qu'il est instancié sur le FPGA. Contrairement au PowerPC, le MicroBlaze est très configurable : il propose un nombre d'instructions optionnelles, un pipeline de profondeur configurable et des caches de taille configurable. Les deux processeurs proposent un accès dédié à de la mémoire locale : le PowerPC possède le bus OCM [53] tandis que le MicroBlaze possède le bus LMB [54]. Vu ces différences de configurabilité et, puisque le but premier de l'outil

d'optimisation de la hiérarchie mémoire est de configurer automatiquement la taille et le contenu de la mémoire locale ainsi que la taille des mémoires caches, le processeur MicroBlaze a donc été choisi.

3.1.2. Processeur MicroBlaze

Le MicroBlaze est un processeur embarqué 32 bits *softcore* ciblant les applications de contrôle. Ce processeur embarqué comprend la majorité des fonctionnalités requises par ce type de système : pipeline, multiplieur et diviseur en matériel, support d'exceptions et interruptions, contrôleur de cache configurable, unité à virgule flottante, liens point à point, bus pour la mémoire locale et bus système. Ces ressources font du MicroBlaze un excellent candidat pour les tâches de calcul et de contrôle.

3.1.2.1. Architecture

La Figure 3.2 présente un schéma bloc simplifié du MicroBlaze version 6. On peut remarquer dans ce schéma le grand nombre d'options configurables (bloc en gris). Bien que le MicroBlaze ne soit pas aussi flexible qu'un processeur configurable comme le Xtensa [55], il reste hautement configurable comparativement au PowerPC. Comme options de configuration, le MicroBlaze possède un pipeline de 3 ou 5 étages qui permet d'améliorer les performances en exécutant des instructions en parallèle. Le nombre de liens point à point (FSL) peut être configuré jusqu'à 8 et une vingtaine d'instructions peuvent être activées, comme par exemple les instructions de calcul à virgule flottante. D'intérêt particulier au projet sont les caches d'instructions et de données qui sont indépendamment configurables et qui permettent donc d'effectuer une exploration architecturale afin d'améliorer la performance du logiciel à exécuter.

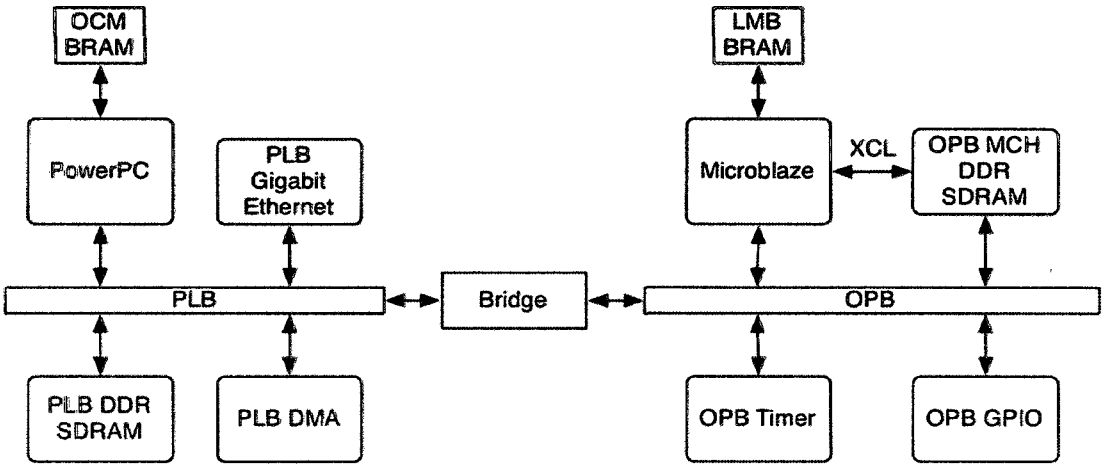

Figure 3.2 Schéma bloc du MicroBlaze

3.1.2.2.Pipeline

Sous la configuration à 3 étages du pipeline, on retrouve les étages *Fetch*, *Decode* et *Execute*. L'étage *Fetch* se charge de récupérer les instructions dans la mémoire selon le *Program Counter* (PC) courant. L'étage *Decode*, comme son nom l'indique, décode l'instruction afin d'acheminer celle-ci vers la bonne unité d'exécution et récupère la valeur des opérandes dans les registres. Finalement, l'étage *Execute* se charge d'exécuter l'instruction, de lire ou écrire les données dans la mémoire et d'écrire la valeur calculée dans le registre de destination. La configuration à 3 étages a pour but de réduire l'utilisation de la logique sur le FPGA mais a également l'inconvénient de réduire la performance du processeur par rapport à la configuration à 5 étages.

La configuration à 5 étages suit l'architecture habituellement retrouvée dans les processeurs RISC simples [13], c'est à dire qu'elle inclut les étages *Fetch*, *Decode*, *Execute*, *Memory* et *Writeback*. Les étages *Fetch* et *Decode* sont identiques à ceux de la configuration 3 étages. La différence se situe dans les étages suivants. L'étage *Execute* n'effectue que les opérations de calcul et d'accès FSL. L'étage *Memory* se charge d'effectuer les lectures et les écritures des données en mémoire. Finalement, l'étage *Writeback* écrit la valeur calculée dans le registre de destination. Cette configuration

implique une plus grande latence pour l'exécution des instructions et une plus grande utilisation de logique mais permet une plus haute fréquence de fonctionnement.

L'utilisation d'un pipeline implique un délai de trois cycles lorsqu'un branchement est pris puisqu'il devient alors nécessaire de vider les étages Fetch et Decode. Le MicroBlaze supporte donc les branchements avec *delay slot* afin de réduire ce délai. Cela signifie que l'instruction suivant l'instruction de branchement sera toujours exécutée que ce branchement soit prit ou non. La présence de ce mécanisme réduit donc à deux cycles le délai de branchement. Certaines instructions ne peuvent se trouver dans la *delay slot* : les instructions IMM (immédiat), les branchements et les instructions *break* (utilisées pour le débogage) puisqu'elles causeraient un comportement erratique de la part du processeur. De plus, si une interruption survient lorsque le processeur exécute une instruction dans la *delay slot*, le traitement de cette interruption sera retardée jusqu'à ce que l'instruction dans la *delay slot* ait terminé son exécution.

Puisque le processeur emploie des instructions multi-cycles comme la division et permet les accès à des périphériques lents, le pipeline doit être bloqué par moment. Afin de minimiser l'impact de ce blocage, un tampon d'instructions est utilisé afin de permettre au processeur de récupérer un certain nombre d'instructions durant le blocage. Ceci permet donc au processeur de reprendre l'exécution plus rapidement. De plus, puisqu'un pipeline introduit des problèmes de dépendance des données, un mécanisme de court-circuitage (*forwarding*) est implémenté permettant d'acheminer le résultat d'un étage à un autre afin de maximiser la performance.

Le MicroBlaze est conçu selon l'architecture Harvard, c'est-à-dire que les mémoires d'instructions et de données sont séparées physiquement. Le processeur implémente donc des bus LMB, OPB et XCL distincts pour les instructions et les données. La mémoire cache est organisée de façon similaire : une mémoire cache pour les instructions et une pour les données.

3.1.2.3.Cache

La mémoire cache disponible sur le MicroBlaze est une mémoire de type *accès direct* qui peut varier en taille de 64 octets à 64 kilo-octets. La taille des lignes de cache est également configurable entre 4 mots (16 octets) et 8 mots (32 octets). De plus, il est possible de spécifier la plage d'adresse que le cache pourra contenir. Ceci signifie que lorsqu'un accès mémoire est fait à l'intérieur de cette plage, la donnée peut être placée dans la mémoire cache. Si l'adresse se trouve à l'extérieur de la plage, le processeur ne peut utiliser le cache pour accélérer l'accès à cette donnée et doit donc toujours passer par le bus. La plage des adresses pouvant être mises en cache est un mécanisme indispensable pour les communications avec des périphériques sur le bus. En effet, si toutes les adresses pouvaient se trouver en cache, les lectures/écritures dans les périphériques pourraient ne pas refléter l'état réel du périphérique puisque cet état peut être modifié sans que le processeur n'en soit avisé. Finalement, le cache du MicroBlaze adopte une politique d'écriture de type *write-through* et donc chaque donnée est écrite à la fois dans le cache et dans la mémoire principale.

3.1.2.4.Lien XCL

Dans la majorité des processeurs, les accès aux données en mémoire ne se trouvant pas en cache doivent passer par le bus système afin de récupérer ou d'écrire la donnée en mémoire. Cela engendre un délai de quelques cycles qui s'ajoute à la latence d'accès de la mémoire. Afin de contrer ceci, le MicroBlaze propose un bus nommé XCL qui permet de connecter le processeur directement à un contrôleur mémoire supportant ce lien. En accord avec l'architecture Harvard, deux liens XCL sont disponibles : IXCL pour les instructions et DXCL pour les données.

Le lien XCL est basé sur une paire de liens FSL : 1 lien maître en écriture et un lien esclave en lecture. Chaque lien possède une largeur de 32 bits, correspondant à la taille

d'un mot. Une fonctionnalité importante du XCL est de permettre de récupérer une ligne de cache de 4 ou 8 mots en une seule lecture. De plus, puisqu'une ligne de cache est récupérée à chaque lecture, le mécanisme de mot critique d'abord - *Critical Word First* (CWF) en anglais - est utilisé afin d'éviter le blocage du processeur si le mot requis n'est pas situé au début de la ligne de cache. L'adresse du mot critique est envoyée au contrôleur de mémoire et la ligne de cache contenant le mot critique est retournée en débutant l'envoi par le mot critique, suivi des autres mots de la ligne de cache. Ceci permet au processeur de poursuivre son exécution dès que le mot critique est récupéré sans devoir attendre le reste de la ligne de cache. Quant à l'écriture, elle se fait à raison d'une donnée par accès : mot, demi-mot ou octet. La Figure 3.3 illustre le fonctionnement du protocole XCL.

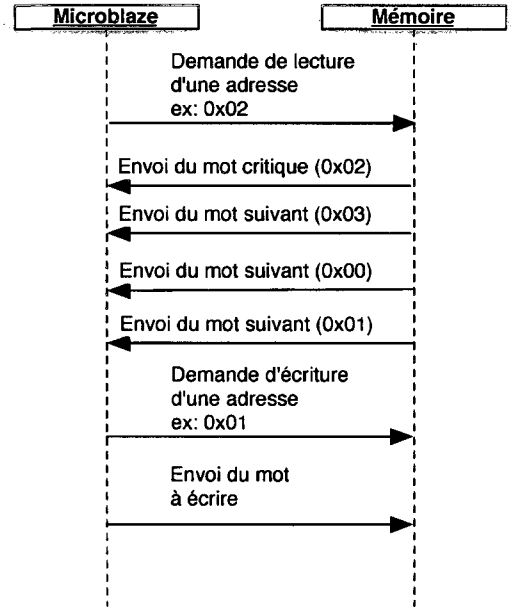

Figure 3.3 Diagramme de séquence du protocole XCL

Contrairement aux autres bus, comme le bus LMB ou OPB, le XCL ne découple pas les adresses et les données. Dans le cas d'une écriture, il devient donc nécessaire d'envoyer d'abord l'adresse suivie de la donnée à écrire. Puisque le XCL utilise des liens FSL, chaque lien comprend une mémoire tampon contenant jusqu'à 16 mots ou 8 données en

écriture. Par conséquent, lorsque le tampon n'est pas plein, deux coups d'horloge sont nécessaires. Dans le cas contraire, le processeur doit attendre que le tampon se vide pour pouvoir écrire. Pour la lecture, l'adresse à lire est d'abord envoyée, suit ensuite un minimum de 1 cycle d'attente entre l'envoi de l'adresse et la réception du premier mot et finalement, un minimum de 1 cycle pour chaque mot de la ligne de cache. Ceci fait un total de 6 coups d'horloge au minimum par lecture. Évidemment, ces temps sont significativement augmentés lorsqu'une mémoire lente, comme de la SDRAM, est utilisée.

3.1.3. Contrôleur de mémoire MCH OPB SDRAM

Afin de supporter des programmes volumineux, la plateforme Xilinx propose un grand nombre de contrôleurs de mémoire externe : EMC, SDRAM, DDR SDRAM et DDR2 SDRAM. En plus de proposer ces contrôleurs en version standard OPB, une version spéciale supportant également l'interface XCL est disponible. Ces contrôleurs permettent donc d'améliorer la performance lorsqu'ils sont liés à un processeur MicroBlaze via l'interface XCL.

Le contrôleur MCH (Multi Channel) OPB SDRAM supporte jusqu'à 4 canaux MCH supportant le protocole XCL. Un contrôleur peut donc accueillir deux processeurs MicroBlaze, chaque processeur utilisant deux liens XCL : un pour les instructions et un pour les données. Chaque canal peut supporter une taille de ligne de cache différente de 1, 4, 8 ou 16 mots, configurable à la synthèse. De ces tailles de ligne de cache, le MicroBlaze ne supporte que les tailles de 4 et 8 mots. Le contrôleur permet aussi d'améliorer la performance des accès sur le bus en supportant les transferts en mode «burst».

Afin de minimiser les ressources utilisées, les interfaces OPB et MCH sont entièrement optionnelles. Il est donc possible de réduire la taille du contrôleur d'environ 266 *slices*,

246 LUT et 166 *flips-flops* sur un FPGA Virtex-4 [56] en désactivant le support du bus OPB si le concepteur sait que la mémoire ne sera utilisée uniquement que lorsque le cache sera activé sur le processeur. De même, si les interfaces MCH ne sont pas requises, elles peuvent être désactivées. Lorsque le bus OPB ainsi qu'un ou plusieurs canaux MCH sont activés, un mécanisme d'arbitration à priorités fixes est utilisé afin de déterminer la prochaine transaction à exécuter.

3.2. Environnement de simulation SPACE

L'environnement SPACE dispose à priori d'un bon nombre de modèles de composants disponibles sur la plateforme Xilinx. Les bus LMB, OPB et FSL sont modélisés ainsi qu'un grand nombre de périphériques comme la mémoire BRAM et SDRAM, le port sériel UART, minuteries, etc. Le MicroBlaze est également disponible sous la forme d'un simulateur d'instructions (ISS) afin de simuler l'exécution de logiciels.

Dans cette section, l'état des composants MicroBlaze et OPB SDRAM sera discuté et comparé à l'implémentation matérielle de Xilinx. La section suivante traitera des modifications apportées à la plateforme afin de concilier ces différences au niveau de la simulation dans SPACE.

3.2.1. ISS du MicroBlaze

L'ISS du MicroBlaze dans SPACE est complètement fonctionnel au niveau programmeur, c'est à dire qu'une application s'exécutant sur cet ISS exhibera le même comportement que le MicroBlaze de Xilinx. Par contre, le modèle ne rencontre pas les contraintes temporelles du MicroBlaze. En d'autres mots, un programme s'exécutant sur l'ISS ne s'exécutera pas en un même nombre de cycles que le MicroBlaze original.

Le modèle (Figure 3.4) est composé d'un processus SystemC, de deux modules d'accès mémoire, d'un tableau modélisant les registres généraux et les registres spéciaux, des bus LMB et OPB, de 8 liens FSL et des signaux d'horloge, reset et interruption. Des fonctions de débogage sont également disponibles.

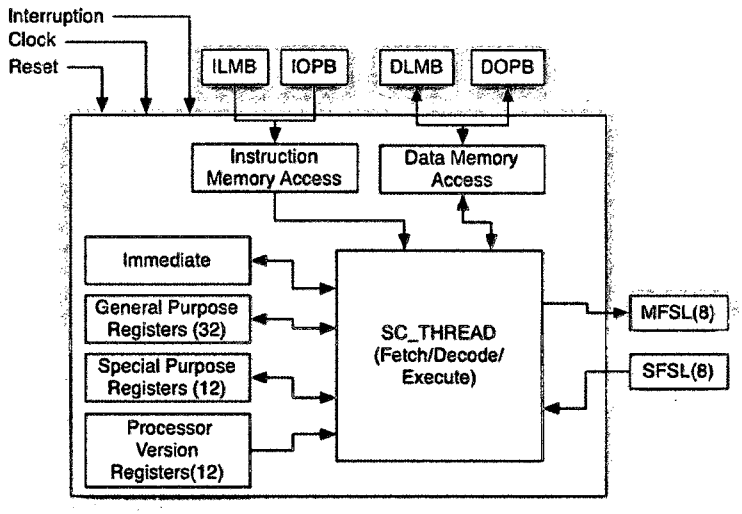

Figure 3.4 Architecture de l'ISS initial du MicroBlaze

Le processus SystemC (**sc_thread**) est une boucle infinie qui effectue les opérations des étages *Fetch*, *Decode* et *Execute*. Puisque le processeur est simulé dans un seul processus SystemC, les accès à la mémoire sont effectués séquentiellement et, par conséquent, ne peuvent tirer avantage des bus d'instructions et de données qui permettent des accès concurrents.

Une étude de la vitesse de simulation et de la précision temporelle du modèle a été effectuée dans [57]. La conclusion des auteurs est que bien que ce modèle soit rapide à simuler (jusqu'à 442 KHz), il est clair qu'il n'est pas approprié pour une simulation qui demande de la précision. En effet, le modèle présente un écart du nombre de cycles simulés allant jusqu'à 151% comparativement au MicroBlaze de Xilinx. De plus, ce nombre varie grandement selon l'application exécutée et la configuration de la mémoire.

Ce modèle est donc adapté pour une simulation purement fonctionnelle mais n'est pas approprié pour une simulation ayant pour but d'estimer la performance du système. Il est donc nécessaire d'implémenter un modèle plus précis.

3.2.2. Contrôleur OPB SDRAM

Le composant OPB SDRAM disponible sur SPACE ne modélise que la mémoire SDRAM et n'inclut aucun support pour les interfaces MCH. Par conséquent, le protocole XCL n'est pas supporté.

3.3. Mise à jour des composantes

À cause des nombreuses différences entre les versions du MicroBlaze et du contrôleur SDRAM disponibles dans SPACE et les spécifications de Xilinx, il est nécessaire de mettre à jour ces composantes afin de permettre une évaluation plus précise de la performance attendue après l'optimisation de la hiérarchie mémoire par l'outil développé. Le MicroBlaze et le contrôleur SDRAM sont les seuls composants touchés puisque les bus et autres périphériques possèdent déjà une implémentation respectant les temps cycliques (*cycle true)* et possèdent également les fonctionnalités requises.

3.3.1. Implémentation du lien XCL

Avant de pouvoir implémenter les différentes fonctionnalités reliées au cache dans le MicroBlaze et le contrôleur mémoire, un modèle du lien XCL est nécessaire. En plus de respecter le protocole XCL, le modèle doit être *cycle true* et rapide. Afin de maximiser la vitesse de simulation, le modèle du lien XCL est fait au niveau TLM et ne modélise donc pas les signaux mais fait abstraction de ceux-ci avec des méthodes et des type de données de haut niveau. Puisque le protocole XCL est relativement simple, il est aisé de réaliser un modèle *cycle true* au niveau TLM.

Ce modèle fait donc abstraction des 2 liens FSL présents dans l'implémentation matérielle. De plus, le protocole XCL est émulé plutôt que réellement simulé tout en assurant un comportement identique. Ceci veut dire que, par exemple, pour une lecture d'une ligne de cache, un seul appel de fonction est nécessaire afin de retourner la ligne de cache, la latence due au protocole et à la mémoire externe étant reproduite à l'aide de la fonction *wait()* [58] de SystemC. De cette façon, le modèle peut s'exécuter rapidement sans avoir à modéliser le protocole et les bus FSL.

Le modèle XCL est composé de 2 classes et de deux interfaces. La classe **XCLMaster** modélise la portion maître du lien XCL qui initie les transferts. La classe **XCLSlave** implémente la portion esclave du lien, qui répond aux requêtes. L'interface **XCLSlaveIF** permet, par exemple, d'utiliser la portion esclave comme type d'un **sc_port**. Finalement, l'interface **XCLSlaveImplIF** spécifie les fonctions spécifiques à l'esclave qui doivent être implémentées par la classe utilisant un objet **XCLSlave** comme attribut.

La Figure 3.5 illustre le diagramme simplifié de classe du modèle XCL ainsi qu'un exemple d'utilisation. La classe **SampleMaster** comprend une instance de la classe **XCLMaster** et la classe SampleSlave comprend une instance de la classe **XCLSlave** et implémente l'interface **XCLSlaveImplIF**. De plus, le port **XCLIFPort** de l'instance **XCLMaster** de la classe **SampleMaster** doit être associé à l'instance **XCLSlave** de la classe **SampleSlave**.

Ceci permet de connecter la portion maître et esclave du lien XCL. Il suffit ensuite d'utiliser les différentes méthodes exposées par la classe **XCLMaster** afin de communiquer via le lien XCL.

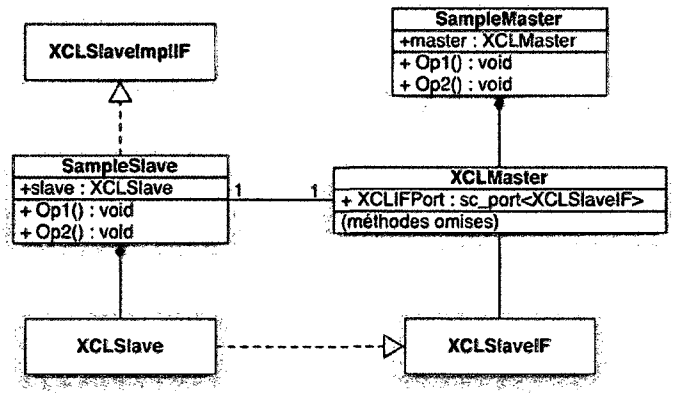

Figure 3.5 Exemple d'utilisation d'un lien XCL en simulation

3.3.2. Implémentation du MicroBlaze version 6

Le MicroBlaze est le composant présentant le plus grand nombre de divergences par rapport à la spécification. Les modifications à apporter au modèle du MicroBlaze sont les suivantes : l'implémentation des mémoires cache d'instructions et de données avec support pour les liens XCL et l'amélioration de l'exactitude (*cycle accuracy*) du modèle. Les résultats de ces modifications sont présentés dans [57].

3.3.2.1.Cache du MicroBlaze

La première modification consiste donc à implémenter le cache du MicroBlaze. Comme mentionné à la section 3.1.2, le MicroBlaze possède deux caches : instructions et données. Chacun de ces deux caches est de type *accès direct* et peut être configuré pour avoir une taille située entre 64 octets et 64 kilo-octets ainsi qu'une taille de ligne de cache de 4 ou 8 mots. De plus, le cache de données du MicroBlaze supporte l'écriture en mode *write-through.*

Le cache du MicroBlaze est modélisé dans l'unité d'accès à la mémoire nommée **uBlazeV6MMU**. Cette unité permet d'abstraire les méthodes d'accès aux données du modèle du processeur et permettra dans le futur d'implémenter la gestion de la mémoire

virtuelle disponible dans la version 7 du MicroBlaze. Dans la version initiale du modèle du MicroBlaze disponible dans SPACE, la classe **uBlazeV6MMU** ne permettait que d'accéder aux données par les bus et ne modélisait pas la mémoire cache. Afin de modéliser cette cache, deux tableaux ont été implémentés ainsi que plusieurs méthodes permettant, par exemple, de déterminer si le cache est activé au niveau logiciel ou si une adresse demandée provoque un *succès* ou un *échec* dans le cache. Plusieurs attributs ont également été rajoutés, ces attributs spécifient la configuration de la mémoire cache ainsi que d'autres attributs comme le masque de sélection de ligne de cache.

Chaque méthode d'accès à la mémoire a ensuite été modifiée afin de prendre la mémoire cache en compte. La logique de décision de la méthode utilisée diffère légèrement selon que l'accès est une lecture ou une écriture. La logique de décision pour la lecture et celle de l'écriture sont illustrées à la Figure 3.6 a) et b). Dans le cas du cache d'instructions, la logique d'écriture n'est pas nécessaire puisqu'on ne peut qu'y lire.

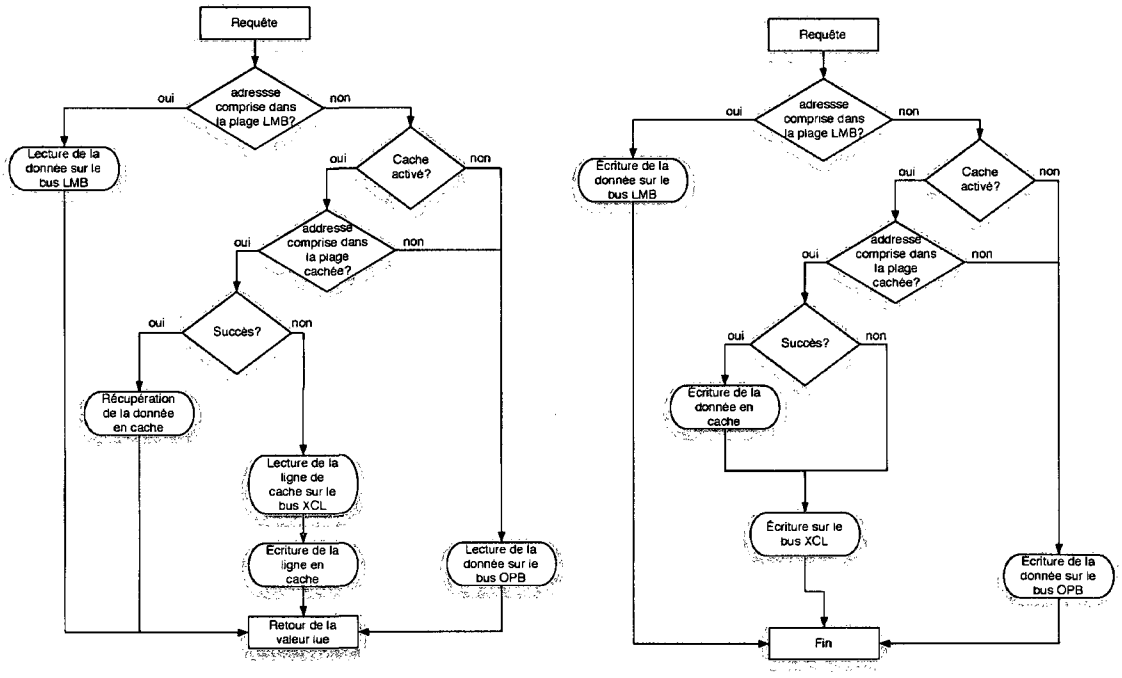

Figure 3.6 Logique de décision de la méthode d'accès

Le cache lui-même est implémenté en deux tableaux : un tableau contenant les lignes de cache (le tableau de cache) et un tableau contenant l'état de chacune de ces lignes (le tableau des *tags*). Le fonctionnement est réalisé à l'aide des méthodes **cache_enabled()**, **cache_hit()**, **cache_shouldUpdate()**, **cache_update()**. La méthode **cache_enabled()** permet de déterminer si le cache est activé ou non en effectuant la lecture du registre de statut du processeur (MSR). Le bit à lire dans ce registre est dépendant du type de cache, instructions ou données, qui est déterminé à l'instanciation de la classe **uBlazeV6MMU**. La méthode **cache_hit()** détermine si l'adresse passée en argument déclenche un succès ou un échec en accédant au tableau des *tags*. La méthode **cache_shouldUpdate()** quant à elle détermine si la ligne de cache correspondant à l'adresse peut être remplacée dans le cas d'un échec. Finalement, la méthode **cache_update()** effectue la lecture de la ligne de cache dans le cas d'un succès et remplace l'ancienne ligne de cache par la nouvelle.

La dernière étape dans l'implémentation du cache est l'ajout du support du bus XCL. Ceci est fait en ajoutant une instance de la classe **XCLMaster** à la classe **uBlazeV6MMU**. Vu la nature asynchrone des transferts XCL dans le MicroBlaze, un processus SystemC **XCLThread()** est ajouté à **uBlazeV6MMU** afin de permettre au processeur de continuer son exécution et de ne pas bloquer durant le transfert XCL. De plus, plusieurs méthodes de préparation du transfert XCL sont implémentées et sont utilisées par les méthodes de lecture et d'écriture du MMU. Lorsque le tout est intégré dans la classe **uBlazeV6MMU**, la structure de cette classe correspond donc à la Figure 3.7.

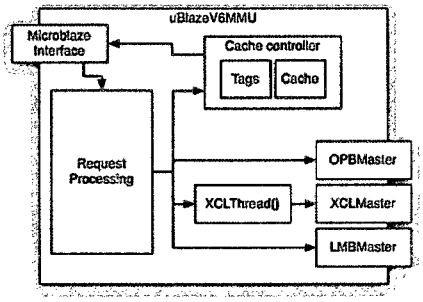

Figure 3.7 Schéma bloc du composant uBlazeV6MMU

3.3.2.2.Pipeline

Afin d'améliorer l'exactitude du modèle, la structure du pipeline ainsi que les mécanismes associés ont été implémentés à haut niveau. Pour ce faire, des mécanismes SystemC tels que des processus, méthodes, FIFO et signaux ont été utilisés pour modéliser les cinq étages et les différentes interactions entre ceux-ci. Le fait de modéliser cette architecture permet de simuler les délais et latences qui ne peuvent être simulés avec le modèle existant puisque ces délais sont dépendants de la séquence d'instructions à exécuter et donc difficilement prévisibles.

Le modèle comprend deux méthodes qui permettent de simuler les étages *Fetch* et *Decode* ainsi que trois processus qui modélisent les étages *Execute*, *Memory* et *Writeback*. Chaque étage est relié au prochain à l'aide d'une primitive **sc_fifo** qui prend le rôle de registre de pipeline (*pipeline register*). De plus, deux signaux indiquant qu'un branchement a eu lieu et si ce branchement comporte une *delay slot,* relient l'étage *Execute* aux étages *Fetch* et *Decode*. Deux signaux s'occupent du court-circuitage (*forwading*) et relient l'étage *Memory* à l'étage *Execute* et l'étage *Execute* à lui-même. La raison de ce choix sera expliquée plus loin. Finalement, afin d'assurer la cohérence des registres, un mécanisme de réservation des registres est implémenté et permet de déterminer si un registre est présentement utilisé par une instruction dans le pipeline à titre de registre de destination. La Figure 3.8 présente un schéma bloc du modèle développé.

3.3.2.3.Registres internes

Le modèle du MicroBlaze utilise plusieurs registres internes [13] afin de maintenir la cohérence des résultats et donc d'assurer le bon déroulement de l'exécution. Ces registres internes sont les registres de pipeline, les registres de court-circuitage et le registre de valeur immédiate.

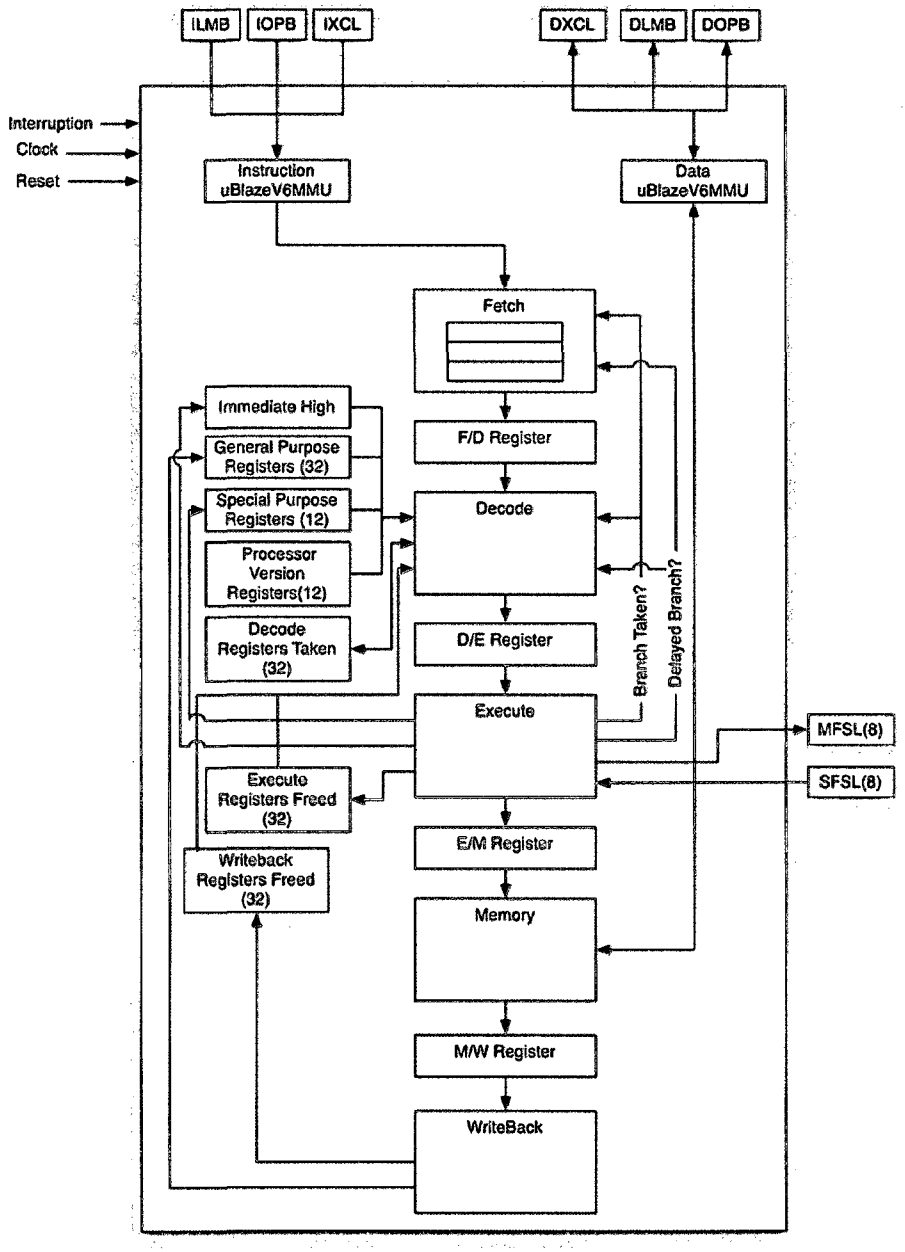

Figure 3.8 Architecture de l'ISS modifié du MicroBlaze

Les registres de pipeline permettent de transférer l'information d'un étage à un autre. En ce sens, ces registres doivent contenir une certaine quantité d'information afin d'assurer l'exécution correcte de l'instruction. Chacun de ces registres contient donc l'instruction elle-même, l'adresse de l'instruction (PC), le type d'opération à effectuer, la liste des

registres utilisés en source et en destination, la valeur de ces registres, la valeur de l'immédiat, la taille du transfert (s'il s'agit d'un accès mémoire) et, finalement, de l'information sur le court-circuitage. Ceci permet à chaque étage de savoir exactement quel travail effectuer pour cette instruction. La structure de ces registres est présentée au Tableau 3.1.

Tableau 3.1 Structure des registres de pipeline

Nom	Type	Description
instr	unsigned long (32 bits)	Instruction encodée
Pc	unsigned long (32 bits)	Adresse de l'instruction (PC)
type	eInstrType	Type d'instruction : ALU, contrôle, immédiat, no-op, lecture mémoire ou écriture mémoire.
subtype	unsigned char (8 bits)	Liste des registres source et destination utilisés (Ra, Rb, Rd, immédiat, registre spécial)
datasize	unsigned short (16 bits)	Taille du transfert (si lecture ou écriture)
Ra	unsigned char (8 bits)	Registre source Ra
Rb	unsigned char (8 bits)	Registre source Rb
Rd	unsigned char (8 bits)	Registre destination Rd
Ra_val	long (32 bits)	Valeur du registre Ra
Rb_val	long (32 bits)	Valeur du registre Rb
Rd_val	long (32 bits)	Valeur du registre Rd
immediate	long (32 bits)	Valeur immédiate
useForwardingRa	bool	Si le court-circuitage doit être utilisé pour le registre Ra.
useForwardingRb	bool	Si le court-circuitage doit être utilisé pour le registre Rb.
useForwardingRd	bool	Si le court-circuitage doit être utilisé pour le registre Rd.
regTakenDecodeRa	long	Nombre de fois que le registre Ra a été réservé par l'étage Decode à l'instant où l'instruction était décodée.
regTakenDecodeRb	long	Nombre de fois que le registre Rb a été réservé par l'étage Decode à l'instant où l'instruction était décodée.
regTakenDecodeRd	long	Nombre de fois que le registre Rd a été réservé par l'étage Decode à l'instant où l'instruction était décodée.
valid	bool	Si les valeurs dans le registre de pipeline sont valides.

En plus des registres de pipeline et des banques de registres généraux et spéciaux, le modèle comprend trois banques de registres de statut dédiées à la gestion de la réservation des registres généraux et qui entrent en compte lors de l'utilisation du court-circuitage. Ces banques de registres de statut permettent de déterminer quels registres

généraux sont utilisés par une instruction en cours d'exécution comme registres de destination et donc pour lesquels la valeur sera modifiée par l'étage *Writeback*. Ces banques de registres permettent donc de respecter la dépendance des données. De plus, ces registres sont internes au modèle et, par conséquent, ne sont pas accessibles par le programme en cours d'exécution sur le MicroBlaze. La première banque nommée **Decode Registers Taken** est modifiée par l'étage *Decode* à chaque fois qu'un registre général doit être réservé. La seconde banque, **Execute Registers Freed**, est modifiée lorsque l'étage *Execute* doit supprimer une instruction à cause d'un branchement (*flush*), le registre de destination réservé par l'instruction supprimée doit donc être libéré. Finalement, la dernière banque, **Writeback Registers Freed**, est modifiée lorsque l'étage *Writeback* libère un registre de destination après l'avoir modifié. L'étage *Decode* utilise alors ces trois banques afin de déterminer si le registre de destination requis par l'instruction décodée est déjà réservé et donc que le mécanisme de court-circuitage doit être utilisé ou l'exécution de l'instruction doit être retardée en attendant que le registre se libère.

Il existe un dernier registre additionnel interne au modèle qui permet de garder la portion supérieure de la valeur immédiate en mémoire. Cette portion (16 bits supérieur) est modifiée par l'instruction IMM lorsque celle-ci est exécutée par l'étage *Execute*. En effet, puisque le processeur ne supporte que des immédiats de 16 bits, le MicroBlaze propose l'instruction IMM qui permet de spécifier les 16 bits supérieurs de la valeur immédiate et de compléter les 16 bits inférieurs avec la valeur immédiate de l'instruction suivante. Ceci permet donc d'obtenir une valeur immédiate de 32 bits en deux instructions plutôt que d'exécuter des opérations de décalage de bits afin de reconstruire le mot de 32 bits et d'utiliser un registre intermédiaire.

3.3.2.4.Étages du pipeline

Comme il fut mentionné plus haut, le modèle utilise deux méthodes SystemC pour les étages *Fetch* et *Decode* et trois processus SystemC pour les étages *Execute*, *Memory* et *Writeback*. Chacun de ces étages exécute une tâche bien précise qui permet le bon fonctionnement du pipeline.

L'étage *Fetch* a pour mission de récupérer les instructions de la mémoire d'instructions. Pour ce faire, une méthode SystemC nommée **fetchStage** envoie le PC courant à un processus nommé **instructionFetchTask** qui se charge de récupérer l'instruction associée au PC de façon asynchrone. Pendant ce temps, **fetchStage** peut répondre aux signaux de branchement provenant de l'étage **Execute**. Si un branchement survient, **fetchStage** supprimera l'instruction récupérée si nécessaire et déclenchera à nouveau le processus **instructionFetchTask** afin de récupérer la nouvelle instruction. Lorsque l'instruction récupérée est valide, **fetchStage** relaie l'instruction à l'étage suivant par l'entremise du registre de pipeline F/D et démarre le processus de récupération de l'instruction suivante en incrémentant le PC. Par contre, il est possible que l'étage *Decode* soit occupé et qu'une instruction non traitée soit déjà présente dans le registre de pipeline. Dans ce cas, **fetchStage** stocke l'instruction dans un tampon simple et récupère ensuite l'instruction suivante. Les instructions présentes dans le tampon seront alors écrites une à une dans le registre F/D lorsque l'étage *Decode* sera libre à nouveau.

L'étage suivant est l'étage de décodage qui permet de récupérer la valeur des registres source et de déterminer le type de l'instruction qui servira aux étages suivant à décider des opérations à effectuer. La méthode SystemC **decodeStage** lit d'abord le contenu du registre de pipeline F/D et détermine le type de l'instruction, son sous-type et la taille du transfert. Ensuite, la valeur des registres de source est récupérée. Si un des registres n'est pas disponible, le mécanisme de court-circuitage est marqué comme nécessaire pour l'obtention de cette valeur. L'instruction décodée est alors écrite dans le registre de

pipeline D/E si ce dernier est libre, sinon **decodeStage** doit attendre au prochain cycle d'horloge. Si un branchement survient durant le décodage de l'instruction ou durant l'attente de libération du registre de pipeline D/E, l'instruction décodée peut être supprimée. Lorsque l'instruction a bien été écrite dans le registre de pipeline et que celle-ci utilise un registre de destination, ce registre sera alors marqué comme étant utilisé dans la banque de registres de statut **Decode Registers Taken**.

L'étage *Execute,* quant à lui, se charge d'exécuter les instructions qui ne requièrent aucun accès mémoire. Ceci comprend les instructions de type ALU (opérations mathématiques et logiques), les instructions de contrôle (branchements), l'instruction IMM ainsi que les instructions qui accèdent aux bus FSL. Le processus **executeStage** détermine en premier lieu s'il y a eu une interruption ou une exception depuis sa dernière exécution. Si oui, il déclenche un branchement vers le vecteur d'interruption approprié. Si aucune interruption ou exception n'est survenue, le processus lit le registre D/E et détermine alors si un branchement est survenu durant sa dernière exécution et si l'instruction lue doit être supprimée ou non. Si l'instruction ne doit pas être supprimée, le processus détermine si elle doit être exécutée selon son type. Il récupère alors les valeurs manquantes en utilisant le court-circuitage en provenance de l'étage d'exécution ou d'accès mémoire ou en attendant que le registre concerné soit libéré par l'étage de *Writeback.* Cette opération est effectuée dans cet étage afin de cacher une latence d'un cycle induite par l'utilisation d'un processus SystemC synchrone. Lorsque toutes les valeurs ont été récupérées, l'instruction peut alors être exécutée. Si l'instruction est une instruction multi-cycles, un *wait()* est utilisé afin de bloquer l'exécution de cet étage durant le nombre de cycles requis. Également, si l'instruction utilise une valeur immédiate, cette valeur immédiate est complétée avec les 16 bits supérieurs si l'instruction était précédée de l'instruction IMM. Après l'exécution, si l'instruction utilise un registre de destination, sa nouvelle valeur est écrite dans le signal de court-circuitage d'exécution. L'instruction est finalement écrite dans le registre de pipeline

E/M pour l'envoyer à l'étage d'accès mémoire. Cette écriture est bloquante si l'étage suivant n'est pas prêt.

L'étage *Memory* s'occupe d'effectuer les lectures et écritures en mémoire. Le processus **memoryStage** lit d'abord le registre de pipeline E/M et effectue l'opération appropriée selon que l'instruction soit une lecture ou une écriture et selon la taille de l'accès à effectuer (mot, demi-mot, octet). Si l'instruction est une instruction de type lecture et donc qui utilise un registre de destination, la valeur lue sera écrite dans le signal de court-circuitage de mémoire à destination de l'étage *Execute*. Ensuite, l'instruction sera dirigée vers l'étage suivant, l'étage *Writeback*.

L'étage final est l'étage **Writeback**. Cet étage n'a qu'une fonction : écrire la valeur finale calculée ou lue dans le bon registre général de destination. Lorsque cette valeur est écrite, le registre de destination est libéré (en incrémentant la bonne valeur dans la banque **Writeback Registers Freed**) et l'instruction termine donc sa course dans le pipeline. Évidemment, si l'instruction n'utilise pas de registre de destination, l'étage n'effectue aucun traitement.

3.3.3. Implémentation du contrôleur MCH OPB SDRAM

Les seules modifications majeures nécessaires au contrôleur SDRAM sont l'ajout des interfaces XCL au modèle ainsi que l'ajout d'un arbitre. Ceci est effectué en ajoutant quatre attributs de type **XCLSlave** puisque la spécification prévoit un maximum de quatre ports MCH/XCL par contrôleur. De plus, il est nécessaire d'implémenter l'interface **XCLSlaveImplIF** dans le contrôleur afin de permettre aux esclaves XCL d'accéder au modèle de mémoire directement. L'arbitre est implémenté simplement sous forme d'un mutex qui régit l'accès au modèle de la mémoire. Lorsque ces changements sont effectués, les connexions entre le MicroBlaze et le contrôleur MCH OPB SDRAM correspondront alors à la Figure 3.9 .

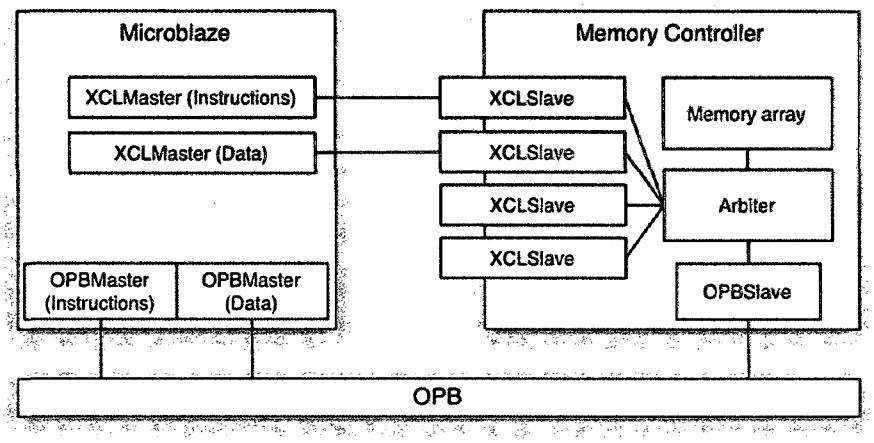

Figure 3.9 Connexions entre le MicroBlaze et le contrôleur MCH OPB SDRAM

3.4. Outil de traçage d'exécution

L'outil d'optimisation de la hiérarchie mémoire nécessite la présence d'une trace d'exécution afin de pouvoir établir le profil de façon non-intrusive, déterminer le contenu de la mémoire locale et enfin, simuler les configurations de cache. Un outil permettant de générer cette trace est donc nécessaire. Le traceur en question doit générer une trace d'exécution permettant de générer le profil et de simuler les configurations de cache. La génération de cette trace doit impliquer un minimum de changements dans le modèle du processeur et supporter les instructions et les données. Finalement, la trace générée doit inclure l'information nécessaire à l'outil d'optimisation de la hiérarchie mémoire afin d'estimer la performance de la version optimisée.

Deux types de trace sont donc utilisées : un fichier de trace dédié à la génération du profil (trace de type mémoire) et un fichier de trace dédié à la simulation des configurations de cache (trace de type cache). Chacun de ces types de fichier utilise la même en-tête de fichier qui indique les différents paramètres de la simulation. Lors de longues simulations et dû à l'importante quantité d'information contenue dans ces traces, le volume de ces traces devient très important, de l'ordre de centaines de méga-octets. Il devient nécessaire de compresser ces traces afin de diminuer la taille des fichiers générés.

Un algorithme spécialisé de compression est donc utilisé pour compresser chacun de ces types de trace. La classe résultante permettant d'effectuer ces traces se nomme **uBlazeV6Tracer**.

3.4.1. Interface de la classe uBlazeV6Tracer

La classe **uBlazeV6Tracer** est relativement simple à utiliser et un nombre restreint de méthodes doivent être utilisées afin de générer une trace complète. Les méthodes se divisent en trois catégories principales: méthodes de collection des entrées de trace, méthodes d'ajout d'un accès et fonctions accesseurs. Les méthodes de collection des entrées de trace permettent de recueillir la trace elle-même en écrivant dans le fichier l'adresse, la taille, le type et la latence de chaque accès. Les méthodes d'ajout d'un accès permettent à la fois d'incrémenter par un le nombre d'accès en lecture ou en écriture comptabilisés et d'ajouter une latence à la latence totale de lecture ou d'écriture. Ceci permet de calculer la latence moyenne d'accès en lecture ou en écriture à la mémoire externe. Finalement, les fonctions accesseurs permettent de récupérer les données de latence totale, de latence moyenne et le nombre total d'accès effectués. Deux dernières méthodes permettent d'assigner le nombre total de cycles simulés (qui peut être différent du nombre total de cycles recueillis par la trace) et finalement de terminer le recueil de la trace et d'effectuer les opérations de nettoyage.

3.4.2. Fichier de trace

Le fichier de trace est séparé en deux portions : l'en-tête et les entrées de trace. L'en-tête contient l'information nécessaire à l'outil d'optimisation qui lui permettra d'estimer la performance. Les entrées de trace permettent, selon le type de trace, d'effectuer le profil ou la simulation de cache. Le Tableau 3.2 expose l'information présente dans l'en-tête tandis que le Tableau 3.3 démontre l'information présente dans chaque entrée de trace pour la trace de type mémoire ou de type cache.

Tableau 3.2 En-tête de fichier de trace

Nom	Type	Description
cachedAddressLow	unsigned long (32 bits)	Adresse de base cachée
cachedRange	unsigned long (32 bits)	Étendue des adresses cachées
totalExecutionCycles	long long (64 bits)	Nombre total de cycles exécutés en simulation
addressingBits	unsigned char (8 bits)	Latence moyenne d'accès en lecture à la mémoire principale
memoryWriteLatency	unsigned char (8 bits)	Latence moyenne d'accès en écriture à la mémoire principale
traceMode	unsigned char (8 bits)	Mode de trace (Mémoire ou Cache)
cacheReadLatency	unsigned char (8 bits)	Latence d'accès en lecture à la cache pour un succès
cacheWriteLatency	unsigned char (8 bits)	Latence d'accès en écriture à la cache pour un échec
memoryReadDedicatedLatency	unsigned char (8 bits)	Latence moyenne d'accès en lecture pour la connexion dédiée à la mémoire principale
memoryWriteDedicatedLatency	unsigned char (8 bits)	Latence moyenne d'accès en écriture pour la connexion dédiée à la mémoire principale
totalAccesses	long long (64 bits)	Nombre total d'accès effectués lors de la simulation

Tableau 3.3 Entrées du fichier de trace

Trace	Nom	Type	Description
Mémoire	address	unsigned long (32 bits) ou unsigned char (8 bits)	Adresse accédée
	mode	unsigned char (8 bits)	Mode d'accès (taille, lecture/écriture, information de compression).
Cache	address	unsigned long (32 bits)	Adresse accédée/Ligne de cache accédée
	mode	unsigned char (8 bits)	Mode d'accès (taille, lecture/écriture)
	count	unsigned char (8 bits)	Accès consécutifs à la même ligne de cache

Afin de réduire la taille de la trace, un algorithme de compression est utilisé pour les deux types de trace. L'algorithme de compression de la trace de mémoire tire avantage du fait que les accès, tant dans une trace d'instruction que dans une trace de données sont généralement faits à des adresses rapprochées. Cela permet d'écrire uniquement un déplacement d'un octet au lieu d'écrire l'adresse entière de 4 octets pour chaque accès. L'algorithme est décrit à la Figure 3.10.

```
uint32b previousAddress

writeMemoryAccess(uint32b address, uint8b latency) {
   int8b offset

   offset = address - previousAddress

   if( | offset | < 128 )
      writeOffset(offset, latency)
   else
      writeAddress(address, latency)
   end if
}
```

Figure 3.10 Algorithme de compression de la trace de mémoire

Pour chaque accès, le déplacement par rapport à l'accès précédent est calculé. Si ce déplacement est situé à l'intérieur de la plage -127 à 127, seul ce déplacement de taille d'un octet sera inscrit dans le fichier de trace. L'entrée aura une taille totale de 2 octets. Lorsque le déplacement dépasse la plage, l'adresse complète de 4 octets sera alors écrite et l'entrée aura alors une taille totale de 5 octets. Trois octets sont donc économisés, ce qui correspond à 60% de la taille non compressée.

L'algorithme de compression pour la trace de cache est légèrement plus complexe que celui de compression de trace de mémoire. En effet, puisque la taille de ligne de cache minimale supportée par le MicroBlaze est de 4 mots, l'algorithme de compression utilise cette taille de ligne de cache afin de réduire l'information à écrire dans la trace. Cet algorithme est décrit à la Figure 3.11. Chaque adresse accédée est alignée selon une ligne de cache. Pour ce faire, un ET-logique est appliqué avec le masque 0xFFFFFFF0 sur l'adresse. Pour chaque accès, l'algorithme détermine alors si cet accès est situé dans la même ligne de cache que l'accès précédent et incrémente le compteur d'accès.

Lorsque l'adresse accédée ne correspond plus à la ligne de cache, l'entrée précédente est écrite dans le fichier de trace avec le nombre d'accès effectués sur la ligne de cache. Ceci permet d'économiser beaucoup d'espace lorsque, par exemple, une petite boucle est exécutée.

```
uint32b previousCacheLine
uint8b  previousMode
uint8b  consecutiveAccesses

writeCacheAccess(uint32b address, uint8b mode, uint8b latency) {
   uint32b cacheLine

   cacheLine = address & 0xFFFFFFF0
   if( (previousCacheLine == cacheLine) and
       (previousMode == mode) )
      if( ++consecutiveAccesses == 255 )
         write(cacheLine, mode, consecutiveAccesses)
         consecutiveAccesses = 0
      end if
   else
      if( consecutiveAccesses != 0 )
         write(cacheLine, mode, consecutiveAccesses)
         consecutiveAccesses = 1
         previousCacheLine = cacheLine
         previousMode = mode
      end if
   end if
}
```

Figure 3.11 Algorithme de compression de la trace de cache

3.4.3. Intégration du traceur dans l'ISS du MicroBlaze

L'intégration du traceur à l'ISS du MicroBlaze ne requiert que quelques modifications au code. La Figure 3.12 présente le pseudo-code de ces modifications. Le traceur doit premièrement être instancié pour les instructions et les données, ce qui peut se faire dans le constructeur de l'ISS. Ensuite, les instructions et données doivent être tracées. Le traçage des données est simple, il suffit de calculer le temps nécessaire à l'accès pour déterminer la latence du transfert et d'appeler la méthode trace*() correspondant au type de transfert (lecture ou écriture et taille du transfert). Ceci se fait dans l'étage *Memory*. Le traçage des instructions quant à lui est plus ardu. Puisque certaines instructions ne sont pas exécutées, elles ne doivent pas être écrites dans la trace afin de ne pas influencer le profil. Par contre, leur latence d'accès à la mémoire doit être comptabilisée puisqu'elle peut affecter le temps moyen d'accès. Il est donc nécessaire d'appeler la méthode d'ajout d'un accès en lecture afin de calculer la latence d'accès moyenne. Ceci est fait dans l'étage *Fetch* puisque c'est dans cet étage que les instructions qui seront

potentiellement exécutées sont récupérées. Le recueil de la trace elle-même se fait à la fin de l'étage *Execute* lors de l'exécution des instructions. Cela assure que seules les instructions réellement exécutées seront présentes dans la trace.

La méthode pour déterminer la latence des transferts est simple : il faut d'abord récupérer le temps de simulation avant le transfert et le temps de simulation après le transfert et soustraire les deux temps. Ensuite, il suffit de diviser le temps obtenu par la période d'horloge, ce qui donne le nombre de cycles de latence du point de vue du processeur.

Avec la présence de cet outil de traçage, il est donc possible de générer des fichiers de trace de taille réduite qui contiennent toutes les données nécessaires aux algorithmes d'optimisation. De plus, il est possible d'intégrer le traceur à d'autres ISS puisque son interface est plutôt générique. Finalement, puisque le format de fichier est simple et connu, il est possible de créer un autre mécanisme de traçage compatible avec celui présenté ici sans briser les applications dépendantes des fichiers de trace.

```
fetchStage(instructionAddress) {
   startSimulationTime = getSimulationTime()
   fetchNextInstruction(instructionAdddress)
   endSimulationTime = getSimulationTime()
   latency = endSimulationTime - startSimulationTime
   writeInstructionTraceLatency(latency)
}


executeStage(instructionAddress, instruction) {
   executeInstruction(instruction)
   writeInstructionTraceEntry(instructionAddress, read)
}
memoryStage(dataAddress, accessMode, data) {
   if( accessMode == read )
      startSimulationTime = getSimulationTime()
      data = readData()
      endSimulationTime = getSimulationTime()

      latency = endSimulationTime - startSimulationTime
      writeDataTraceLatency(latency)
      writeDataTraceEntry(dataAddress, read)
```

```
   else
       startSimulationTime = getSimulationTime()
       writeData(data)
       endSimulationTime = getSimulationTime()

       latency = endSimulationTime - startSimulationTime
       writeDataTraceLatency(latency)
       writeDataTraceEntry(dataAddress, write)
    end if
}
```

Figure 3.12 Ajout de la trace dans l'ISS

Grâce aux concepts présentés au chapitre 2 ainsi qu'aux modifications à la plateforme de simulation décrites dans ce chapitre, tous les instruments sont en place afin de permettre l'implémentation de l'outil d'optimisation de la hiérarchie mémoire. Dans le chapitre suivant, l'outil MemoryOptimizer est décrit en détails. Cet outil permet à la fois d'automatiser le partitionnement de mémoire et d'effectuer l'optimisation des mémoires caches.

CHAPITRE 4

MEMORYOPTIMIZER : OPTIMISATION DE LA HIÉRARCHIE MÉMOIRE

L'objectif de MemoryOptimizer, l'outil de partitionnement de mémoire et d'optimisation de la mémoire cache développé ici, est de tenter d'accélérer l'exécution du programme exécuté sur un processeur embarqué, tout en minimisant l'utilisation de la mémoire intégrée au SoC. L'accomplissement de cette tâche passe par quatre phases majeures : génération de la trace d'exécution par l'entremise d'une simulation, partitionnement de la mémoire, optimisation de la mémoire cache et finalement, la simulation de la configuration finale. De ces phases, MemoryOptimizer automatise les étapes de partitionnement et d'optimisation. L'outil est relativement portable puisqu'il est codé en Java mais dépend de certains exécutables de Xilinx et est donc limité aux plateformes sur lesquelles les outils de Xilinx sont disponibles.

Dans ce chapitre, MemoryOptimizer est présenté en détails. Le flot d'exécution de l'outil est d'abord abordé afin de donner une idée générale du fonctionnement. Ensuite, la portion de partitionnement de mémoire est décrite. Suivra finalement l'optimisation de la configuration de la mémoire cache. L'interface graphique, quant à elle, est décrite à l'annexe B.

4.1. Flot d'exécution

Le flot d'exécution (Figure 4.1 Flot d'exécution de MemoryOptimizer) débute par la simulation d'un système exécutant un programme sur l'ISS du MicroBlaze afin de récolter les traces d'exécution de mémoire. Ensuite, à l'aide du fichier exécutable ELF, des deux fichiers de trace et de certains paramètres d'entrée, la phase de partitionnement

peut effectuer son travail. Lorsque le partitionnement de mémoire est terminé, un fichier de type *linker script* (script de l'éditeur de liens) est généré. Ce fichier permet de recompiler l'exécutable avec la nouvelle configuration de la mémoire. Si l'utilisateur a indiqué à l'outil qu'il désire également effectuer l'optimisation de la mémoire cache, deux fichiers de trace de cache sont également générés. Ces fichiers contiennent uniquement les références aux instructions et données qui sont toujours en mémoire, ce qui permet donc de représenter les accès au nouveau contenu de la mémoire externe sans avoir à recompiler et à simuler de nouveau.

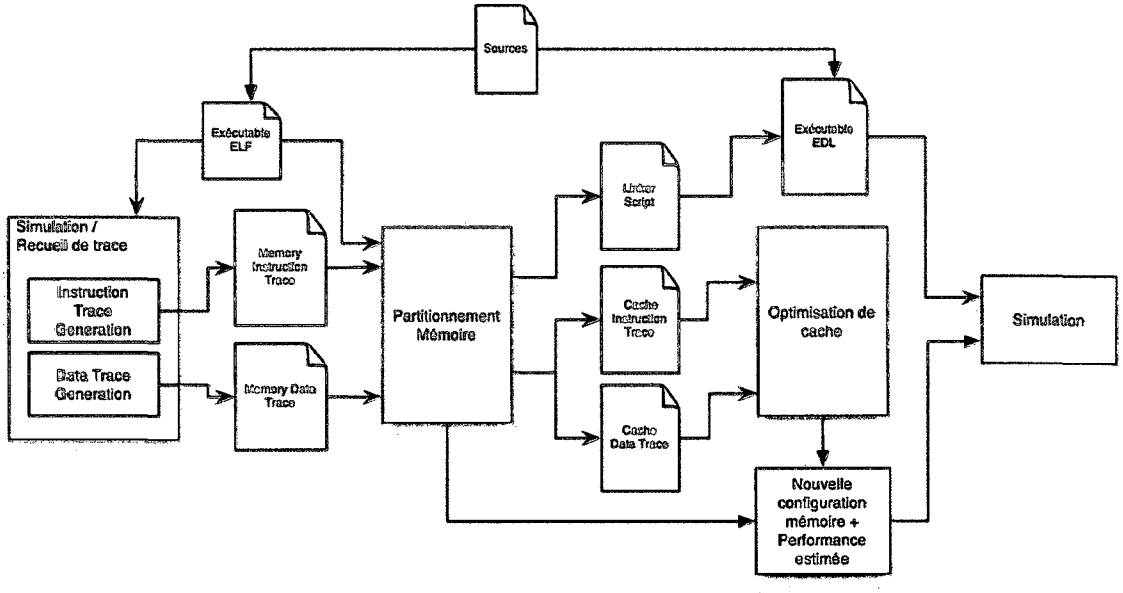

Figure 4.1 Flot d'exécution de MemoryOptimizer

Avec ces deux fichiers de trace de cache, la phase d'optimisation du cache peut procéder et déterminer la meilleure configuration en termes de ressources et de performance. L'outil fournit également des estimations de la performance qui doit être attendue avec la nouvelle configuration suggérée. Enfin, l'utilisateur doit recompiler son fichier exécutable à l'aide du script de l'éditeur de liens et simuler de nouveau son système afin de vérifier la performance finale de ce dernier.

Les phases de partitionnement de mémoire et d'optimisation de la mémoire cache constituent en eux-mêmes des outils distincts et en ce sens, ces deux phases sont indépendantes. À l'aide de l'outil MemoryOptimizer, il est donc possible de n'effectuer que l'une ou l'autre de ces phases. Par exemple, si l'utilisateur ne désire qu'optimiser la configuration des mémoires cache sans utiliser de mémoire locale, l'étape de partitionnement de la mémoire n'a pas à être exécutée. Par contre, cela doit être déterminé lors de la phase de simulation et de récolte des traces puisque l'optimisation de mémoire cache utilise un format de fichier de trace différent de celui ciblant le partitionnement de mémoire.

Il est important de souligner que MemoryOptimizer assume que le système à analyser respecte une certaine configuration. Premièrement, il assume que le processeur supporte une mémoire locale de taille arbitraire avec une connexion directe au processeur, ce qui est le cas du MicroBlaze et du PowerPC qui sont utilisés sur la plateforme Xilinx. De plus, lors de la phase d'optimisation de la mémoire cache, MemoryOptimizer assume que le processeur supporte le même système de cache que le MicroBlaze. De plus, il est assumé que la mémoire locale affiche une performance supérieure à la mémoire cache en terme de latence d'accès et donc est favorisée. Finalement, MemoryOptimizer assume que le programme exécuté était situé complètement en mémoire externe lors de la génération de la trace et que le fichier ELF n'a pas été modifié depuis. Si une des ces conditions n'est pas remplie, l'outil peut afficher un comportement erratique.

4.2. Partitionnement de mémoire

L'objectif du partitionnement de mémoire est d'accélérer l'exécution du programme en relocalisant les symboles (fonctions, variables) présentant la plus grande utilisation du temps d'exécution dans la mémoire locale du processeur. Ceci permet de grandement diminuer le temps d'accès à ces symboles en plus de réduire la pression sur la mémoire externe et le bus système.

Cette section s'attarde donc à la description du fonctionnement du partitionnement de mémoire. Un survol de l'architecture de ce sous-système sera d'abord présenté et sera suivi d'une description plus détaillée de chacune des étapes impliquées dans le partitionnement.

4.2.1. Architecture

La méthode de partitionnement de mémoire implantée dans MemoryOptimizer est issue des méthodologies statiques présentées à la section 2.3.1. En effet, dans un système embarqué, le programme exécuté effectue, en général, les mêmes tâches à répétition. Il n'est donc pas nécessaire d'implémenter un partitionnement de mémoire dynamique puisque le contenu de la mémoire locale ne changera que rarement. De plus, le MicroBlaze utilisé dans ce projet ne supporte pas de MMU et donc, certaines techniques de partitionnement dynamique ne sont pas possibles sur ce processeur.

Le partitionnement de mémoire (Figure 4.2) nécessite trois fichiers d'entrée : le fichier exécutable ELF utilisé lors du recueil des traces, la trace des instructions et finalement la trace des données. En sortie, de un à trois fichiers seront générés selon les options choisies. De ces fichiers, seul le script de l'éditeur de liens est toujours généré. Ensuite, si l'utilisateur désire poursuivre l'analyse avec l'optimisation de mémoire cache, les traces de cache d'instruction et de données sont générées.

L'analyse du fichier ELF constitue la première étape du partitionnement de mémoire. Lors de cette phase, les différentes sections et symboles présents dans le fichier ELF sont examinés. La taille et l'adresse de chaque section et symbole sont déterminées et chaque symbole est associé à la section spécifiée dans le fichier exécutable. De même, chaque section se voit attribué la liste des symboles lui appartenant. Une hiérarchie de

symboles et sections est ainsi créée et permettra aux phases subséquentes de disposer de toute l'information nécessaire sur l'exécutable.

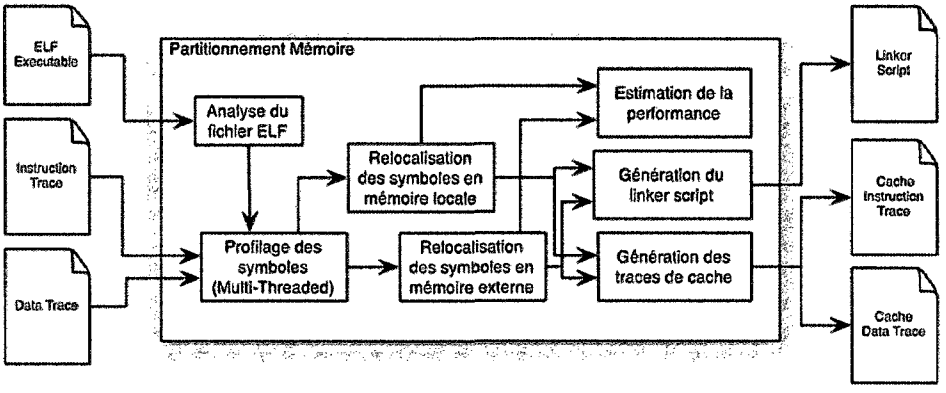

Figure 4.2 Diagramme de flux de données du partitionnement de mémoire

La phase suivante est la phase de profilage. Lors de cette phase, les deux fichiers de trace sont lus en entier et pour chaque accès lu, l'algorithme détermine à quel symbole l'accès fait référence et incrémente le compteur d'accès en lecture ou en écriture. Dans le cas de la trace de données, une opération supplémentaire est exécutée. Pour chaque accès pour lequel le symbole associé correspond à la pile ou au tas, l'algorithme détermine l'utilisation de ce segment de mémoire. Ceci permet d'évaluer l'utilisation maximale faite de la pile ou du tas et permettra par la suite à MemoryOptimizer de recommander à l'utilisateur une taille pour chacun de ces segments de mémoire.

L'algorithme de profilage analyse les traces d'instructions et de données séquentiellement. Par contre, puisque la correspondance adresse à symbole demande beaucoup de temps de calcul, le profilage est séparé en plusieurs fils d'exécutions. Ceci permet de tirer avantage des ordinateurs multi processeurs et des processeurs multi cœurs.

La relocalisation des symboles dans la mémoire locale constitue l'étape subséquente. La relocalisation s'effectue selon la règle empirique de localité 90/10 [13]. Cette règle

stipule que 90% des instructions exécutées constituent 10% du code. Il est donc possible de déplacer une petite partie du programme en mémoire locale et ainsi obtenir une bonne accélération. L'algorithme tente donc de remplir une mémoire locale en relocalisant les symboles qui occupent jusqu'à 90% du temps d'accès à la mémoire. Afin de limiter la taille de la mémoire locale, l'utilisateur peut spécifier une taille maximale de mémoire à utiliser. L'algorithme de relocalisation se terminera donc si l'une de ces deux conditions est remplie. Les symboles à relocaliser en mémoire locale sont ajoutés dans une liste et les symboles devant rester dans la mémoire externe sont ajoutés dans une seconde liste.

À partir de ce moment, le script de l'éditeur de liens peut être généré puisque toutes les informations sur l'emplacement des symboles sont disponibles. Chaque symbole devait être relocalisé en mémoire locale est ajouté au script de l'éditeur de liens et spécifiée comme étant localisé dans cette mémoire. Puisque ce fichier contient déjà des sections par défaut, aucune modification supplémentaire au niveau de ces sections et symboles n'est nécessaire. Les symboles situés en mémoire externe seront automatiquement placés dans cette mémoire par l'éditeur de lien par l'entremise des sections présentes par défaut. La dernière modification effectuée au script de l'éditeur de liens est la taille suggérée de la pile et du tas. Les tailles recommandées de ces segments sont calculées et inscrites au début du fichier.

L'estimation de la performance peut également être effectuée à ce point. À l'aide des listes des symboles en mémoire locale et en mémoire externe, du profil d'exécution ainsi que des différentes latences d'accès recueillies lors de la simulation du système, il est possible d'estimer la performance du programme après le partitionnement de mémoire. Par contre, cette mesure ne reste qu'un guide et ne constitue pas une valeur de performance absolue. Une simulation est nécessaire afin de recueillir des métriques plus précises.

Finalement, la dernière étape du processus est la génération des traces de cache. Cette étape est optionnelle et n'est requise que si l'utilisateur désire effectuer subséquemment l'optimisation de la configuration des mémoires cache. Lors de la génération de ces traces, les fichiers de trace de mémoire originaux sont lus et chaque accès à un symbole désormais situé en mémoire locale est filtré. Les accès aux symboles non filtrés sont finalement écrits dans la trace de cache en version compressée. Lorsque cette étape est achevée, le partitionnement de mémoire est également complètement terminé.

4.2.2. Récupération des symboles et des sections

Afin de récupérer la liste des symboles, la liste des sections ainsi que les relations entre les symboles et les sections, il est nécessaire d'analyser le fichier exécutable ELF. Ces opérations sont effectuées par la classe **CELFInfoLoader** (voir l'annexe A.1 pour la description des classes), utilisée par la classe **CELFInfo**, qui utilise quatre exécutables permettant de lire le contenu du fichier ELF et de décoder (*demangle*) les noms des symboles. Le décodage des symboles consiste à passer du nom C++ (*mangled*) à un nom compréhensible par un humain (*demangled*) [66].

CELFInfoLoader procède d'abord à charger la liste des sections en exécutant l'outil `mb-objdump`. Ce programme permet de récupérer une grande quantité d'information sur l'exécutable et en particulier la liste des sections avec l'option `-h`. L'information produite par `mb-objdump` est sous la forme suivante :

Tableau 4.1 Format de la sortie de l'outil mb-objdump

Ligne 1 :	Index	Nom	Taille	VMA	LMA	Offset	Alignement
Ligne 2 :	Attributs						

Par exemple, les deux premières entrées retournées pour un exécutable ciblant le MicroBlaze sont généralement les vecteurs de reset et d'exception logicielle. Ces vecteurs sont d'une taille de 8 octets, doivent être alloués et chargés en mémoire, sont en

lecture seule et sont exécutables. Les emplacements en mémoire réelle et en mémoire virtuelle sont identiques puisque la mémoire virtuelle n'est pas utilisée. Finalement, l'alignement de ces deux entrées est fait sur 4 octets.

Tableau 4.2 Exemple de sortie de l'outil mb-objdump

0	.vectors.reset	00000008	00000000	00000000	000000f4	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.vectors.sw_exception	00000008	00000008	00000008	00000104	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					

L'étape suivante consiste en la lecture des symboles et de leur adresse de départ. Ceci est effectué à l'aide de l'outil `mb-nm` qui permet d'énumérer tous les symboles contenus dans le fichier EFL ainsi que leur adresse de départ. Par contre, il est nécessaire d'exécuter `mb-nm` une deuxième fois, cette fois-ci avec l'option `-size-sort`, afin de récupérer la taille de chacun des symboles.

Lorsque toutes les sections et les symboles ont été chargés, **CELFInfoLoader** procède alors au décodage du nom des symboles. Cela pourrait normalement se faire directement avec l'outil `mb-nm` et intégré à l'étape précédente, mais dû à un bogue dans l'implémentation de la version spécifique au MicroBlaze de l'outil, cette fonctionnalité est défectueuse. Il est donc nécessaire d'utiliser l'outil `c++filt` qui prend en argument le nom encodé et retourne le nom décodé en sortie. Cet outil est utilisé dans le mode en ligne afin d'accélérer la vitesse de traitement des symboles.

Finalement, chaque symbole se voit associé à une section selon son adresse et chaque section se voit associé une liste des symboles. Pour chaque symbole dans la liste des symboles, l'algorithme recherche la section contenant l'adresse du symbole. La section est alors associée au symbole et le symbole est ajouté à la liste des symboles de la section. Ces associations sont très utiles lors de la phase de génération du script de l'éditeur de liens .

Lorsque toutes ces opérations sont complétées, les listes de symboles et de sections sont retournées à l'objet **CELFInfo** sous la forme de deux arbres bicolores indexés par adresse de départ et contenant respectivement des objets de type **CELFSymbol** et **CELFSection**. L'objet **CELFInfo** permet alors de récupérer un symbole ou une section associé à une adresse ou de récupérer un symbole ou section selon son nom. Il permet également de récupérer la liste complète des symboles et des sections

4.2.3. Lecture de la trace

La lecture des fichiers de trace de mémoire se fait à l'aide de la classe **CMemoryProfileFileAccess** qui hérite de la classe **CProfileFileAccess**. La classe de base abstraite **CProfileFileAccess** implémente la lecture et l'écriture de l'en-tête de fichier de trace ainsi que la lecture et l'écriture des types de base (*byte*, *short*, *int*, *long*). La lecture et l'écriture des entrées de trace sont la responsabilité des classes dérivées **CMemoryProfileFileAccess** et **CCacheProfileFileAccess**.

La class **CProfileFileAccess** utilise un objet de type **RandomAccessFile** (inclus dans Java) afin d'implémenter les fonctions de bas niveau de lecture et d'écriture en mode binaire. Par contre, **RandomAccessFile** utilise un mode de lecture et d'écriture direct sans utiliser de tampon. Ceci pose de sérieux problèmes de performance. Java propose une vaste sélection de classes permettant l'accès aléatoire à des fichiers dont des classes utilisant un tampon afin d'accélérer les accès mais aucune classe dans Java 1.5 ne supporte l'accès aléatoire avec tampon d'un fichier binaire. Un mécanisme de tampon a donc été implémenté dans la classe **CProfileFileAccess** qui permet ainsi l'accès aléatoire accéléré par un tampon à un fichier binaire.

Ce tampon fonctionne comme une mémoire cache. Lorsqu'une donnée à lire n'est pas présente dans le tampon, une lecture de la taille du tampon est effectuée dans le fichier et remplace les anciennes données présentes dans le tampon. Lorsqu'une écriture doit être

effectuée, le tampon est d'abord chargé avec les données requises et l'écriture s'effectue dans le tampon, qui est marqué comme étant modifié (*dirty*). Lorsqu'une nouvelle écriture ou lecture doit s'effectuer dans le tampon et si ce dernier a été modifié, il est d'abord écrit sur le disque avant de charger les nouvelles données.

En plus d'implémenter un tampon, la lecture et l'écriture des types de base ainsi que les fonctions de plus haut niveau de lecture et d'écriture d'en-tête de fichier de trace, la classe **CProfileFileAccess** permet également de déplacer le curseur de position courante au début de l'en-tête et au début des données de trace.

La classe **CMemoryProfileFileAccess**, dérivant de **CProfileFileAccess**, permet de lire l'en-tête spécifique à la trace de mémoire. Par contre, pour l'instant, cet en-tête est identique à l'en-tête lu par la classe de base **CProfileFileAccess**. De plus, **CMemoryProfileFileAccess** permet de lire et d'écrire une entrée de trace (**CMemoryProfileFileAccess**) d'un fichier de trace compressé ou non.

La lecture du fichier de trace est initiée par **CMemoryEvaluator**. Chaque accès lu est envoyé au module de profilage via une FIFO qui peut ensuite effectuer son traitement sur l'accès. Une FIFO synchronisée est utilisée puisque le profilage est effectué parallèlement par plusieurs tâches et chaque tâche doit pouvoir lire une entrée dans cette FIFO sans affecter l'intégrité des données.

4.2.4. Profilage

L'étape de profilage implique de multiples interactions entre les classes mais n'est pas complexe pour autant. **CMemoryEvaluator** est encore une fois le gestionnaire du processus. Le profilage est tout d'abord réalisé en deux étapes : la trace d'instructions est d'abord traitée, suivie de la trace de données. Pour chacune de ces étapes, **CMemoryEvaluator** s'occupe premièrement de créer les tâches qui effectueront le

profilage en parallèle (Figure 4.3). **CMemoryEvaluator** effectue ensuite la lecture des accès dans les fichiers de trace et les envoient vers les tâches de profilage. Au fur et à mesure que les accès sont écrits dans la FIFO, chaque tâche lit un nombre d'accès de cette FIFO et effectue son traitement sur ces accès. Lorsque les tâches ont terminé leur travail pour l'entièreté de la trace, **CMemoryEvaluator** fusionne les profils de chaque tâche. Lorsque les instructions et les données ont été traitées, deux profils distincts restent: le premier contient le profil des instructions et le deuxième, le profil des données. Ces deux profils sont par la suite fusionnés afin d'obtenir le profil final pour l'ensemble des symboles.

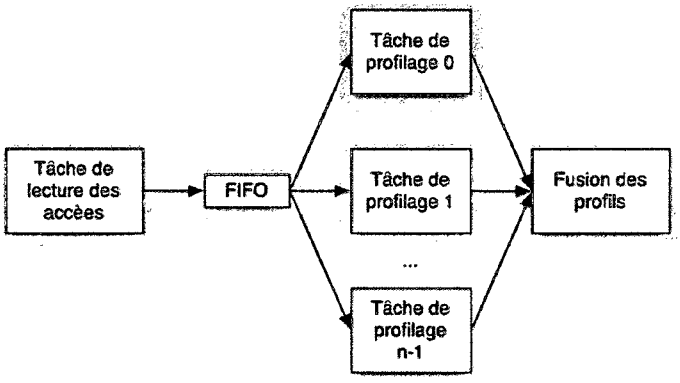

Figure 4.3 Modèle de multithreading pour le profilage

4.2.4.1.Tâche de lecture

La première tâche est la tâche de lecture de la trace qui effectue le travail décrit à la section précédente. Cette tâche écrit les accès lus dans une FIFO afin de les faire traiter par les tâches de profilage. Afin de diminuer la surcharge causée par la FIFO qui doit être *thread-safe*, c'est-à-dire que son état doit rester cohérent lors d'accès concurrents, les accès lus sont regroupés par groupes d'un maximum de 1000 accès. Cela permet à la tâche de lecture de la trace de réduire le nombre d'accès en écriture à la FIFO et également aux tâches de profilage de réduire le nombre d'accès concurrentiels en lecture à la FIFO.

4.2.4.2.Tâches de profilage

Le profilage est effectué par les classes **CInstructionMemoryProfileProcessor** et **CDataMemoryProfileProcessor** qui dérivent de **CMemoryProfileProcessor**. La classe **CMemoryProfileProcessor** implémente la méthode *run()* de la classe **Thread** et effectue dans celle-ci la lecture dans la FIFO des accès par bloc de 1000 accès ou moins.

Chacune des classes dérivées implémente une méthode (*processProfileFileEntry()*) qui est appelée par la méthode *run()* lorsqu'un accès doit être ajouté au profil. Pour chacune des classes, la méthode détermine le symbole correspondant à l'accès à l'aide de la classe **CELFnfo**. Une entrée de type **CMemoryProfileEntry** correspondant au symbole est ensuite recherchée dans le profil existant. Si l'entrée existe déjà dans le profil, son compte d'accès en lecture ou en écriture est incrémenté de 1. Si le symbole n'est pas déjà présent dans le profil, une nouvelle entrée est alors créée.

En plus de ce traitement, la classe **CDataMemoryProfileProcessor** détermine également l'utilisation de la pile et du tas. Pour chaque accès à traiter, l'algorithme suivant est appliqué :

```
static adressePlusBasseUtiliseeDansPile
static adressePlusHauteUtiliseeDansTas
Si l'adresse de l'accès est contenue dans la pile
    Si adressePlusBasseUtiiseeDansPile >= adresse de l'accès
       adressPlusBasseUtiliseeDansPile = adresse de l'accès
    Fin si
Sinon
   Si l'adresse de l'accès est contenue dans le tas
      Si adressePlusHauteUtiliseeDansTas < adresse de l'accès
          adressePlusHauteUtiliseDansTas = adresse de l'accès
      Fin Si
   Fin Si
Fin Si
```

Figure 4.4 Algorithme d'évaluation de l'utilisation de la pile et du tas

À la fin de l'analyse, les tailles maximales utilisées de la pile et du tas sont augmentées de 50% afin de compenser pour les conditions lors des exécutions subséquentes qui peuvent être différentes que celles rencontrées lors du recueil des traces d'exécution.

Lorsque chaque tâche de profilage a terminé son traitement, les résultats de profilage sont fusionnés dans un seul objet de type **CInstructionMemoryProfileProcessor** ou **CDataMemoryProfileProcessor** selon que l'algorithme analyse la trace d'instruction ou la trace de données. La méthode de fusion est simple : l'algorithme parcourt chaque profil et fait la sommation des accès des entrées faisant référence au même symbole. Dans le cas de **CDataMemoryProfileProcessor**, en plus de cette fusion, la taille maximale utilisée de la pile et du tas pour chaque profil est également déterminée.

Le même processus est effectué lorsque les instructions et les données ont été traitées. Les profils d'instructions et de données sont fusionnés afin de donner un profil global d'utilisation pour tous les symboles. Ce profil global est stocké dans **CMemoryEvaluator** à l'aide d'une instance de **CMemoryProfile**. Les tailles maximales de la pile et du tas sont également récupérées et stockées dans des attributs de la classe **CMemoryEvaluator**.

4.2.5. Algorithme de relocalisation des symboles

L'algorithme de relocalisation des symboles s'occupe tout d'abord de déterminer quels symboles doivent être relocalisés en mémoire locale. Les symboles restants sont par la suite marqués comme localisés en mémoire externe. Il est à noter que seuls les symboles qui ont été accédés durant l'exécution se retrouvent dans l'une de ces listes. Les listes des symboles en mémoire locale et en mémoire externe, de même que le profil global ne constituent pas un inventaire exhaustif des symboles présents dans le programme. La seule liste exhaustive des symboles se trouve dans la classe **CELFInfo**.

La relocalisation assume que chaque symbole présent dans le fichier ELF peut être déplacé individuellement dans la mémoire locale. Ceci implique que chaque symbole doit être défini dans sa propre section de code lors de la compilation. Ceci est fait avec les options de compilation de GCC spécifiés à la section 2.2.1. Pour les fonctions et les données définies en assembleur, il est nécessaire d'assigner une section à chacun de ces symboles manuellement en suivant la nomenclature de GCC.

La relocalisation utilise deux paramètres afin de limiter la taille de la mémoire locale utilisée : le pourcentage de couverture et la taille maximale de mémoire locale pouvant être utilisée. Pour chaque symbole présent dans le profil, l'algorithme détermine le pourcentage d'utilisation du temps total par ce symbole. Ce pourcentage est déterminé par l'équation:

$$Util_{\%} = \frac{NbLectures \times LatMemExtLecture_{Moy} + NbÉcritures \times LatMemExtÉcriture_{Moy}}{Cycles_{total}} \times 100\% \quad \text{Eq. 4.1}$$

Le nombre total de cycles dans le profil global ($Cycles_{total}$) est calculé selon l'équation suivante :

$$Cycles_{total} = \sum_{i}^{Symboles} NbLectures_i \times LatenceMemExtLecture + NbÉcritures_i * LatenceMemExtÉcriture \quad \text{Eq. 4.2}$$

Puisque le profil est trié en ordre décroissant, les symboles utilisant le plus de temps apparaissent au début du profil. Il suffit donc de faire la somme des pourcentages d'utilisation et d'ajouter des symboles à la liste des symboles en mémoire locale jusqu'à ce que le pourcentage cumulatif atteigne le pourcentage de couverture. Également, la taille de chaque symbole est récupérée et additionnée à la taille courante de la mémoire locale afin de déterminer la taille optimale.

La pile et le tas (ainsi que certaines autres sections) subissent un traitement spécial par rapport aux autres symboles. En effet, le symbole présent dans le profil sera le symbole indiquant le début de la pile ou du tas. La relocalisation de ce symbole seul n'est pas suffisante afin de relocaliser ces segments dans la mémoire locale si l'algorithme détermine que l'un ou l'autre doit être relocalisé. La seule relocalisation de ces symboles entraînerait un comportement erratique du programme qui plantera assurément puisque les symboles indiquant la fin de la pile ou du tas seraient toujours situés en mémoire externe. Il est donc nécessaire de relocaliser la section elle-même de la pile ou du tas en mémoire locale.

La méthode de relocalisation présentée jusqu'ici contient par contre une faille. Si les symboles utilisant une grande quantité de cycles sont également volumineux, la taille de la mémoire locale peut rapidement exploser. Un exemple d'un cas problématique serait un algorithme travaillant sur un tableau de grande taille (par exemple de 4 Mo). Les accès au tableau, qui ne seraient identifiés que par un seul symbole, représenteraient une grande partie du temps d'accès total à la mémoire externe par le programme. L'algorithme déterminerait donc que ce tableau doit être présent en mémoire locale. Il est bien évident qu'une mémoire locale d'une taille de 4 Mo n'est que rarement envisageable dans un système sur puce. Afin de résoudre ce problème, une limite de taille est spécifiée par l'utilisateur. L'algorithme tentera alors de rencontrer la contrainte de couverture tout en respectant la taille maximale de la mémoire locale. Lorsque la taille du symbole à ajouter causerait un dépassement de la taille maximale de la mémoire locale ou que le symbole utiliserait 75% de la mémoire locale par lui-même, l'algorithme passerait tout simplement au prochain symbole dans le profil.

La taille de la mémoire locale pose un deuxième problème. Puisqu'en matériel, la taille de la mémoire locale est en général une puissance de deux, il arrive parfois que l'algorithme atteigne le pourcentage de couverture et qu'une certaine quantité de mémoire reste toujours disponible. Par exemple, pour un programme donné, 3008 octets

sont nécessaires afin d'atteindre le pourcentage de couverture. Par contre, la taille de la mémoire locale doit alors être de 4096 octets. Il reste donc 1088 octets de libres dans la mémoire locale. Afin de remplir cet espace libre, l'algorithme continuera donc à traverser le profil en ajoutant des symboles jusqu'à ce que la taille finale soit le plus près de 4096 octets.

Un troisième problème potentiel est causé par la relocalisation des symboles situés dans la section *.bss*. En effet, cette section est initialisée à 0 au démarrage de l'application par le moteur d'exécution du langage C. Lorsque des symboles situés dans le *.bss* sont relocalisés en mémoire locale, ils ne sont pas initialisés à 0. En simulation ceci ne pose aucun problème puisque les mémoires sont automatiquement initialisées à 0. Par contre, sur un FPGA, le comportement du logiciel peut être affecté puisque les mémoires ne sont pas explicitement initialisées. Une solution possible serait de s'assurer que le programme n'utilise jamais de variables qui ne sont pas explicitement initialisées par le programmeur.

La relocalisation des symboles en mémoire locale pose un dernier problème, celui du calcul de la taille des symboles. Dans un système sur puce, la mémoire locale et la mémoire externe sont bien souvent distancées par plusieurs méga-octets dans l'espace d'adressage et ceci cause un problème lors des branchements. En effet, les instructions de branchements prennent souvent en argument un immédiat afin de spécifier le déplacement. Cet immédiat a une taille de 16 bits signé et est donc limité à un déplacement maximal de 32Ko. Lorsqu'une instruction présente en mémoire locale doit effectuer un branchement vers une instruction en mémoire externe ou vice-versa, l'éditeur de lien insère une instruction IMM qui permet alors de produire un immédiat de 32 bits. Par contre, ces instructions supplémentaires ne sont rajoutées que lors de la phase d'édition des liens, après la génération du script de l'éditeur de liens optimisé par l'algorithme. Il est donc nécessaire d'estimer l'augmentation de taille de chaque symbole qui sera causée par ces instructions supplémentaires. L'algorithme ajuste donc

la taille de chaque symbole avec un pourcentage empirique de 1,5%. Ce pourcentage a été déterminé en déplaçant plusieurs symboles de la mémoire externe à la mémoire locale et en observant l'augmentation de leur taille. En moyenne, la taille des symboles augmentait d'environ 1%. Ce pourcentage a été augmenté à 1,5% afin de parer à certaines éventualités.

Avec tous ces algorithmes et mécanismes, l'outil est alors en mesure de déterminer le contenu de la mémoire locale et d'estimer la taille de cette dernière. Toutes les données nécessaires sont donc disponibles afin d'estimer la performance après partitionnement et d'effectuer la génération du script de l'éditeur de liens.

4.2.6. Algorithme d'estimation de la performance

La performance de l'application après le partitionnement de mémoire est estimée à l'aide du nombre d'accès en mémoire locale et en mémoire externe. Les accès sont divisés en accès aux instructions et aux données. Les accès aux instructions en mémoire locale et en mémoire externe sont d'abord multipliés par la latence de leur mémoire respective pour donner le nombre de cycles, puis ces latences sont ensuite additionnées. Le même calcul est fait aux accès aux données. Ensuite, la valeur maximale entre le nombre de cycles d'instructions et de cycles de données est déterminée, ce qui donne le nombre de cycles total utilisés par l'exécution de la version optimisée. Le nombre de cycles total de la version originale non optimisée est finalement divisé par le nombre de cycles total de la version optimisée, ce qui donne l'accélération estimée. Les métriques disponibles suite à cette estimation de performance sont :

- le nombre total de cycles obtenu en pré-optimisation;
- le nombre de cycles pour les instructions et données en mémoire externe obtenu en pré-optimisation;
- le nombre de cycles pour les instructions et les données en mémoire externe qui seront relocalisées en mémoire locale;

- le nombre de cycles total obtenu en post-optimisation;
- le nombre de cycles pour les instructions et données en mémoire locale obtenu en post-optimisation;
- le nombre de cycles pour les instructions et données en mémoire externe obtenu en post-optimisation.

Cette estimation n'est par contre pas exacte et donc ne permet pas d'estimer avec précision le nombre de cycle que la version optimisée nécessitera lors de l'exécution. Premièrement, elle ne compte pas les instructions récupérées mais non exécutées puisqu'elles ne sont pas présentes dans le fichier de trace (voir section 3.4.3). De plus, seuls les accès à des données présentes en mémoire sont comptabilisés, les accès aux périphériques ne sont pas pris en compte et ceci peut donc biaiser le calcul davantage. Finalement, certains symboles relocalisés par le partitionnement de mémoire ne peuvent être relocalisés en pratique, par exemple si le symbole n'est pas défini dans sa propre section. Ceci est le cas pour certaines librairies précompilées, pour le moteur d'exécution du langage C et pour des étiquettes (*labels*) définies en assembleur.

4.2.7. Génération du script de l'éditeur de liens

Afin de pouvoir générer le script de l'éditeur de liens, l'utilisateur doit d'abord spécifier les mémoires disponibles dans le système. Ceci se fait à l'aide de méthodes disponibles dans la classe **CMemoryEvaluator** qui à son tour appelle les méthodes appropriées du sous-système de génération du script de l'éditeur de liens Lorsque cela est fait et que le partitionnement a été effectué, **CMemoryEvaluator** procède alors en assignant les tailles minimales estimées de la pile et du tas. Ensuite, chaque symbole à relocaliser est ajouté sous forme d'une nouvelle section située en mémoire locale. L'algorithme réalisant cette opération est décrit à la Figure 4.5.

Lorsque toutes les sections ont été ajoutées, **CMemoryEvaluator** peut ensuite appeler la méthode de **CLinkerScriptGenerator** permettant de générer le script de l'éditeur de liens Ce dernier est finalement écrit dans un fichier texte et ce fichier est prêt à être utilisé lors de la prochaine compilation.

```
Pour chaque symbole à relocaliser en mémoire locale
   Si la section liée au symbole doit être déplacée en
    mémoire locale au lieu du symbole seul
      AjoutSection(nom de la section liée au symbole)
   Sinon
      NomSectionSymbole = nom de la section liée au symbole +
                          nom encodé du symbole
      AlignementPréObjet = Déterminé selon le  type de la section
                           (text, data, bss, etc)
      Objet = NomSectionSymbole
      AlignementPostObjet = Déterminé selon le type de la section
                           (text, data, bss, etc)
      AjoutSection(NomSectionSymbole, AlignementPréObjet, Objet,
         AlignementPostObjet)
   Fin Si
Fin Pour chaque
```

Figure 4.5 Algorithme d'ajout des sections en mémoire locale

4.2.8. Génération de la trace de cache

La dernière opération est la génération des traces de cache en vue de l'optimisation de la configuration des mémoires cache. Cette étape est optionnelle puisque l'utilisateur pourrait ne pas vouloir optimiser la mémoire cache ou même pourrait ne pas vouloir utiliser de mémoire cache du tout.

La génération de la trace de cache passe d'abord par l'identification de la liste complète des symboles qui seront situés en mémoire externe. Cette opération est effectuée en parcourant la liste des symboles (contenue dans **CELFInfo**) et en ne copiant dans une liste séparée que les symboles ne se retrouvant pas en mémoire locale. Ceci génère alors une liste complète des symboles en mémoire externe, contrairement à la liste générée à l'étape de relocalisation qui ne contient que les symboles profilés.

Ensuite, la liste des symboles (ordonnée en ordre croissant d'adresse de départ) est aplanie (Figure 4.6) afin d'éliminer les espaces vides en mémoire laissés par les symboles situés maintenant en mémoire locale. Cette opération est effectuée afin de tenter de reproduire la structure de la mémoire résultante après la recompilation avec le nouveau script de l'éditeur de liens .

Lorsque ce travail est accompli, les traces de cache d'instructions et de données peuvent alors être générées. L'algorithme accompli le même travail pour les deux traces. Chaque entrée dans le fichier de trace original est lue. Pour chacune de ces entrées, l'algorithme détermine d'abord si l'entrée fait référence à un symbole désormais situé en mémoire externe. Si oui, l'adresse accédée par l'entrée est recalculée par rapport à la nouvelle position du symbole en mémoire. Ceci permet de générer une nouvelle entrée de trace de cache.

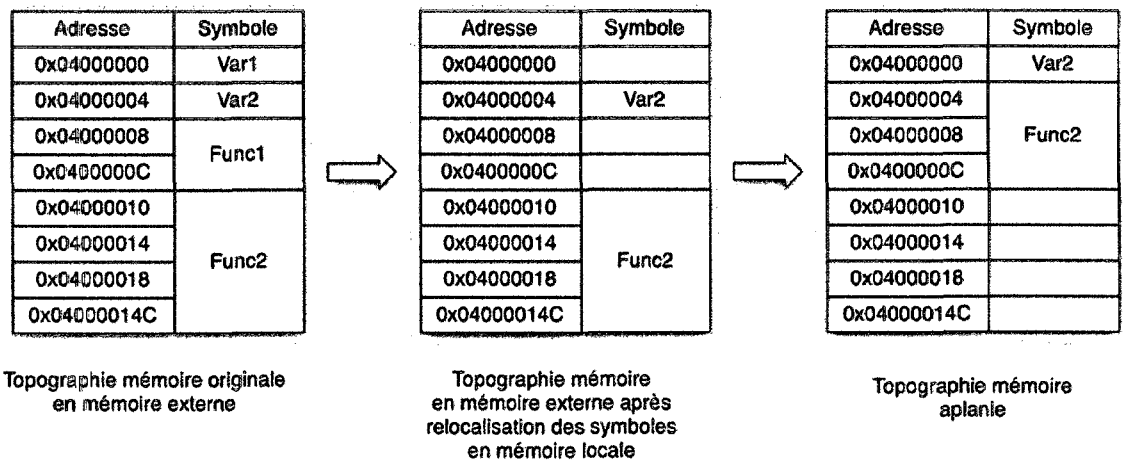

Figure 4.6 Aplanissement de la topographie mémoire

L'algorithme effectue ensuite le traitement pour la compression du fichier des entrées du fichier de trace de cache et les entrées sont finalement écrites dans le fichier de trace. Cette série d'opération est répétée pour la trace d'instructions et de données afin de générer les deux traces de cache nécessaires à la phase d'optimisation de la configuration de la mémoire cache.

4.3. Optimisation de la mémoire cache

L'optimisation de la mémoire cache permet de déterminer automatiquement la configuration de cache utilisant le moins de mémoire tout en permettant d'accélérer de façon significative l'exécution du programme. Cette optimisation peut se faire à la suite du partitionnement de mémoire ou indépendamment de celui-ci.

Dans cette section, le système permettant d'effectuer cette optimisation est présenté. L'architecture générale du système est d'abord décrite et est suivie d'une description plus spécifique de chacune des étapes impliquées dans ce processus.

4.3.1. Architecture

L'algorithme d'optimisation de la mémoire cache est basé sur la technique présentée dans [40] mais elle permet, grâce à l'utilisation de deux tâches, d'évaluer l'ensemble des configurations de cache en parallèle. L'architecture adoptée est donc conséquente de ces choix.

Le système d'optimisation de cache est beaucoup plus simple que celui du partitionnement de mémoire et ne comprend que trois étapes majeures (Figure 4.7). La première étape consiste à lire les entrées du fichier de trace et à simuler les accès pour chaque configuration de cache. Lorsque la simulation est terminée, le système évalue la performance de chaque configuration de cache. Finalement, un résultat est attribué à chaque configuration et la meilleure configuration de cache est sélectionnée. Cette série d'étapes est effectuée indépendamment pour la trace d'instructions et de données.

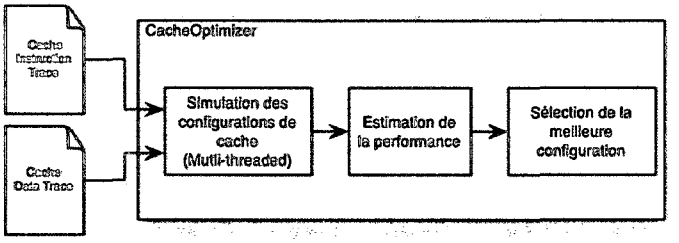

Figure 4.7 Diagramme de flux de données de l'optimisation de mémoire cache

4.3.2. Simulation des configurations de cache

La simulation des configurations de caches se fait en deux étapes : la simulation des configurations de caches d'instructions et la simulation des configurations de caches de données. Également, l'algorithme d'inclusion utilisé pour effectuer ces simulations signifie qu'il est nécessaire d'effectuer une simulation par longueur de ligne de cache. Puisque le MicroBlaze permet de configurer chacune des caches avec une taille de ligne de cache de 4 ou 8 mots, quatre simulations sont donc nécessaires afin d'obtenir les informations de performance pour toutes les configurations possibles.

4.3.2.1.Déroulement

La simulation de caches d'instructions et de données est faite séquentiellement en débutant par la mémoire cache d'instructions. Par contre, lors de la simulation, les caches ayant une taille de ligne de cache de 4 et 8 mots sont simulés en parallèle dans deux tâches différentes.

La tâche de lecture du fichier de trace, démarrée par **CCacheEvaluator** (voir l'annexe A.2 pour la description des classes), écrit les accès lus dans le fichier de trace d'instructions dans deux FIFO synchronisées qui communiquent chacune avec une tâche de simulation : la tâche simulant les caches de 4 mots et la tâche simulant les caches de 8 mots. Les mêmes accès sont écrits à la fois dans les deux FIFO. Ensuite, chacune des

tâches de simulation lit sa propre FIFO et pour chaque accès lu, exécute l'algorithme de simulation. Les tâches terminent leur traitement lorsque le fichier de trace est lu en entier. Ces opérations sont ensuite répétées avec la trace de données afin de simuler les caches de données.

4.3.2.2.Algorithme de simulation

L'algorithme de simulation utilisé est basé sur le principe d'inclusion utilisé dans [40]. Comme expliqué au chapitre 2, le principe d'inclusion stipule que, si un accès provoque un succès dans un cache d'une certaine taille, cet accès provoquera également un succès dans un cache de taille plus grande. En effet, puisque la méthode pour déterminer à quelle ligne de cache la donnée doit être située est une opération de type modulo, une donnée présente dans une cache d'une certaine taille se retrouvera toujours dans une cache de taille plus grande. L'inverse n'est par contre pas vrai, une donnée se retrouvant dans une cache de taille plus importante ne se retrouvera pas nécessairement dans une cache plus petite puisqu'il se peut que la donnée en question ait été remplacée par une autre à cause du manque d'espace.

Ce principe impose par contre une restriction : seule la taille du cache peut varier, la taille des lignes de cache ou la politique de remplacement ne peut varier d'une cache à l'autre. Si la taille de ligne de cache varie, l'alignement des lignes de cache entre plusieurs tailles de cache ne correspond plus et ceci viole la prémisse de base du principe. De façon similaire, si la politique de remplacement varie, il n'est alors pas garanti qu'un bloc se retrouvera dans deux caches de même taille et encore moins dans deux caches de taille différente.

Dans le contexte du MemoryOptimizer, les caches à optimiser n'étant que des caches de type *accès direct*, la politique de remplacement n'a donc pas à être considérée. Par contre, la taille des lignes de cache peut varier. Afin de permettre l'utilisation du

principe d'inclusion, il est donc nécessaire de simuler séparément les deux tailles de ligne de cache d'où l'utilisation de deux tâches de simulation.

4.3.3. Estimation de la performance

L'estimation de la performance est faite pour chaque configuration de cache simulée. Lors de la simulation, le nombre de succès en lecture et en écriture est comptabilisé ainsi que le nombre d'échecs en lecture et en écriture. De ces nombres ont peut déduire le nombre total de succès et le nombre total d'échecs. Il devient ensuite aisé de calculer les différentes mesures de performances.

La première mesure de performance d'un cache est le pourcentage de succès et d'échecs. Ce calcul se fait simplement en divisant le nombre total de succès ou d'échecs sur le nombre total d'accès à la cache. La deuxième mesure calculée est le nombre de cycles d'exécution qu'entraînera l'utilisation de ce cache. Cette mesure est calculée en multipliant le nombre d'accès par la latence de la mémoire. Comme un succès implique une latence différente d'un échec et qu'une écriture implique également une latence différente d'une lecture, il est nécessaire de calculer la latence totale pour chacun de ces types d'accès. La performance en terme de cycles pour chaque cache est alors calculée par l'équation suivante :

$$Performance_{cycles} = NbCycles_{Lecture-Hit} + NbCycles_{Lecture-Miss} + NbCycles_{Écriture-Hit} + NbCycles_{Écriture-Miss} \quad \text{Eq. 4.3}$$

où chaque NbCycles est calculée par :

$$\begin{aligned} NbCycles_{Lecture-Hit} &= NbHits_{Lecture} \times Latence_{Cache-Lecture} \\ NbCycles_{Lecture-Miss} &= NbMiss_{Lecture} \times Latence_{XCL-Lecture} + NbMiss_{Lecture} \times (Taille_{Ligne} - 1) \\ NbCycles_{Écriture-Hit} &= NbHits_{Écriture} \times Max(Latence_{XCL-Écriture}, Latence_{Cache-Écriture}) \\ NbCycles_{Écriture-Miss} &= NbMiss_{Écriture} \times Max(Latence_{XCL-Écriture}, Latence_{Cache-Écriture}) \end{aligned} \quad \text{Eq. 4.4}$$

L'algorithme utilisé présente par contre une faille qui peut biaiser ces calculs de performance. Les caches sont considérés actives pour toute la durée de la simulation et les instructions de désactivation du cache ou de réinitialisation du cache disponibles dans le MicroBlaze sont complètement ignorées. Les accès sont donc toujours tous simulés et ceci peut avoir pour effet de sous-estimer le nombre de cycles. Également, puisque les instructions récupérées mais non exécutées lors d'un branchement ne sont pas recueillies par la trace, ceci implique également une sous-estimation du nombre de cycles d'exécution.

4.3.4. Sélection de la configuration optimale

La performance en termes de cycles du cache ne permet pas à elle seule de sélectionner la meilleure configuration de cache dans le contexte d'un système sur puce. En effet, si la performance était le seul critère, la taille des caches serait trop souvent le maximum alloué par le matériel. Il est donc nécessaire de prendre en compte le coût en matériel de ces caches. Pour ce faire, un calcul simple basé sur l'utilisation de la BRAM par le cache est fait. La BRAM est allouée sur le FPGA par bloc de 2 Ko. La taille de la mémoire des données se traduit directement en un certain nombre de blocs de BRAM. La taille de la mémoire de *tags* est par contre fonction du nombre de lignes de cache. Chaque entrée dans ce tableau est composée de l'étiquette et du bit de validité de la ligne de cache. En connaissant le nombre de lignes de cache, il est possible de calculer la taille en bits de chaque étiquette et donc de connaître la taille de chaque entrée dans le tableau des étiquettes.

Le résultat est calculé en divisant l'accélération par le nombre de blocs de BRAM utilisés. L'accélération est simplement le nombre de cycles d'exécution sans la présence du cache, divisée par le nombre de cycles d'exécution avec la présence du cache qui est calculé avec la relation présenté à l'équation 4.5. Ceci implique donc que plus le résultat

est élevé, plus la configuration du cache est efficace. Il suffit donc de sélectionner le cache d'instructions et de données qui affiche le plus grand résultat.

$$Score_{cache} = \frac{Accélération_{Cache}}{NbBlocsBRAM_{Cache}} \qquad \text{Eq. 4.5}$$

Dans ce chapitre l'outil MemoryOptimizer a été présenté. Cet outil permet d'effectuer le partitionnement de mémoire et l'optimisation des mémoires caches d'un système sur puce à partir de traces d'exécution générées par la simulation du système. Les résultats générés par l'outil incluent la nouvelle configuration du programme en mémoire, la configuration des mémoires caches ainsi qu'une estimation de la performance du système optimisé. L'étape suivant consiste donc en la validation de cette approche à l'aide d'applications de test. Le chapitre suivant présente donc la méthodologie d'expérimentation, les résultats obtenus et une analyse de ces résultats.

CHAPITRE 5

ANALYSE, PERFORMANCES ET DISCUSSION

L'objectif principal de l'outil d'exploration architectural étant d'accélérer l'exécution d'une application contenue en mémoire externe, en tirant profit de la hiérarchie de mémoire, il est de mise d'analyser les performances obtenues pour différentes applications à la suite de cette exploration architecturale.

Ce chapitre présente donc tout d'abord la méthode de recueil des résultats ainsi que les applications utilisées pour vérifier la performance de l'outil. Pour chacune de ces applications, les résultats sont présentés et discutés. Brièvement, l'outil permet d'obtenir une performance très proche de la performance obtenue lorsque le programme est contenu complètement en mémoire locale tout en diminuant par près de 6 fois la taille de la mémoire utilisée sur le FPGA.

Une comparaison avec d'autres travaux est ensuite effectuée et la possibilité d'étendre l'outil à d'autres processeurs est examinée. Finalement, une analyse de l'outil d'exploration architecturale par rapport aux objectifs énoncés au premier chapitre est présentée.

5.1. Méthodologie d'analyse des performances

La collecte de résultats est effectuée à l'aide de deux applications de test. De plus, deux volets sont examinés : l'exploration architecturale et la compression de trace. Dans cette section, la méthodologie de collecte des résultats pour chacun de ces volets est d'abord expliquée. La méthodologie sera ensuite suivie par la présentation des deux applications

permettant d'analyser l'efficacité de l'outil de même qu'une discussion sur les paramètres utilisés pour chacune de ces applications.

5.1.1. Méthodologie

Le premier volet consiste à évaluer la performance des configurations générées par l'outil d'exploration et de comparer la performance de ces configurations en simulation par rapport à la performance des configurations non optimisées et optimisées à la main. En premier lieu, seule la mémoire cache est optimisée. Ensuite, seule la mémoire locale est partitionnée. Finalement, le partitionnement de mémoire locale et l'optimisation de mémoire cache sont activées.

Le deuxième volet consiste à faire varier la taille des traces d'exécution afin de déterminer l'efficacité des algorithmes de compression sur la taille de celles-ci, de même que sur la durée du processus d'exploration architectural. La trace de mémoire et la trace de cache sont analysées indépendamment et les résultats sont comparés entre eux.

Deux applications de test sont utilisées : Dhrystone et IDCT. Pour chacune de ces applications, la méthodologie de génération de résultats suivante, discutée brièvement plus haut, est appliquée :

1. Simuler l'exécution lorsque l'application est contenue exclusivement en mémoire externe et avec la mémoire cache désactivée.
2. Simuler l'exécution lorsque l'application est contenue exclusivement en mémoire locale.
3. Effectuer plusieurs simulations afin de déterminer la configuration optimale des caches ainsi que la performance obtenue avec cette configuration. Ceci est effectué en démarrant avec des caches de 64 octets et 4 mots par ligne de cache pour les caches d'instructions et de données et en augmentant progressivement

les tailles de chacun des caches jusqu'à ce que la performance maximale possible (plus petit nombre de cycles) soit atteinte. Ceci peut demander une vingtaine de simulations.

4. Générer la trace de cache, exécuter l'optimisation automatique de la cache et simuler la nouvelle configuration de cache.
5. Générer la trace de mémoire, exécuter le partitionnement de mémoire en spécifiant une taille maximale de mémoire locale à utiliser. La taille maximale de 64 Ko est d'abord choisie et permet à l'algorithme de déterminer la taille optimale de la mémoire locale. Pour les exécutions subséquentes de l'algorithme, la taille maximale de la mémoire locale est progressivement diminuée jusqu'à la taille minimale de 2 Ko. Le nombre de configurations de mémoire testées dépendra donc du programme exécuté. Par exemple pour Dhrystone, ceci donne des tailles de mémoire locale de 8 Ko, 4 Ko et 2 Ko. Les configurations résultantes sont ensuite simulées afin de recueillir leurs performances en simulation et de comparer ces performances avec les performances estimées par l'algorithme.
6. Générer la trace de mémoire, exécuter le partitionnement de mémoire pour plusieurs tailles maximales de mémoire locale (tel qu'effectué à l'étape 5), exécuter l'optimisation de mémoire cache et finalement, simuler les configurations résultantes.
7. Effectuer plusieurs simulations en faisant varier la longueur de l'exécution afin de faire varier la taille de la trace de cache avec compression. Effectuer l'optimisation de la mémoire cache pour chacune des traces recueillies.
8. Répéter l'étape 7 en générant la trace de mémoire compressée et en effectuant le partitionnement de mémoire
9. Répéter les étapes 7 et 8 avec les traces non compressées.

Les simulations et les optimisations ont été effectuées sur un ordinateur Core Solo 1.83GHz sous Windows XP SP3 roulant dans une machine virtuelle avec 1 Go de mémoire vive, Xilinx EDK 9.1, Eclipse 3.1 et Java 1.6 update 7.

5.1.2. Dhrystone

Dhrystone [59] est un test de performance créé en 1984 par Reinhold P. Weicker. La dernière version, 2.1, a été conçue en 1988 et est la version la plus utilisée. C'est donc celle-ci qui est utilisée dans ce travail. Dhrystone permet de mesurer la performance d'un processeur en effectuant un grand nombre d'opérations sur des entiers et sur des chaînes de caractères. À l'issue de l'exécution, la performance est donnée sous la forme d'itérations de Dhrystone par seconde et de MIPS Dhrystone. La mesure de performance Dhrystone MIPS est calculée en divisant le nombre d'itérations par seconde par 1757. Le nombre 1757 est utilisé puisque lors de la conception du test, l'ordinateur VAX était très répandu et ce nombre correspond au nombre de MIPS du VAX. Le nombre de MIPS Dhrystone est donc relatif à la performance du VAX pour ce test.

Cette application de test a été choisie parce qu'elle est simple à comprendre et à porter sur un processeur comme le MicroBlaze. Également, le code source est relativement compact mais dépend de plusieurs librairies C [60], ce qui donne une taille finale à l'exécutable d'environ 64 Ko. Le jeu de données est très petit et le code possède une localité de référence élevée. Finalement, aucune communication avec des périphériques n'est nécessaire durant l'exécution. Tout ceci fait de Dhrystone un bon candidat afin de tester l'outil durant son développement et afin de déterminer l'efficacité de l'algorithme.

La longueur de l'exécution est contrôlée par le nombre d'itérations à effectuer. La longueur de l'exécution est directement proportionnelle au nombre d'itérations. Lors du premier volet, l'évaluation du partitionnement de mémoire et l'optimisation du cache, ce paramètre est fixé à 100 000 itérations. Ce nombre a été choisi puisqu'il permet à l'application de passer la majorité du temps à exécuter la boucle principale de Dhrystone

et ainsi de minimiser l'impact du temps requis par les fonctions d'initialisation du moteur d'exécution du langage C ou de l'initialisation du cache. Ceci reproduit donc plus fidèlement un programme exécuté sur un processeur embarqué qui peut fonctionner pendant des heures, voire des jours sans être réinitialisé.

Lors du volet d'analyse de la performance des algorithmes de compression des fichiers de trace et de la performance de l'algorithme d'exploration architecturale, le nombre d'itérations varie entre 10 000 et 400 000. Des traces de 12 Mo à 1 Go sont ainsi créées, ce qui permettra d'analyser le comportement des algorithmes selon la taille de traces.

Suite à l'optimisation manuelle du cache pour Dhrystone, il a été déterminé que la configuration permettant d'obtenir la meilleure performance est de 4 Ko de cache d'instructions et 1 Ko de cache de données. Ces deux caches sont configurées avec une taille de ligne de 4 mots.

5.1.3. IDCT

La deuxième application testée est une transformée inverse en cosinus discrète s'exécutant sur la plateforme logicielle SPACE qui elle-même utilise le RTOS μC [61]. La transformée elle-même ne comprend qu'une fonction d'environ 100 lignes de code et est exécutée dans une tâche μC. Par contre, les calculs effectués sont exigeants puisqu'ils consistent principalement en des chargements et stockages ainsi que des additions, soustractions et multiplications. La transformée est appliquée à répétition sur les blocs de données disponibles. Ces blocs de données sont générés par une tâche MicroC qui envoie les données à la tâche de calcul via une FIFO.

L'utilisation de la plateforme logicielle SPACE entraîne une légère surcharge en termes de mémoire utilisée et d'opérations à effectuer. En tout, l'application utilise 85 Ko de mémoire. L'utilisation des deux tâches entraîne également un coût de gestion des tâches

et de synchronisation afin d'assurer la communication entre ces deux tâches et un coût dû à la gestion des interruptions. De ce fait, un grand nombre de fonctions sont alors utilisées lors de l'exécution du programme.

La longueur de l'exécution est contrôlée par le nombre de blocs à traiter. Lorsque ce nombre de blocs est atteint, l'exécution cesse. Encore une fois, la longueur de l'exécution est directement proportionnelle au nombre de blocs à traiter. Lors du premier volet, le nombre de blocs est fixé à 1000 pour les mêmes raisons que Dhrystone. Lors du deuxième volet, le nombre de bloc varie entre 100 et 4000. Ceci permet de générer des traces de tailles variant de 12 Mo à 704 Mo.

Pour ce qui est de la taille optimale des caches, il a été déterminé que la configuration permettant d'obtenir la meilleure performance est de 64 Ko de cache d'instructions et de 32 Ko de cache de données, tout deux utilisant une taille de ligne de cache de 8 mots. Il est important de remarquer que cette taille de cache dépasse la taille totale du programme. Ceci est dû au fait que la taille de chaque cache augmente par puissances de deux.

5.2. Résultats de l'exploration architecturale

Dans cette section, les résultats de l'exploration architecturale pour Dhrystone et IDCT sont présentés. Pour chacune de ces applications, les performances estimées et simulées sont présentées. Ces performances sont ensuite comparées entre elles et avec les performances obtenues lorsque l'application est entièrement contenue en mémoire locale, entièrement contenue en mémoire externe et finalement, entièrement contenue en mémoire externe avec les mémoires caches optimisées.

5.2.1. Dhrystone

La Figure 5.1 illustre les résultats du nombre de cycles d'exécution estimés par l'outil et simulés pour chaque configuration de mémoire. La nomenclature pour chaque configuration est la suivante : méthode de configuration utilisée (manuelle ou automatique) - taille de la mémoire locale - taille du cache d'instructions/taille des lignes de cache – taille du cache de données/taille des lignes de cache.

Il est tout d'abord possible de remarquer que lorsque le programme est entièrement contenu dans la mémoire locale (Manuel – 64K-0K-0K), ce dernier s'exécute 10 fois plus rapidement que lorsqu'il est situé en mémoire externe (Manuel – 0K-0K-0K). Par contre, lorsque le programme est situé en mémoire externe et que la mémoire cache est configurée pour donner une performance maximale, le gain de vitesse n'est que de 6,7. Évidemment, pour atteindre cette performance, seuls 5 Ko de mémoire cache sont nécessaires, comparativement au 64 Ko nécessaires en mémoire locale. Passer d'un gain de 6,7 à 10 nécessite donc une mémoire au moins 10 fois plus volumineuse.

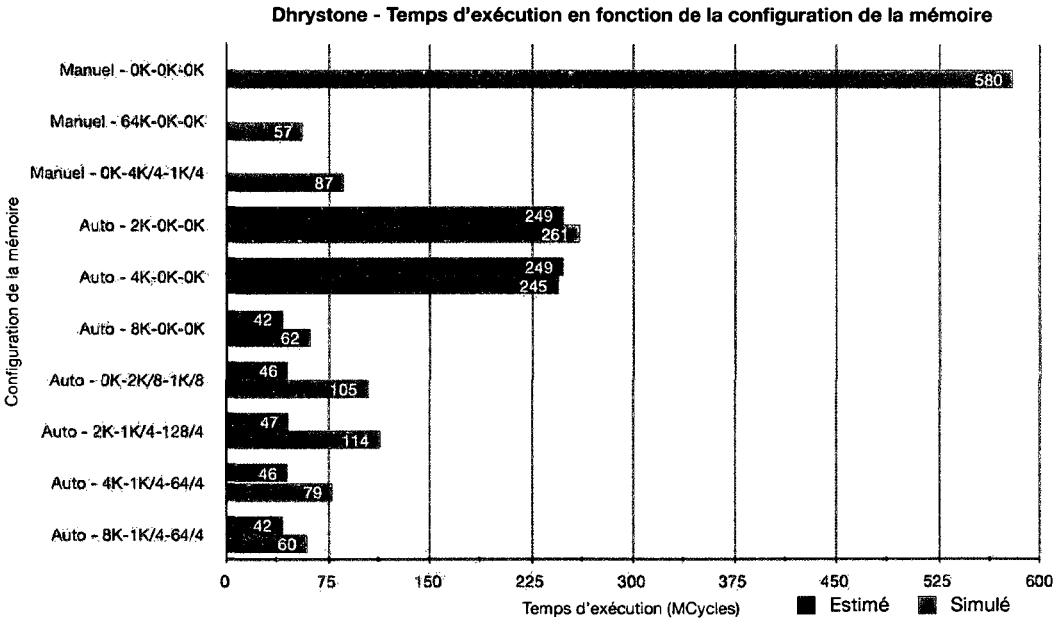

Figure 5.1 Temps d'exécution en fonction de la configuration de la mémoire pour Dhrystone

Le deuxième groupe de résultats montre les résultats lorsque seul le partitionnement de mémoire est activé. Il est donc possible de voir la performance lorsque 2 Ko, 4 Ko et 8 Ko de mémoire locale sont utilisés. La meilleure performance est atteinte lorsque 8 Ko de mémoire locale sont utilisés. Avec cette taille, le critère du pourcentage de couverture (90%) est respecté et même dépassé légèrement (voir l'annexe C). Le nombre de cycles d'exécution passe donc de 580 millions à 62 millions, ce qui représente une accélération de 9,35. Cette performance est très rapprochée de la performance atteinte lorsque le programme est entièrement contenu en mémoire locale, la différence de performance n'étant que de 8.8%.

L'utilisation d'une mémoire cache permet d'améliorer la performance de façon significative lorsque la taille de la mémoire locale n'est pas optimale. En effet, la configuration Auto – 2K-1K/4-128/4 permet d'améliorer la performance par un facteur de 2,3 par rapport à la configuration Auto – 2K-0K-0K. Pareillement, la configuration Auto – 4K-1K/4-64/4 permet d'améliorer la performance par un facteur de 3,1 par rapport à la configuration Auto – 4K-0K-0K. Par contre, ce gain devient négligeable pour la configuration Auto – 8K-1K/4-64/4 qui passe alors à 60 millions de cycles par rapport à la configuration Auto – 8K-0K-0K qui nécessite 62 millions de cycles. Ultimement, il revient à l'utilisateur de déterminer si ces 2 millions de cycles sont critiques. À noter que 8 Ko de mémoire locale est la taille minimale pour atteindre le pourcentage de couverture de 90%, à partir de ce moment, la taille de la mémoire locale n'augmente plus même si la taille maximale allouée est augmentée.

Jusqu'à maintenant, les observations se sont concentrées sur les performances en simulation. Par contre, une fonctionnalité importante de l'outil est de tenter de prédire la performance de l'application. En observant les résultats du nombre de cycles estimés par rapport au nombre de cycles en simulation, il est évident que l'algorithme d'estimation de la performance ne rencontre par les attentes. Les temps d'exécution estimés pour la

mémoire locale sont assez rapprochés de la réalité, par contre, les temps estimés lorsque la mémoire cache entre en jeu sont très éloignés des temps en simulation.

Une partie de l'explication se retrouve dans la méthode de recueil de la trace. En effet, lors du recueil de trace, les instructions récupérées, mais non exécutées, ne sont pas enregistrées. Ceci peut donc biaiser les résultats en donnant une meilleure performance puisque moins d'accès sont pris en compte. La deuxième partie de l'explication vient de la façon dont la simulation de cache fonctionne lors du processus d'optimisation. Lors de l'exécution d'un programme, le cache doit être activé en logiciel par le programme, or, cela ne peut être fait qu'au plus tôt dans la fonction *main()* et donc après le code d'initialisation du moteur d'exécution du langage C. Également, la désactivation du cache se fait immédiatement avant la sortie du *main()* et donc avant l'exécution du code de nettoyage du moteur d'exécution. Le simulateur de cache ne tient pas compte de l'état d'activation du cache, il considère que celle-ci est toujours activée. Ce comportement, combiné à l'explication précédente et le fait qu'en pratique, certains symboles ne peuvent être relocalisés (voir section 4.2.6), peut donc biaiser les résultats de façon significative.

Les derniers résultats significatifs concernent la taille estimée du tas et de la pile. Dans la version initiale de l'application, le tas et la pile occupaient chacun un espace de 2 Ko, ce qui semblait raisonnable puisque le programme utilise un bon nombre de variable locale, d'appels de fonctions imbriquées et une certaine utilisation de l'allocation dynamique. Ces tailles permettaient donc d'assurer le bon fonctionnement du programme sans toutefois être minimales. Par contre, après l'exécution de l'étape de partitionnement de mémoire, l'algorithme suggérait une taille de 192 octets pour le tas et de 640 octets pour la pile. L'espace économisée ne semble peut être pas énorme, mais les nouvelles tailles permettent à l'algorithme de relocaliser complètement le tas et la pile dans la mémoire locale et donc d'accélérer significativement l'accès aux données.

Tableau 5.1 Tailles suggérées du tas et de la pile pour Dhrystone

	Taille (octets)
Tas	192
Pile	640

5.2.2. IDCT

Les résultats (Figure 5.2) pour l'IDCT sont similaires à ceux obtenus pour Dhrystone avec quelques différences. Encore une fois, on observe un gain de 10 fois lorsque le programme est complètement contenu en mémoire locale comparativement à la configuration en mémoire externe. Lorsque le cache est utilisé pour accélérer la performance du programme localisé en mémoire externe, le temps d'exécution diminue d'un facteur de 6,9.

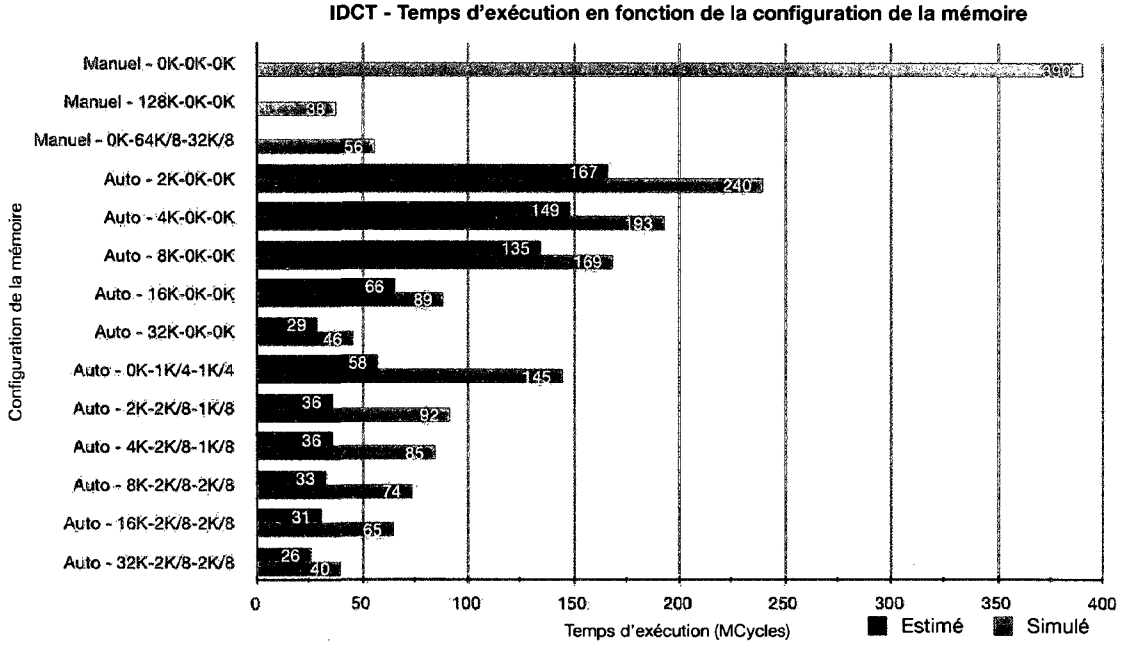

Figure 5.2 Temps d'exécution en fonction de la configuration de la mémoire pour IDCT

L'utilisation du partitionnement de mémoire permet de réduire d'un facteur de 4 l'utilisation de la mémoire locale (Auto - 32K-0K-0K) tout en obtenant des performances

très proche de la configuration présentant la meilleure performance (Manuel - 128K-0K-0K). Cette réduction de l'utilisation de la mémoire locale n'est pas aussi importante que pour Dhrystone, qui affichait une réduction d'un facteur de 8, mais cela reste une bonne amélioration, compte tenu de la performance atteinte.

L'algorithme d'optimisation de la mémoire cache suggère un cache d'instructions de 1 Ko et un cache de données de 1 Ko comme meilleur compromis entre la performance et la taille des caches. Avec cette configuration, on obtient un modeste gain de vitesse de 2,7. Si le concepteur n'est pas satisfait de ce choix, il est libre d'examiner les tableaux de résultats pour chaque configuration de cache, correspondant ici au Tableau 5.23 et au Tableau 5.24 en annexe, et décider de la configuration qui lui semble être la mieux adaptée.

Comme c'était le cas pour Dhrystone, l'utilisation de la taille optimale de mémoire locale déterminée par le partitionnement combinée à la configuration optimisée de mémoire cache (Auto - 32K-2K/8-2K/8) permet de rejoindre la performance obtenue avec la configuration Manuel - 128K-0K-0K. Par contre, il est possible de remarquer que l'utilisation de la mémoire cache dans ce cas ne fait qu'économiser 6 millions de cycles. Bien que ceci représente un gain de vitesse de 13%, il revient au concepteur de décider si cela en vaut la peine.

Lorsque la taille de la mémoire locale n'est pas optimale, la situation en est autrement. Contrairement à la configuration Auto - 32K-2K/8-2K-8, l'utilisation du cache pour la configuration Auto - 16K-2K/8-2K/8 permet d'économiser 24 millions de cycles, soit 27%. Cette configuration permet alors d'atteindre 61% de la performance de la configuration Auto - 32K-2K/8-2K/8 avec une mémoire deux fois plus petite. Dans un système multiprocesseur, ce compromis peut être intéressant.

Une dernière observation peut être tirée du graphique, celle-ci reliée aux temps d'exécution estimés. Comme il était le cas pour Dhrystone, les temps estimés pour les configurations avec cache sont très différents des temps simulés. Par contre, contrairement à Dhrystone, les temps estimés pour les configurations où seul le partitionnement de mémoire entre en jeu sont également très optimistes. Ceci s'explique par le fait que beaucoup de symboles ne peuvent être déplacés en mémoire locale lors de la recompilation du programme et donc le temps estimé est sous-évalué par rapport au temps simulé.

Pour ce qui est de la taille suggérée du tas et de la pile, l'algorithme suggère un tas de 0 octet et une pile de 1344 octets. La taille nulle du tas s'explique par le fait que la plateforme logicielle SPACE ne supporte pas pour l'instant l'allocation dynamique de mémoire, le tas reste donc inutilisé durant toute l'exécution.

Tableau 5.2 Tailles suggérées du tas et de la pile pour IDCT

	Taille (octets)
Tas	0
Pile	1344

5.3. Efficacité des algorithmes de compression de trace

L'objectif des algorithmes de compression des fichiers de trace est de réduire la taille des fichiers mais aussi de réduire le temps d'exécution du partitionnement et de l'optimisation du cache en réduisant le temps passé à lire les traces. Cette dernière affirmation relève de l'hypothèse que le processus de lecture de la trace a un impact significatif sur le temps total d'exécution des algorithmes d'exploration architecturale.

5.3.1. Efficacité de la compression sur la taille des traces

Comme le démontre le Tableau 5.3, la compression de la trace de cache permet de réduire la taille de la trace par près du tiers pour les traces d'instructions. Cette réduction

est possible grâce à la localité de référence élevée des instructions. Cela n'est pas le cas avec les traces de cache des données. En effet, la réduction est beaucoup moins importante, 71% et 86%.

Si l'on compare ces résultats avec ceux obtenus pour les traces de mémoire, on observe que ces dernières sont beaucoup plus efficaces pour compresser les accès aux données. L'algorithme de compression des traces de cache n'est donc pas aussi performant que prévu en ce qui concerne la réduction de la taille de la trace de données. En somme, les deux méthodes sont appropriées au travail à accomplir mais elles mériteraient de s'y attarder afin d'améliorer leur efficacité en terme de réduction de taille.

Tableau 5.3 Taille moyenne relative des traces compressées par rapport aux traces non-compressées

Application	Trace	Instructions (%)	Données (%)
Dhrystone	Mémoire	49,29	49,56
	Cache	30,37	71,41
IDCT	Mémoire	45,71	59,79
	Cache	35,43	86,63

5.3.2. Efficacité de la compression sur le temps d'exécution

Le deuxième objectif de la compression est de réduire le temps d'exécution de l'exploration architecturale, aussi appelé le temps d'analyse. Comme mentionné plus haut, l'hypothèse est que si la taille de la trace est réduite, le temps de lecture sur le disque sera réduit, ce qui aura un impact sur le temps total d'exécution de l'algorithme d'exploration architecturale.

Tout d'abord, on peut remarquer dans la Figure 5.3 et la Figure 5.4 que la taille de la trace de mémoire compressée a un impact négligeable sur le temps d'exécution de l'algorithme de partitionnement. L'algorithme de compression de la trace de mémoire permet donc uniquement de réduire l'espace utilisé sur le disque et ne permet pas de réduire le temps d'exécution de l'algorithme de partitionnement de façon significative. Il

est donc possible de conclure que la lecture de la trace n'est pas un goulot d'étranglement en ce qui concerne le partitionnement de mémoire.

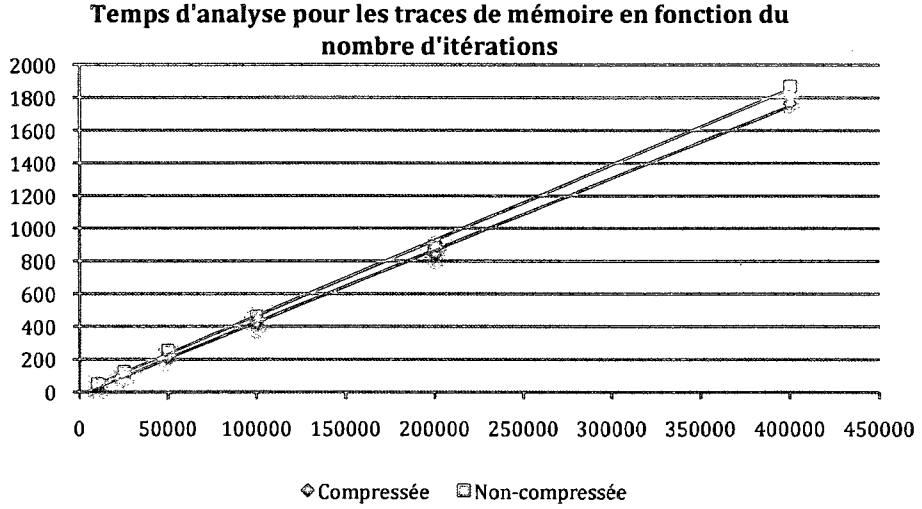

Figure 5.3 Temps d'analyse pour la trace de mémoire en fonction du nombre d'itérations pour Dhrystone

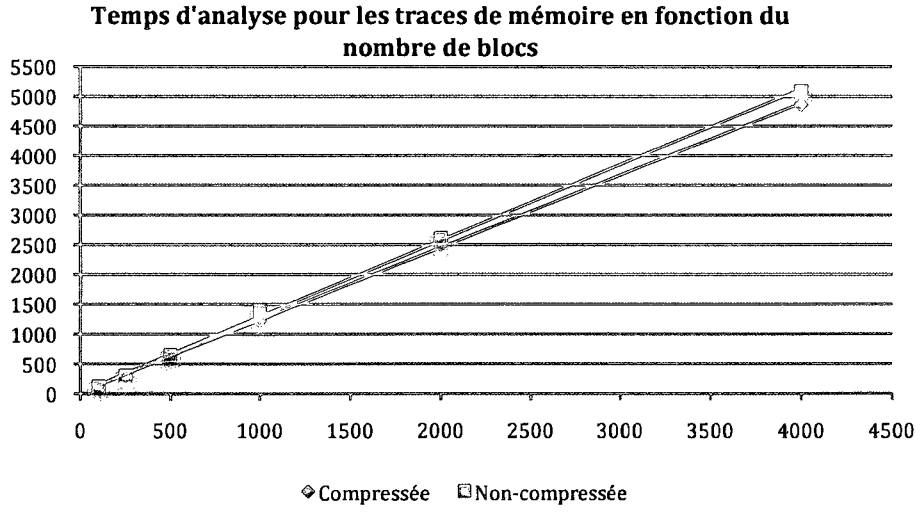

Figure 5.4 Temps d'analyse pour la trace de mémoire en fonction du nombre de blocs

En ce qui concerne l'algorithme de compression de la trace de cache, les résultats sont plus encourageants. En effet, la trace compressée permet d'économiser beaucoup de temps comme le démontrent les graphiques à la Figure 5.5 et à la Figure 5.6.

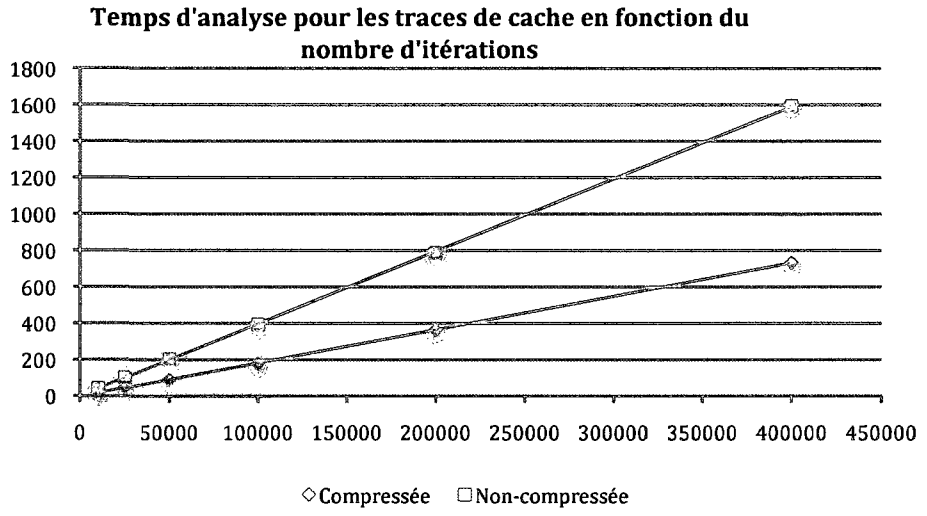

Figure 5.5 Temps d'analyse pour la trace de cache en fonction du nombre d'itérations

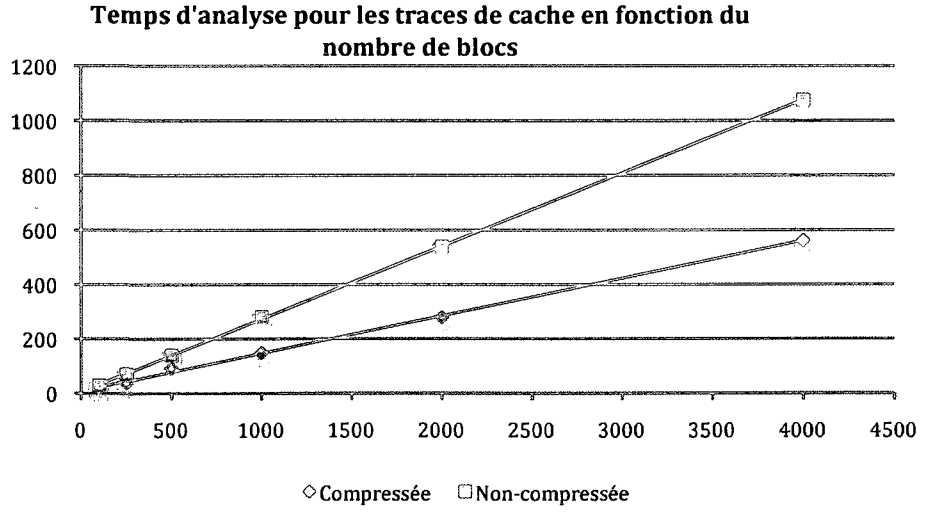

Figure 5.6 Temps d'analyse pour la trace de cache en fonction du nombre de blocs pour IDCT

Pour ces deux graphiques, on remarque que le temps économisé augmente rapidement selon la longueur de la simulation. Ceci est dû principalement au fait que l'algorithme de compression permet de regrouper plusieurs accès à la même ligne de cache en un seul accès. L'algorithme de simulation des caches peut donc utiliser directement ce nombre d'accès afin de simuler d'un coup ces accès. Ceci contraste avec la méthode de compression de la trace de mémoire qui ne peut effectuer ce genre de traitement et qui doit donc lire et traiter chaque accès indépendamment.

5.4. Extensibilité à d'autres processeurs/plateformes de simulation

L'outil d'exploration architecturale a été conçu de façon à être flexible en terme de processeurs pouvant être supportés. Par contre, le support d'une nouvelle plateforme de simulation est plus complexe.

Puisqu'il est nécessaire d'intégrer l'outil de trace d'exécution dans un simulateur d'instructions, il est essentiel d'avoir une plateforme de simulation en langage C/C++ afin de pouvoir réutiliser le code de trace existant. De plus, il est nécessaire d'avoir accès au code source de l'ISS afin d'intégrer la trace dans ce dernier. Dans le cas où la plateforme n'est pas en C/C++, il est alors nécessaire de concevoir un nouvel outil de trace qui respecte le format de fichier. L'inaccessibilité au code source de l'ISS pose un problème plus important puisqu'il devient alors impossible d'intégrer l'outil de trace à ce dernier. Si l'ISS possède des signaux de débogage, il est alors possible de s'y connecter et de tracer l'exécution, mais si l'ISS ne possède pas ces signaux et ne fournit aucune méthode de trace, il devient alors très difficile, voire impossible, de recueillir la trace. L'outil d'exploration architecturale ne pourra donc pas fonctionner.

De plus, l'outil de trace assume que le processeur est un processeur 32 bits. Ceci a des répercussions en ce qui a trait au stockage des adresses accédées dans la trace. Par

exemple, pour un processeur 64 bits, il devient nécessaire de stocker des adresses de 64 bits dans la trace et non des adresses de 32 bits. Une seconde modification importante est liée à la compression de la trace de cache. Puisque l'algorithme de compression de la trace de cache assume que la taille minimale des lignes de cache est de 4 mots de 32 bits, donc 128 bits, un problème survient lorsque cette taille de ligne de cache n'est pas supportée par le processeur ou si la taille minimale n'est pas de 4 mots.

Du côté du partitionnement de mémoire, l'algorithme ne requiert aucune modification majeure si le processeur respecte l'architecture assumée par l'algorithme. Le PowerPC, le processeur Xtensa ainsi que le NIOS-II [63] respectent cette architecture. Il est bien sûr nécessaire de modifier les temps d'accès, les tailles de mémoires possibles ainsi que quelques autres paramètres, mais les changements restent mineurs.

Des changements importants sont par contre nécessaires pour l'algorithme d'optimisation de la mémoire cache. En effet, la plupart des processeurs embarqués ne sont pas limités à un cache à accès direct (*direct-mapped)* et supportent plutôt des caches de type associatifs par ensembles (*set-associative)*. C'est le cas pour le PowerPC et le Xtensa qui supportent soit un degré d'associativité fixe pour le PowerPC ou configurable pour le Xtensa. Certains processeurs plus avancés implémentent des caches supplémentaires, comme des caches de niveau 2 ou 3 et ou des caches de victimes (*victims caches)* [64]. Ces architectures ne sont pas supportées par l'algorithme d'optimisation et, par conséquent, requièrent un algorithme diffèrent. Les systèmes implémentant la cohérence de cache [13] ne sont pas non plus supportés puisque le contenu du cache peut être modifié par un autre processeur.

Finalement, un dernier obstacle au support d'autres processeurs, ce qui inclut le MicroBlaze version 7, est le support pour la mémoire virtuelle [13]. En effet, les algorithmes n'ont aucune connaissance de la mémoire virtuelle et leur comportement sera donc erratique si celle-ci est employée. De plus, l'implémentation de la mémoire

virtuelle varie d'un processeur à un autre, ce qui complique davantage les modifications à apporter aux algorithmes.

5.5. Comparaison avec d'autres travaux

Bien que l'algorithme de partitionnement permette d'obtenir de bons résultats, il n'est pas parfait et peut être amélioré. Les travaux présentés dans [30] peuvent être particulièrement utiles car ils touchent le partitionnement de mémoire avec une granularité plus fine que la méthode utilisée dans MemoryOptimizer. En effet, la plus petite unité sur laquelle MemoryOptimizer peut travailler pour le partitionnement de mémoire est le symbole qui peut être une fonction ou une donnée. Ceci pose problème lorsque l'algorithme rencontre des fonctions ou des données volumineuses, celles-ci doivent être complètement relocalisées en mémoire locale. Souvent, une fonction contenant une boucle peut passer beaucoup de temps à exécuter cette boucle et peu de temps à exécuter le reste de la fonction. Il devient alors utile de pouvoir isoler cette boucle et de la relocaliser en mémoire locale.

Du côté de l'optimisation du cache, la technique utilisée dans MemoryOptimizer est simple mais efficace. Par contre, plusieurs améliorations peuvent être apportées à l'algorithme. Tout d'abord, la compression de la trace de cache peut être grandement améliorée en adoptant l'algorithme de compression par échantillonnage présenté dans [37]. Ceci permettrait de réduire la taille de la trace mais aussi d'améliorer la vitesse de simulation des caches.

Également, les méthodes proposées dans la littérature ne se limitent pas à optimiser des mémoires cache *à accès direct* mais permettent également de simuler des caches associatifs. Par exemple, l'algorithme présenté dans [39] pourrait permettre d'accélérer la simulation de ce type de cache. Selon l'algorithme, il nécessaire de modéliser le comportement d'un cache associatif par ensembles en plus d'un cache *à accès direct*.

Comme MemoryOptimizer modélise déjà les caches *à accès direct*, le travail à accomplir est quelque peu réduit.

Il existe également une méthode permettant de minimiser les échecs en relocalisant certains bouts de code à des emplacements différents dans une même mémoire. Cette méthode, présentée dans [28], pourrait être particulièrement utile afin de réduire davantage la taille de la mémoire cache.

Pour ce qui est de la performance des configurations résultantes de l'exécution de l'outil de partitionnement de mémoire et d'optimisation de cache, elles se comparent favorablement à la méthode proposée dans [29]. La méthode proposée par les auteurs permet d'obtenir des gains de performance de 7 à 8x alors que MemoryOptimizer permet d'obtenir des gains de performance de près de 10x. Il n'est par contre pas possible de comparer directement ces accélérations puisque les architectures matérielles utilisées et les applications de test sont différentes dans le deux cas. En revanche, il est possible de conclure que les accélérations obtenues sont du même ordre de grandeur et ce malgré le fait que MemoryOptimizer utilise des algorithmes d'optimisation beaucoup plus simples que ceux présentés par Panda, Dutt et Nicolau. Par contre, la méthode d'estimation de performance présentée dans leur travail est supérieure à celle de MemoryOptimizer. Il serait donc approprié de tenter d'adapter leur méthode d'estimation à MemoryOptimizer.

5.6. Rencontre des requis

L'outil de partitionnement de mémoire et d'exploration de la mémoire cache MemoryOptimizer doit rencontrer un nombre de requis tel qu'énoncé à la section 1.3. La plupart de ces requis sont évalués de façon qualitative puisque seules les mesures de performance peuvent être évaluées quantitativement.

L'objectif principal de l'outil est de permettre d'effectuer automatiquement le partitionnement de mémoire et l'optimisation de la mémoire cache dans un système sur puce. Les résultats de l'optimisation de la hiérarchie de mémoire pour deux applications présentées à la section 5.2 sont concluants. Il est donc possible de déterminer automatiquement le contenu de la mémoire locale ainsi que minimiser la taille de cette dernière. De plus, la configuration de cache la plus appropriée est déterminée pour les caches d'instructions et de données.

L'utilisation de l'outil afin d'optimiser la hiérarchie est également plus rapide que la méthode manuelle. En effet, l'optimisation de la configuration des mémoires cache peut nécessiter plusieurs dizaines de simulations pour déterminer la configuration optimale alors qu'il n'en faut qu'une pour permettre à l'outil de déterminer automatiquement la performance de chaque configuration de cache. Le temps d'exécution de l'outil est environ 5 fois plus rapide que la simulation comme le démontre le Tableau 5.4.

Tableau 5.4 Temps de simulation et d'exploration pour Dhrystone

Nb Itérations	Temps d'exécution de la simulation (s)	Temps d'exécution de l'outil d'exploration architecturale (s)
10000	287,3	41,4
25000	596,0	112,3
50000	1209,6	203,8
100000	2562,5	410,0
200000	5131,2	837,1
400000	10420,2	1766,0

L'outil d'optimisation est relativement indépendant de la plateforme de simulation. En effet, il ne nécessite que des fichiers de trace, le fichier ELF ainsi que les programmes permettant d'effectuer la lecture du fichier ELF. La plateforme de simulation doit également permettre de respecter la configuration assumée par l'outil, c'est-à-dire la configuration processeur, mémoire locale, mémoire cache, bus et mémoire externe. Il suffit donc principalement de porter l'outil de trace d'exécution sur une autre plateforme et l'outil d'optimisation pourra alors être utilisé.

L'outil de trace peut facilement être intégré à d'autres processeurs. Comme il fut énoncé à la section précédente, certains ajustements sont nécessaires si le processeur n'est pas 32 bits et si la taille minimale des lignes de cache n'est pas de 4 mots. Mis à part ces modifications, il est possible d'utiliser facilement l'outil de trace à l'aide de l'API fourni.

L'interface graphique permet de simplifier le processus d'exploration architecturale. Tout d'abord, l'utilisateur n'a pas à entrer les paramètres en mode texte mais peut utiliser des artefacts graphiques connus comme des boîtes de texte ou des menus déroulants. De plus, l'interface graphique permet de visualiser les résultats sous forme de graphiques, ce qui permet à l'utilisateur d'avoir une vue d'ensemble facile à comprendre.

L'intégration dans la plateforme SPACE ne devrait poser que peu de problèmes. Le modèle du MicroBlaze peut être utilisé tel quel dans la librairie SPACE. Par contre du côté de SpaceStudio, il est nécessaire de concevoir les mécanismes qui permettront de configurer le MicroBlaze et le cache. Les liens XCL doivent également être supportés par l'environnement de développement.

L'intégration de l'interface graphique peut se faire assez aisément. Les API et les classes sont bien documentés et l'interface graphique constitue en elle-même un exemple d'intégration des classes **CMemoryEvaluator** et **CCacheEvaluator** dans un environnement. Par contre, si un grand degré d'intégration entre SpaceStudio et l'interface graphique est requis, il pourrait devenir nécessaire de recoder plusieurs parties de l'interface graphique afin de l'informer automatiquement de la configuration du système modélisé. Ceci permettrait de minimiser les données devant être spécifiées par l'utilisateur lorsque celui-ci désire optimiser son système.

CONCLUSION ET TRAVAUX FUTURS

Avec la tendance à l'intégration d'un nombre grandissant de processeurs dans les SoC, l'utilisation efficace de la mémoire intégrée à ces SoC devient un enjeu important afin de maintenir la performance. De nouvelles méthodes permettant d'exploiter au maximum le potentiel de ce type de mémoire sont donc nécessaires et les travaux effectués dans ce domaine le prouvent.

Les différents concepts reliés à ce domaine ont d'abord été exposés. Les concepts de mémoire locale et de caches ont été expliqués, suivis des différentes méthodes d'optimisations publiées dans la littérature. Par contre, ces méthodes se concentrent majoritairement sur l'optimisation de la mémoire locale ou de la mémoire cache. Un seul ouvrage proposait une méthodologie permettant d'optimiser les deux types de mémoire.

Un modèle du processeur MicroBlaze, intégré à la plateforme SPACE, a été présenté. Ce modèle de processeur permet d'évaluer de façon plus précise le comportement du système que l'ISS déjà existant dans SPACE. À ce modèle de processeur, un mécanisme de recueil de trace a été adjoint. Cette trace permet de connaître l'ensemble des accès à la mémoire effectués par le processeur et forme le point de départ de l'outil d'exploration architecturale de la hiérarchie de la mémoire.

Par la suite, l'outil MemoryOptimizer a été introduit. Cet outil, se basant sur des travaux antérieurs, permet d'effectuer à la fois un partitionnement de mémoire et d'optimiser la configuration des mémoires caches. Ainsi, la taille de la mémoire utilisée sur le FPGA est réduite tout en offrant de meilleures performances que lorsque la mémoire externe seule est utilisée. De plus, cette réduction de taille permet au concepteur d'intégrer plus de processeurs sur un MPSoC tout en maintenant une bonne performance.

Pour une application donnée, l'utilisation du partitionnement de mémoire et de l'optimisation du cache permet de réduire par un facteur de près de 8 la taille de la mémoire intégrée utilisée. De plus, cette réduction de la taille de la mémoire se fait avec un impact négligeable sur la performance puisque cette performance ne diminue que de 5%. L'outil MemoryOptimizer permet donc d'utiliser de façon efficace la mémoire intégrée en tirant profit d'une combinaison de mémoire locale et de mémoire cache.

Travaux futurs

La présence de l'outil MemoryOptimizer ouvre un grand nombre de possibilités, tant à la plateforme SPACE que de façon indépendante. Il devient maintenant possible de déterminer automatiquement les tailles optimales des caches ainsi que le contenu et la taille de la mémoire locale. Parmi les travaux d'intégration on retrouve :

1. Intégration de l'ISS et de l'outil de trace dans la librairie SPACE et utilisation de MemoryOptimizer indépendante de SpaceStudio. MemoryOptimizer n'est pas intégré à SpaceStudio et opère de façon indépendante.
2. Intégration de l'ISS, de l'outil de trace dans la librairie SPACE et intégration complète de MemoryOptimizer dans SpaceStudio. MemoryOptimizer peut donc accéder à la configuration du système et il devient alors possible de reconfigurer automatiquement le système après l'exploration architecturale.
3. Port de l'outil de trace et de MemoryOptimizer afin de supporter un processeur différent ou une plateforme de simulation autre que SPACE.
4. Combinaison des algorithmes d'exploration architecturale avec les travaux de partitionnement automatique de Laurent Moss [65].

L'outil même n'est par contre pas sans faute et plusieurs améliorations peuvent être apportées. Des améliorations au niveau algorithmique ont été discutées au chapitre 5 mais d'autres améliorations peuvent également être apportées à l'outil. Parmi ces

améliorations, il y a d'abord l'évaluation et le perfectionnement de la division du travail en tâches. En effet, bien que les calculs aient été divisés en tâches, l'effet de cette division n'a pas été étudié. À la lumière de cette étude, il serait donc possible de caractériser le comportement et d'améliorer cette division des calculs. L'algorithme d'optimisation de la mémoire cache pourrait tout particulièrement bénéficier d'une telle amélioration puisque pour l'instant, seules deux tâches sont utilisées afin de paralléliser la simulation des caches.

Il serait également très intéressant d'explorer des hiérarchies de mémoire plus complexes que celles supportées par l'outil. Une hiérarchie employant une mémoire cache unifiée de deuxième niveau serait particulièrement intéressante à explorer. Dans la même avenue, une architecture comportant une mémoire locale connectée au bus au lieu d'être connectée directement au processeur, serait également intéressante puisque cela signifierait que le cache serait en moyenne plus rapide que cette mémoire.

Il serait également possible d'approcher le problème sous un autre angle. Au lieu d'effectuer d'abord le partitionnement avec une limite de mémoire et ensuite l'optimisation de la mémoire cache, une approche alternative serait d'effectuer une exploration architecturale sur les deux types de mémoire à la fois et l'utilisateur pourrait spécifier une taille globale maximale de mémoire à utiliser. L'algorithme pourrait alors déterminer la taille optimale de mémoire locale et de mémoire cache en respectant cette taille. Cette approche rejoint les travaux réalisés par Panda, Dutt et Nicolau dans [28].

Ensuite, l'interface graphique pourrait être grandement modifiée afin de la rendre plus conviviale à utiliser. Il serait donc possible de simplifier les options présentées à l'utilisateur et d'implémenter des fonctionnalités pouvant lui simplifier la vie comme la sauvegarde des résultats dans un fichier.

Finalement, un dernier travail, très important, à effectuer serait d'évaluer la performance des configurations générées par l'outil dans un système sur FPGA. Dans le présent travail, les configurations n'ont été testées qu'en simulation. Bien que tous les efforts possibles on été faits afin de s'assurer de la précision des résultats en simulation, il reste qu'il est impératif de tester ces configurations sur un système réel.

RÉFÉRENCES

[1] Embedded.com. Under the Hood: Robot Guitar embeds autotuning [En ligne], 2007. http://www.embedded.com/underthehood/207401418 (page consultée le 28 septembre 2008).

[2] Embbeded Tech Journal. Mixing Fossil Fuel and Electrons [En ligne]. 2007. http://www.embeddedtechjournal.com/articles_2007/20070313_fossil.htm (page consultée le 1er octobre 2008).

[3] Embedded.com. The Two Percent Solution [En ligne], 2002, http://www.embedded.com/story/OEG20021217S0039 (page consultée le 28 septembre 2008).

[4] Netrino. Embedded Systems Glossary: E | Netrino [En ligne]. 2008. http://www.netrino.com/Embedded-Systems/Glossary-E (page consultée le 1er octobre).

[5] G. Moore, Cramming more components onto integrated circuits, Electronics, Vol. 38, No. 8, April 1965.

[6] G. Moore, Progress in Digital Integrated Electronics [En ligne], 1975, http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1975_Speech.pdf (page consultée le 13 octobre).

[7] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[8] Intel. "Intel Demonstrates Industry's First 32nm Chip and Next-Generation Nehalem Microprocessor Architecture", Press Release, Intel Developer Forum, 18 septembre 2007.

[9] Freescale Semiconductor, Inc. "DSP56301 24-Bit Digital Signal Processor Technical Data Sheet", Freescale Semiconductor, Inc, 2006.

[10] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, D. Shippy, Introduction to the Cell multiprocessor. IBM Journal of Research and Development, Vol. 49, No. 4/5, July/September 2005.

[11] M.D. Hill, "A case for direct-mapped caches,". *IEEE Computer*, 21(12):25-41, December 1988.

[12] N.P. Jouppi. "Cache write policies and performances,". In *International Symposium on Computer Architecture*, pages 191-201, San Diego, CA, May 1993.

[13] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kauffman, 2003.

[14] Xilinx, Inc. «MicroBlaze Processor Reference Guide : Embedded Development Kit 9.1i». Xilinx, Inc. 2007.

[15] P. Genua, "A Cache Primer", Freescale Semiconductor, Inc., Austin, Texas, 2004.

[16] Sun Microsystems, Inc., "Memory Hierarchy in Cache-Based Systems", Sun Microsystems, Inc., Santa Clara, California, November 2002.

[17] D. Novillo, From Source to Binary : The Inner Workings of GCC [En ligne]. http://www.redhat.com/magazine/002dec04/features/gcc/ (page consultée le 20 novembre 2007).

[18] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1993.

[19] GNU, GNU Compiler Collection Internals: Dividing the Output into Sections [En ligne], 2003, http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gccint/Sections.html (page consultée le 9 octobre 2008).

[20] GNU, C Extenstions – Using the GNU Compiler Collection (GCC) [En ligne], 2008, http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html (page consultée le 9 octobre 2008).

[21] J. Levine,. *Linkers and Loaders*. Morgan Kauffman, 2000.

[22] Sun Microsystems, Inc. Linker and Libraries Guide [En ligne], 2008, http://dlc.sun.com/pdf/817-1984/817-1984.pdf (page consultée le 9 octobre 2008).

[23] GNU, LD [En ligne], 2008, http://sourceware.org/binutils/docs/ld/ (page consultée le 9 octobre 2008).

[24] Free Standards Group. Generic ELF Specification [En ligne], 2007, http://www.linux-foundation.org/spec/book/ELF-generic/ELF-generic/book1.html (page consultée le 9 octobre 2008).

[25] M. Dunlavey, Peformance Tuning with Instruction-Level Cost Derived from Call-Stack Sampling. *ACM Sigplan Notices*, 42(8) :4-8, 2007.

[26] D.C. Suresh, W.A. Najjar, F. Vahid, J.R. Villarreal, and G. Stitt. "Profiling Tools for Hardware/Software Partitionning of Embedded Applications," *SIGPLAN Not. 38, 7,* p 189-198, July 2003.

[27] K. Pettis and R.C. Hansen. Profile guided code positioning. In PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, pages 16-27, 1990.

[28] P. Panda, N. Dutt, et A. Nicolau. Memory Issues in Embedded Systems-On-Chip. Kluwer Academic Publishers. 1999.

[29] P. Panda, N. Dutt, and A. Nicolau. "Architectural Exploration and Optimization of Local Memory in Embedded Systems," *System Synthesis, 1997. Proceedings., Tenth International Symposium on*, pp. 90-97, 17-19 Sep 1997.

[30] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, "A Post-Compiler Approach to Scratchpad Mapping of Code," *Compilers, Architecture, and Synthesis For Embedded Systems 2004, Proceedings of the 2004 International Conference on*, pp. 259-267, September 2004.

[31] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pages 115-125, 2005.

[32] C. Park, J. Lim, K. Kwon, J. Lee, and S.L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In EMSOFT'04: The ACM Conference on Embedded Software, September 2004.

[33] B. Egger, J. Lee, and H, Shin, "Scratchpad Memory Management for Portable Systems with a Memory Management Unit", Proceedings of the 2006 IEEE International Conference on Embedded Software, pp. 321-330, October 2006.

[34] GNU, Basic Blocks – GNU Compiler Collection (GCC) Internals [En ligne], 2008, http://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html (page consultée le 15 octobre 2008).

[35] S. Fontaine, S. Goyette, J.M.P. Langlois, et G. Bois, Acceleration of a 3D Target Tracking Algorithm Using an Application Specific Instruction Set Processor. *Proceedings of the IEEE International Conference on Computer Design 2008*, 2008.

[36] R. L. Mattson, J. Gecsei, D.R. Slutz, and I. L. Traiger. "Evaluation techniques for storage hierarchies". IBM System Journal, 9(2):78-117, 1970.

[37] S. Laha, J. Patel et R. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, 37(11) : 1325-1336. 1988.

[38] Z. Wu et W. Wolf. Iterative Cache Simulation of Embedded CPUs with Trace Stripping. *Proceedings of the Seventh International Workshop on Hardware/Software Codesign 1999*. P. 95-99. 1999.

[39] W.-H. Wang et J.-L. Baer. Efficient Trace-Driven Simulation Methods for Cache Performance Analysis. *ACM Transactions on Computer Systems*, 9(3) : 222-241. 1991.

[40] A. Silva, G. Esmeraldo, P. Viana, E. Barros. Cache-Analyzer : Design Space Evaluation of Configurable-Caches in a Single-Pass. *18th IEEE International Workshop on Rapid System Prototyping*, p. 3-9. 2007.

[41] J. Edler et M. D. Hill, "Dinero IV trace-driven uniprocessor cache simulator," Computer Architecture News.

[42] S.J.E. Wilton, N.P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid State Circuits, vol. 31, no. 5, 1996.

[43] J.L.D. Mendes, L.M.N. Coutinho, and C.A.P.S. Martins, "Mcsim – multilevel and split cache simulator," 26th Frontiers in Education Conference (FIE), 2006.

[44] L. Filion, M.-A. Cantin, L. Moss, E. M. Aboulhamid, and G. Bois, "Space Codesign: A SystemC Framework for Fast Exploration of Hardware/Software Systems", in *Design & Verification Conference and Exhibition (DVCON'07)*, San Jose, CA, 2007.

[45] SystemC. SystemC User's Guide [En ligne], 2008. http://www.systemc.org. (Page consultée le 15 octobre 2008).

[46] Xilinx, "Fast Simplex Link (FSL) Bus v2.00a", Datasheet, Xilinx, 2005.

[47] Xilinx, "Fabric Co-processor Bus (FCB) (v1.00a)", Datasheet, Xilinx, 2005.

[48] Xilinx, "PowerPC 405 Processor Block Reference Guide – Embedded Development Kit EDK 9.1i", Xilinx, 2007.

[49] Xilinx, "MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 9.1i", Xilinx, 2007.

[50] Xilinx, "PicoBlaze 8-bit Embedded Microcontroller User Guide", Xilinx, 2008.

[51] IBM Microelectronics, "128-Bit Processor Local Bus 4.7: Architecture Specifications", IBM Corp., Mai 2007.

[52] IBM Microelectronics, "On-Chip Peripheral Bus 2.1 : Architecture Specifications", IBM Corp., Avril 2001.

[53] Xilinx. "Instruction-Side OCM Bus v1.0 (v2.00b)", Datasheet, Xilinx, 2007.

[54] Xilinx, "Local Memory Bus (LMB) V1.0 (v1.00a)", Datasheet, Xilinx, 2007.

[55] Nurmi, J.; Leibson, S.; Campi, F.; Panis, C., "Extensible and Configurable Processors for System-on-Chip Design," *Advanced Signal Processing, Circuits, and System Design Techniques for Communications, 2006* , pp.45-97, May 2006

[56] Xilinx, "MCH OPB Synchronous DRAM (SDRAM) Controller (v1.00a)", Datasheet, Xilinx, 2006.

[57] S. Fontaine, L. Filion, G. Bois, "Exploring ISS Abstractions in Embedded Software Design", *Digital System Design Architectures, Methods and Tools, 2008. DSd 2008. Proceedings of the 10th Euromicro Conference on*, pp. 651-655, 3-5 Sept. 2008.

[58] IEEE. IEEE Std 1666-2005 IEEE Standard SystemC Language Reference. Technical report, IEEE, 2005.

[59] R.P. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules", SIGPLAN Notices 23,8, pp.49-62. August 1988.

[60] A.R. Weiss, "Dhrystone Benchmark: History, Analysis, 'Scores' and Recommendations", ECL LLC., 2002.

[61] J.J. Labrosse, MicroC/OS-II: The Real Time Kernel, CMP Books, 2002.

[62] Wikipedia. CPU cache [En ligne], 2008. http://en.wikipedia.org/wiki/CPU_cache. (page consultée le 1er novembre 2008).

[63] Altera, NIOS-II Processor Reference Handbook, Document Revision 7.2, Altera Inc, Octobre 2007.

[64] Wikipedia. CPU Cache [En ligne], http://en.wikipedia.org/wiki/CPU_cache. (Page consultée le 2 novembre 2008).

[65] L. Moss. *Profilage, caractérisation et partitionnement fonctionnel dans une plate-forme de conception de systèmes embarqués*, Thèse de doctorat, Département de Génie Informatique, École Polytechnique de Montréal, Parution prévue pour décembre 2009.

[66] Wikipedia. Demangling [En ligne], http://en.wikipedia.org/wiki/Demangling. (Page consultée le 17 novembre 2008).

ANNEXE A

DIAGRAMMES DE CLASSES

A.1.Classes du partitionnement de mémoire

La phase de partitionnement de mémoire est composée d'une trentaine de classes et interagit avec une demi-douzaine de classes et d'interfaces supplémentaires. La Figure 5.7 illustre les classes directement impliquées dans le partitionnement de mémoire dans un diagramme de classe simplifié.

La classe **CMemoryEvaluator** est la classe principale du sous-système de partitionnement. Elle permet de gérer les différents processus et les interactions entre ceux-ci. **CMemoryEvaluator** utilise plusieurs classes pour effectuer le travail de lecture du fichier ELF, de lecture des fichiers de trace, de profilage, de génération du script de l'éditeur de liens et d'écriture des fichiers de trace de cache. Par contre, elle effectue elle-même le travail de relocalisation des symboles, d'estimation de performance et génère le contenu des fichiers de trace de cache.

Les classes **CELFInfo** et **CELFInfoLoader** permettent respectivement de stocker et de lire les symboles (**CELFSymbol**) et les sections (**CELFSection**) contenus dans un fichier ELF. La classe **CELFInfoLoader** en particulier lit le fichier ELF, établit les relations entre les symboles et les sections et stocke ces informations dans des structures contenues dans la classe **CELFInfo**. La classe **CELFInfo** permet de stocker ces informations et propose des méthodes facilitant l'accès à ces informations.

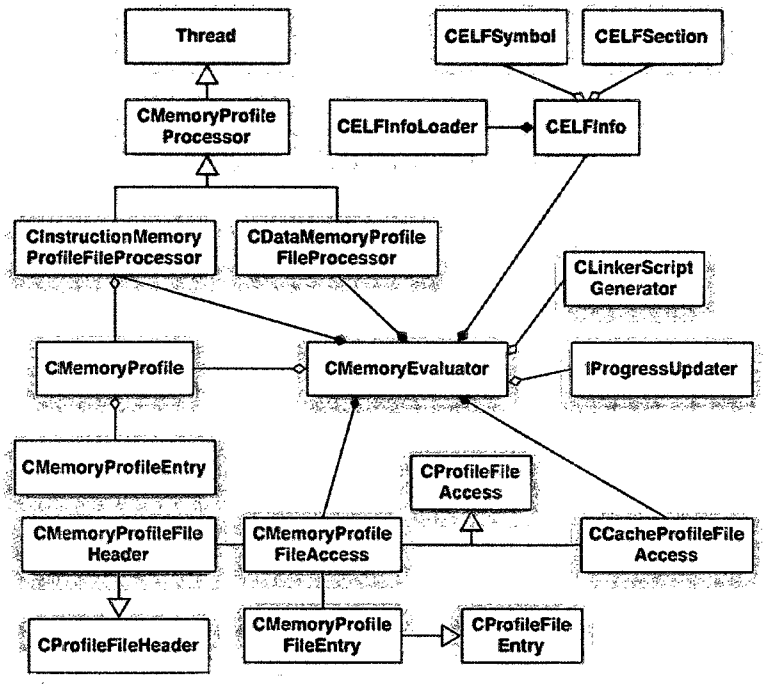

Figure 5.7 Diagramme de classes simplifié du partitionnement de mémoire

La lecture des fichiers de trace s'effectue à l'aide des classes **CMemoryProfileFileAccess** et **CCacheProfileFileAccess** qui dérivent toutes deux de la classe de base **CProfileFileAccess**. Ces classes permettent de lire l'en-tête de fichier (**CMemoryProfileFileHeader** ou **CCacheProfileFileHeader**) et de lire les entrées (compressées ou non) contenues dans le fichier de trace. Chaque entrée de trace lue est chargée sous la forme d'une instance des classes **CMemoryProfileFileEntry** ou **CCacheProfileFileEntry**. Ces classes permettent donc d'abstraire le format des fichiers de traces et de simplifier l'accès aux entrées.

Les profils d'exécution quant à eux sont construits à l'aide des classes **CInstructionProfileProcessor** et **CDataProfileProcessor**. Chacune de ces classes dérive de la classe **CMemoryProfileProcessor** qui dérive à son tour de la classe Thread. Ceci permet à ces classes d'exécuter les opérations de profilage dans leur propre fil d'exécution. Les informations de profilage résultantes sont stockées dans un objet de type **CMemoryProfile**.

La classe **CMemoryProfile** contient l'information de profilage pour chaque symbole rencontré durant la lecture du fichier de trace. Cette information est stockée sous la forme d'une liste d'objets **CMemoryProfileEntry**. Ces objets contiennent un lien vers le symbole (situé dans la classe **CELFInfo**) auquel l'objet réfère ainsi qu'un compte des accès en lecture et en écriture. La liste est triée dans un ordre décroissant, les symboles les plus utilisés sont situés au dessus de la liste et les symboles les moins utilisés sont situés au bas de la liste. Les symboles qui ne sont pas accédés dans la trace ne se retrouvent pas dans cette liste.

La classe **CLinkerScriptGenerator** ainsi que les classes qui lui sont reliées (non représentées sur la Figure 5.7) permettent de générer le fichier script de l'éditeur de liens qui permettra de relocaliser les symboles en mémoire locale. Ce sous-système est décrit en détails à la section 4.2.7. Finalement, l'interface **IProgressUpdater** permet d'informer l'interface graphique (si elle est présente) de la progression de l'analyse.

A.2.Classes de la génération de script de l'éditeur de liens

Le partitionnement de mémoire utilise la classe **CLinkerScriptGenerator** afin de générer le fichier script de l'éditeur de liens correspondant au nouveau partitionnement de mémoire. À l'aide de cette classe, l'algorithme de partitionnement peut spécifier les symboles à relocaliser ainsi que la taille de la pile et du tas.

Le système de génération du script de l'éditeur de liens (Figure 5.8) permet de générer un script spécifique à un processeur en héritant de la classe **CLinkerScriptGenerator** et en implantant un certain nombre de méthodes ainsi qu'en spécialisant la classe **CLinkerScriptStartupElement**. À l'aide de ce système, il est possible d'ajouter un nombre arbitraire de sections et de mémoires au script de l'éditeur de liens ainsi que de spécifier la taille de la pile et du tas. Finalement, il est possible de spécifier l'emplacement en mémoire de l'ensemble des sections définies.

La classe **CLinkerScriptGenerator** contient une liste d'éléments de type **CLinkerScriptElement**. Ces éléments peuvent être une ou des définitions, le point de départ de l'exécutable, le bloc de déclaration des mémoires ou le bloc de déclaration de sections. L'élément **CLinkerScriptMemoriesElement** contient à son tour une liste des mémoires disponibles dans le système. L'élément **CLinkerScriptSectionsElement**, quant à lui, contient les données à contenir dans le bloc de sections sous la forme d'une liste de **CLinkerScriptSectionElement** et contient donc des éléments de définition d'un symbole sous la forme d'un objet de type **CLinkerScriptSectionElementDefine** ainsi que les sections proprement dites sous la forme d'un objet de type **CLinkerScript SectionElementSection.**

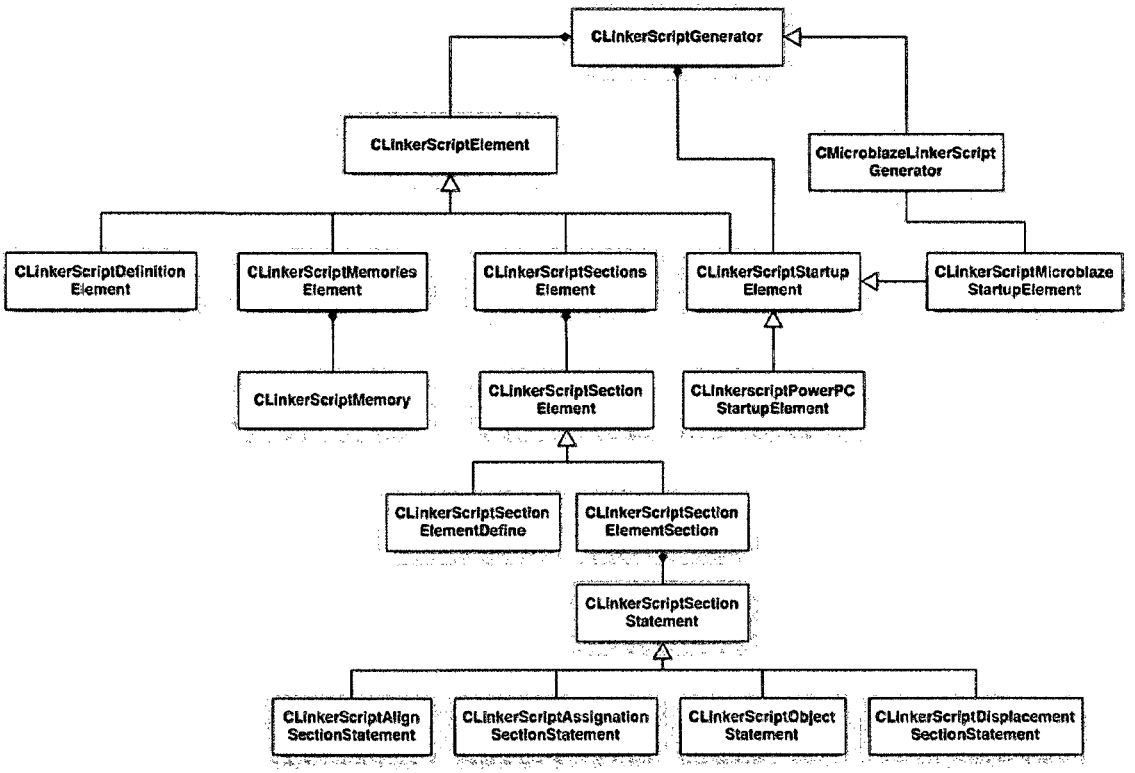

Figure 5.8 Diagramme de classes du générateur de scripts de l'éditeur de liens

L'objet **CLinkerScriptSectionElementSection** peut à son tour contenir un certain nombre d'énoncés. Ces énoncés spécifient un alignement, une assignation de valeur à un symbole, un assignation de déplacement à un symbole ainsi qu'un énoncé spécifiant l'objet à inclure dans cette section. Cet objet constitue généralement une section définie dans un fichier objet.

Finalement, chaque classe impliquée dans la génération du script de l'éditeur de liens implémente la méthode *toString()*. Cette méthode est appelée récursivement par les objets et permet de générer le *script* sous la forme d'une chaîne de caractères qui peut ensuite être écrite dans un fichier.

A.3.Classes de l'optimisation du cache

En accordance avec la simplicité du flux des données, le diagramme de classes (Figure 5.9) correspondant est également simple et ne comprend qu'une dizaine de classes. La classe principale est la classe **CCacheEvaluator** qui effectue la création des tâches de simulation, la lecture des fichiers de trace, l'estimation de la performance et la sélection des meilleures configurations de cache.

Afin de lire la trace, **CCacheEvaluator** utilise la classe **CProfileFileAccess** qui permet la lecture d'un fichier de trace de cache compressé. Chaque entrée lue dans la trace est représentée par un objet de type **CProfileFileEntry**. Il est également possible de lire l'en-tête de fichier qui est alors chargé dans un objet **CCacheProfileFileHeader**.

L'algorithme de simulation des caches, basé sur le principe d'inclusion, est implémenté dans la classe **CCacheAnalysisThread** qui permet donc d'évaluer plusieurs configurations de cache en parallèle. Cette classe lit les accès (**CCacheAccess**) à simuler en provenance de **CCacheEvaluator** (via une FIFO synchronisée) et effectue la simulation de cet accès pour chaque cache.

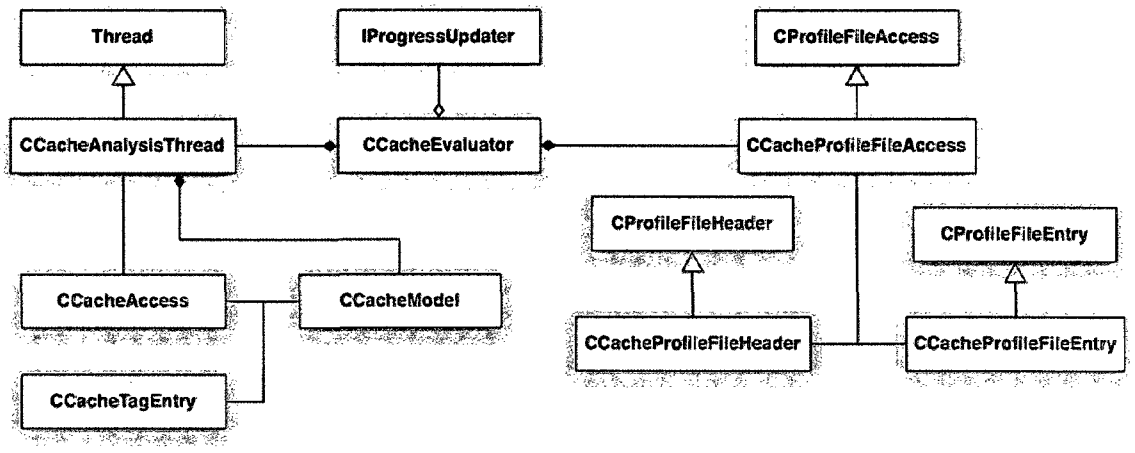

Figure 5.9 Diagramme de classes simplifié de l'optimisation de mémoire cache

Chaque configuration de cache est représentée par une instance de la classe **CCacheModel**. Cette classe modélise une cache de type *accès direct* et permet de spécifier un grand nombre de paramètres : taille du cache, taille d'une ligne de cache, plage d'adresses à cacher ainsi que les différentes latences d'accès. Puisque seuls les adresses accédées sont nécessaires pour la simulation, le tableau des *tags* est suffisant et le contenu du tableau des données en cache n'a pas à être simulé. Ce tableau de *tags* est composé d'objets de type **CCacheTagEntry** qui contiennent l'étiquette ainsi que les drapeaux de statut nécessaires à la simulation.

ANNEXE B

INTERFACE GRAPHIQUE

Afin de simplifier l'utilisation des outils de partitionnement de mémoire et d'optimisation des mémoires cache, une interface graphique a été développée. À l'aide de ce GUI, il est possible de spécifier le fichier ELF, les fichiers de trace à analyser ainsi que les différents paramètres d'analyse. Finalement, l'interface permet de démarrer l'optimisation et d'afficher les résultats sous forme de graphiques, de tableaux et de texte.

B.1. Flot

Le flot d'utilisation de l'interface graphique consiste d'abord à configurer puis à lancer le processus d'optimisation de la hiérarchie mémoire. Lorsque cette optimisation est terminée, les résultats globaux, de même que les résultats spécifiques au partitionnement de mémoire et à l'optimisation du cache, sont affichés. L'utilisateur peut alors sauvegarder le script de l'éditeur de liens généré par le partitionnement de mémoire et prendre en note la configuration recommandée par l'outil. À l'aide de ces informations, le système sur puce peut être reconfiguré par l'utilisateur et simulé à nouveau.

B.1.1. Configuration

Avant de pouvoir démarrer l'optimisation de la hiérarchie mémoire, il est nécessaire de spécifier plusieurs paramètres, comme le fichier ELF utilisé lors de la simulation, les fichiers de trace générés par cette même simulation, la taille maximale de la mémoire locale, etc. Cette étape de configuration est activée en créant une nouvelle session d'optimisation (Figure 5.10). Cette action affiche la fenêtre de configuration (Figure 5.11) et l'utilisateur peut alors choisir s'il désire effectuer soit l'opération de

partitionnement de mémoire, soit l'optimisation de la mémoire cache ou les deux. Si l'utilisateur sélectionne le partitionnement de mémoire, il doit spécifier dans les cases appropriées les fichiers de trace de mémoire.

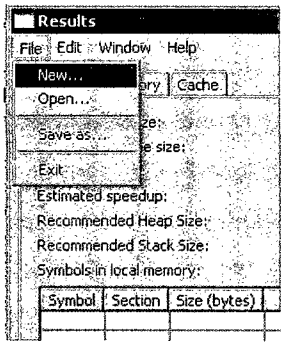

Figure 5.10 Création d'une nouvelle session d'optimisation

Par contre, s'il sélectionne uniquement l'optimisation de la mémoire cache, il doit spécifier les traces de cache dans les cases appropriées. Dans le cas où l'utilisateur désire effectuer les deux opérations, il n'a qu'à spécifier les traces de mémoire puisque les traces de cache seront générées automatiquement par le processus de partitionnement de mémoire.

Lorsque l'utilisateur sélectionne le partitionnement de mémoire, des options supplémentaires sont activées. Le pourcentage de couverture peut être modifié (la valeur par défaut est de 90%) et la taille maximale de la mémoire locale peut être changée.

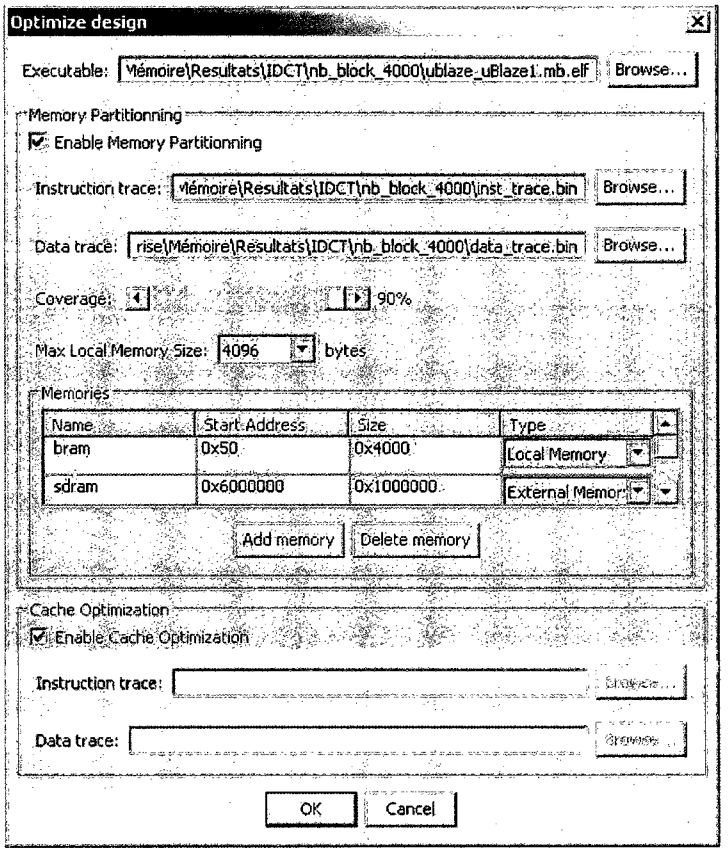

Figure 5.11 Fenêtre de configuration

Finalement, les mémoires disponibles dans le système sont ajoutées par l'utilisateur. Ce dernier doit spécifier le nom de la mémoire qui apparaîtra dans le script de l'éditeur de liens, l'adresse de départ de la mémoire et la taille de celle-ci. La dernière option est le type de mémoire : locale ou externe. Les symboles devant être placés en mémoire locale seront placés dans la mémoire de type locale et de même pour les symboles devant être placés en mémoire externe.

Lorsque la configuration est terminée, l'utilisateur appuie sur le bouton *OK* et l'optimisation peut démarrer si les options choisies sont toutes valides. L'optimisation débutera et une fenêtre indiquant la progression sera affichée (Figure 5.12).

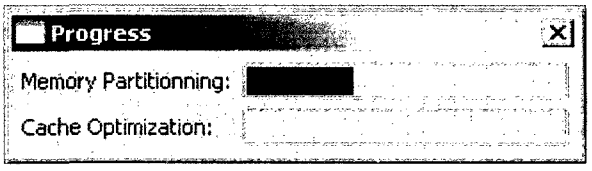

Figure 5.12 Fenêtre indiquant la progression de l'optimisation

B.1.2. Résultats

Lorsque l'étape d'optimisation est terminée, les fenêtres de configuration et de progression font place à la fenêtre des résultats (Figure 5.13). Cette fenêtre présente les résultats du partitionnement effectué et de l'optimisation des mémoires cache. Dans le premier onglet nommé *Overview*, on retrouve l'essentiel de la nouvelle configuration mémoire. La taille de la mémoire locale est affichée de même que la taille des deux mémoires cache. Ensuite, l'accélération estimée tenant compte du partitionnement et des caches est affichée. De plus, les tailles du tas et de la pile sont présentées. Ensuite, on retrouve le tableau des symboles relocalisés en mémoire locale et finalement un graphique illustrant le nombre estimé de cycles.

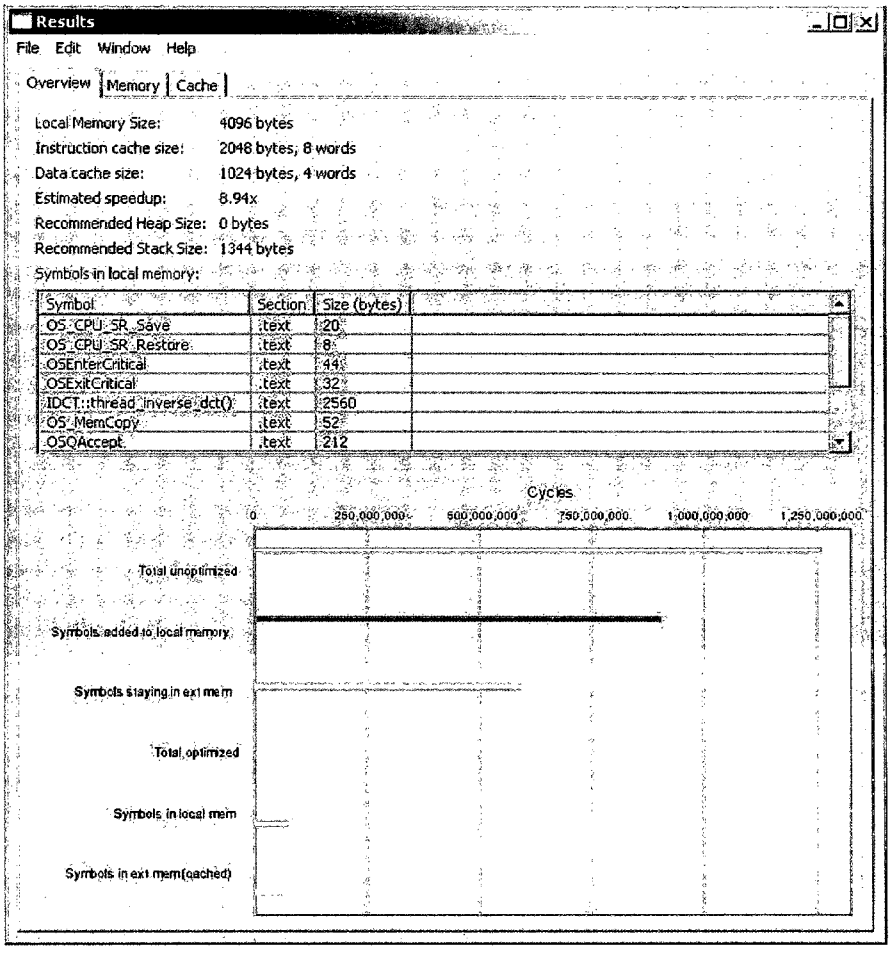

Figure 5.13 Fenêtre des résultats – Survol

Le deuxième onglet, nommé *Memory* (Figure 5.14), présente les résultats spécifiques au partitionnement de mémoire. Dans le premier sous-onglet, *Results*, on retrouve sensiblement les mêmes informations présentées à l'onglet de survol des résultats. Le tableau des symboles relocalisés est d'abord présenté. Ce tableau affiche chaque symbole qui doit être relocalisé, la section dans laquelle ce symbole se retrouve ainsi que la taille du symbole. On retrouve ensuite la taille suggérée du tas et de la pile ainsi que la taille de la mémoire locale. L'accélération estimée et le graphique du nombre estimé de cycles sont également affichés. Par contre, l'accélération et le graphique ne tiennent compte que du partitionnement de mémoire, contrairement à ce qui est présenté dans l'onglet *Overview*.

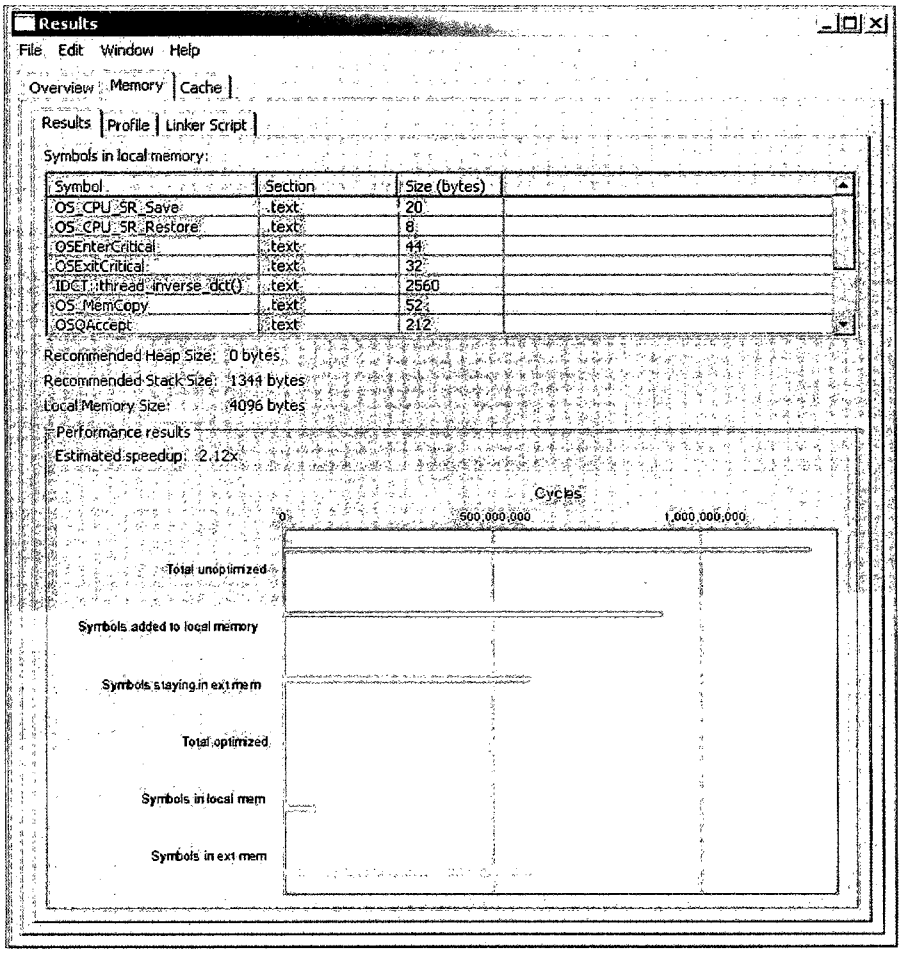

Figure 5.14 Sous-onglet des résultats de partitionnement

Le deuxième sous-onglet présente les résultats du profilage sous la forme d'un graphique à barres. Le nom de chaque symbole ainsi que son temps relatif est affiché et trié en ordre décroissant. L'utilisateur peut ainsi comparer ce profilage avec les symboles relocalisés en mémoire locale pour confirmer les décisions de l'algorithme de partitionnement. Si l'utilisateur décide que les symboles relocalisés ne sont pas à sa satisfaction, il est libre de modifier manuellement le script de l'éditeur de liens présenté dans le troisième sous-onglet *Linker Script*. Il peut ensuite sauvegarder sur le disque ce fichier script afin de l'utiliser lors de la prochaine compilation de son logiciel.

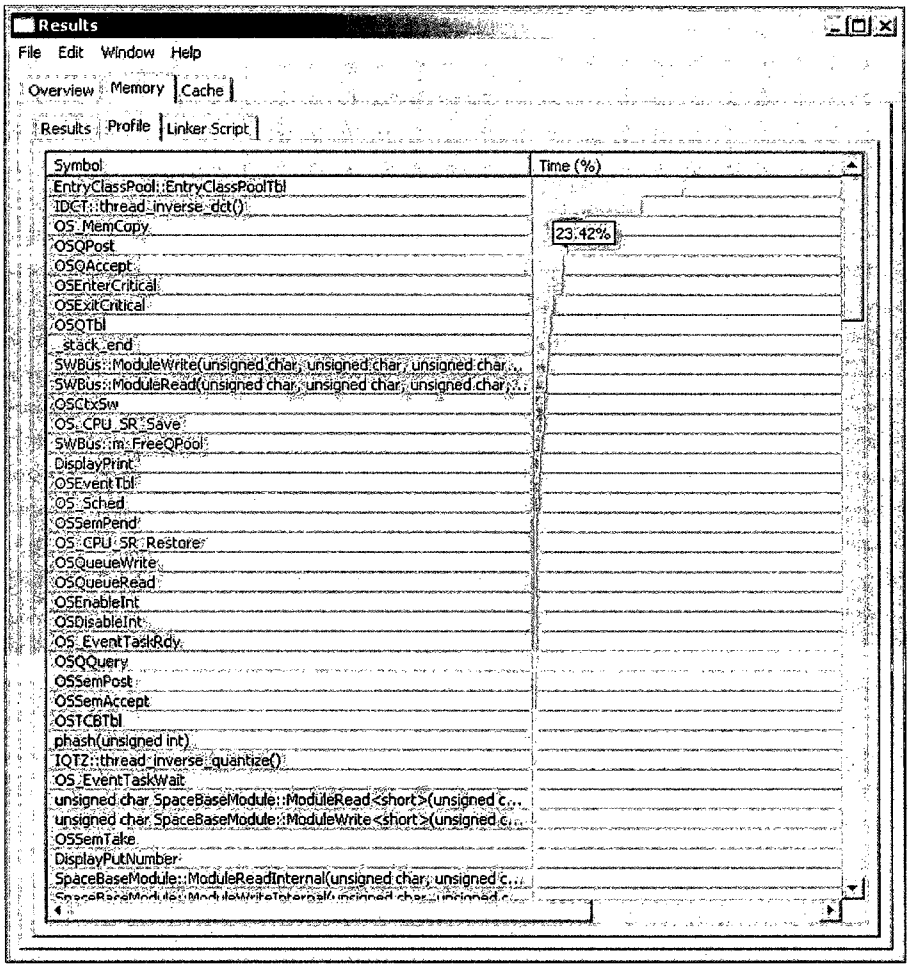

Figure 5.15 Sous-onglet de profilage des symboles

Le dernier onglet présente les résultats de l'optimisation des mémoires cache. Le sous-onglet *Results* (Figure 5.16) présente les résultats sous la forme de deux tableaux. Le premier tableau présente les configurations des caches d'instructions tandis que le deuxième tableau présente celles des caches de données. Chaque entrée dans ces tableaux contient la configuration du cache selon le format « taille de la cache en octets/ taille des lignes de cache en mots». Le résultat de la configuration est ensuite donné suivi de l'accélération estimée pour cette configuration. Finalement, la latence totale d'accès au cache est présentée et le pourcentage de succès et d'échecs sont présentés. La configuration présentant le meilleur résultat du tableau est surligné en vert et indique la configuration choisie par l'algorithme.

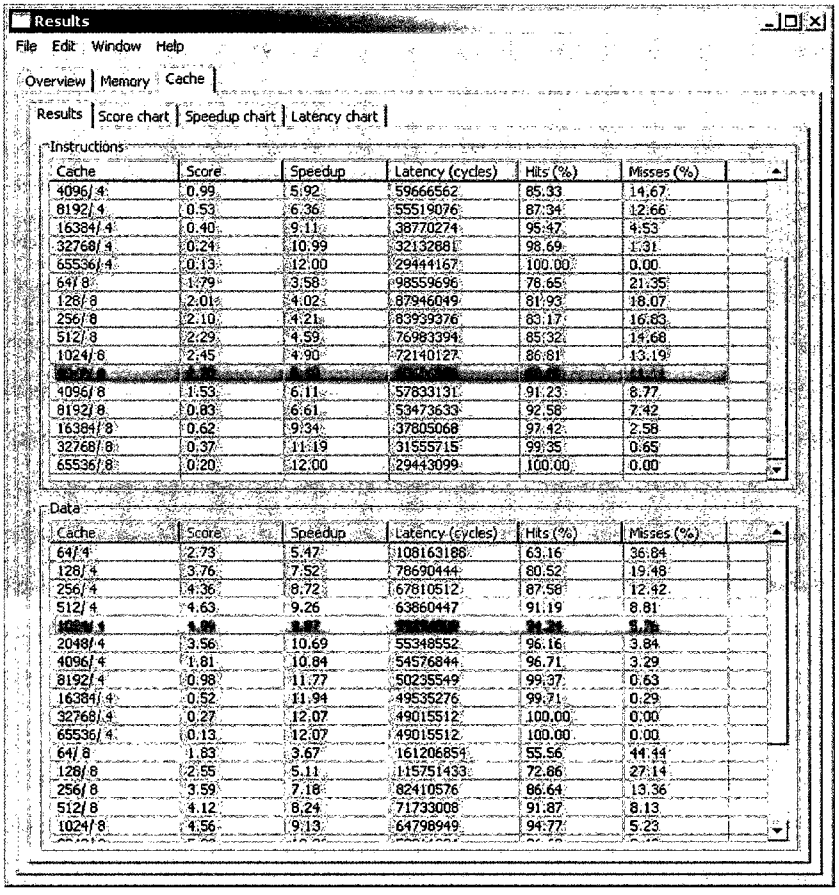

Figure 5.16 Onglet de résultats de l'optimisation des mémoires cache

Les trois derniers sous-onglets de l'onglet *Cache* présentent le résultat (Figure 5.17), l'accélération et la latence estimée sous la forme de graphiques. Sur chaque graphique on retrouve la liste des configurations de cache (instructions ou données selon la sélection au bas de la fenêtre) à l'ordonnée et la métrique à l'abscisse. Le sous-onglet *Score chart*, *Speedup chart* et *Latency chart* présentent le résultat, l'accélération et la latence estimée, respectivement.

À l'aide de ces onglets il est donc possible de visualiser graphiquement les métriques de performance de chaque configuration de cache. Selon les résultats présentés, si l'utilisateur n'est pas satisfait de la configuration choisie par l'algorithme, il possède

alors toutes les données de performance nécessaires afin d'effectuer un choix éclairé. Il peut donc décider que les caches offrant les meilleures performances, peu importe leur taille, sont la configuration la plus désirable.

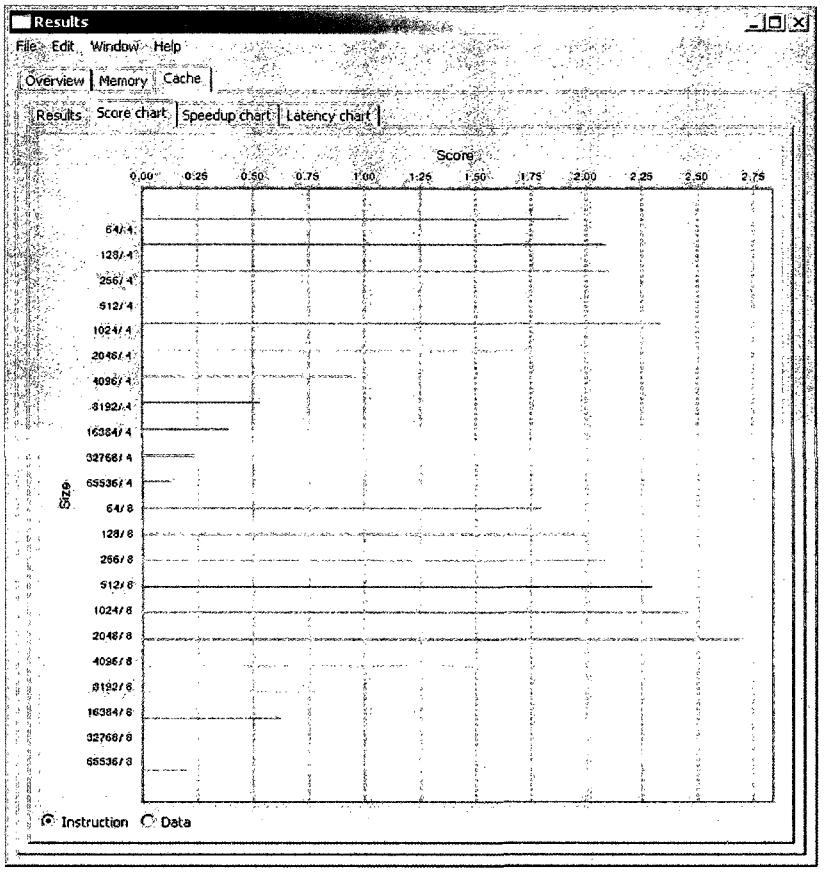

Figure 5.17 Sous-onglet des résultats des configurations de cache

B.1.3. Classes

L'interface graphique est composée d'un bon nombre de classes. La classe principale est **CMemoryOptimizerMain** qui gère les fenêtres de configuration, de résultats et de progression en plus de démarrer les processus d'optimisation et de recueillir les résultats de ces derniers afin de les afficher dans la fenêtre de résultats.

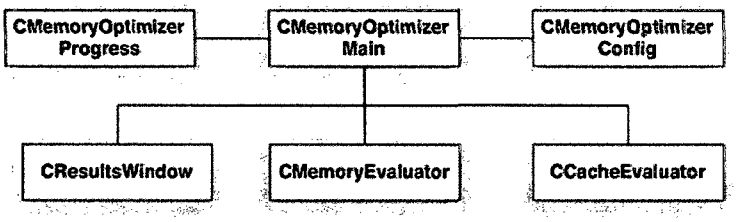

Figure 5.18 Diagramme de classe des classes principales du GUI

La fenêtre de résultats (**CResultsWindow**) est composée de trois onglets de résultats: survol, partitionnement de mémoire et optimisation des caches. Chacun de ces onglets hérite de la classe **CResultTab** qui implémente les fonctionnalités communes pour tous les onglets et sous-onglets de l'Interface graphique. Les onglets sont contenus dans un objet **TabFolder** (classe faisant parti de SWT) dans **CResultsWindow**. Finalement, afin de permettre l'accès en écriture aux données à afficher, la classe expose un certain nombre de méthodes.

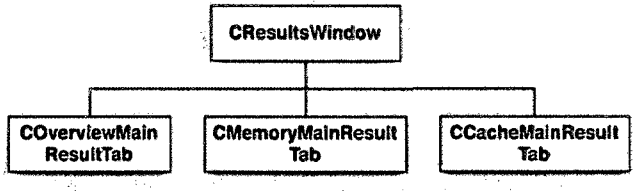

Figure 5.19 Diagramme de classe de la fenêtre de résultats

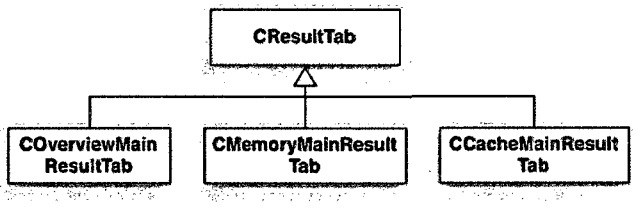

Figure 5.20 Diagramme de classe des onglets

Chaque objet héritant de **CResultTab** contient un objet **TabFolder** qui lui permet de contenir ses propres sous-onglets. Dans le cas de **COverviewMainResultTab**, cet objet ne contient aucun sous-onglet puisque toute l'information peut être présentée sur une seule page.

CMemoryMainResultTab, qui présente les résultats du partitionnement de mémoire, contient trois sous-onglets. Le premier, **CResultsMemoryResultTab** présente les recommandations de l'algorithme de partitionnement de mémoire. Le deuxième sous-onglet, **CProfileMemoryResultTab**, se charge de présenter les résultats de profilage tandis que le dernier sous-onglet, **CLinkerScriptMemoryResultTab**, affiche le script de l'éditeur de liens et permet de le sauvegarder.

Afin de supporter la sauvegarde de ce fichier, la classe **CReplaceFileDialog** est utilisée lorsqu'un fichier possède le même nom que celui spécifié par l'utilisateur comme nom du *script*. **CReplaceFileDialog** permet donc d'afficher une fenêtre de dialogue qui demande à l'utilisateur s'il désire ou non écraser le fichier existant et retourne la décision à **CLinkerScriptMemoryResultTab**.

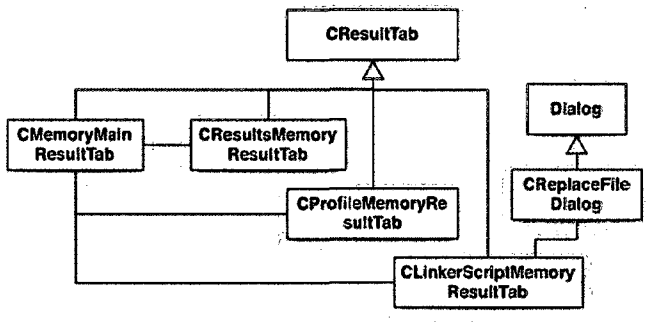

Figure 5.21 Diagramme de classe des sous-onglets de partitionnement de mémoire

Finalement, l'onglet présentant les résultats de l'optimisation de cache, **CCacheMainResultTab**, contient quatre sous-onglets. Le premier sous-onglet, **CResultsCacheResultTab** présente les résultats combinés pour toutes les configurations de cache. Les trois autres sous-onglets, **CScoreChartCacheResultTab**, **CSpeedupChartCacheResultTab** et **CLatencyChartCacheResultTab**, présentent respectivement les graphiques des résultats, accélérations et latences.

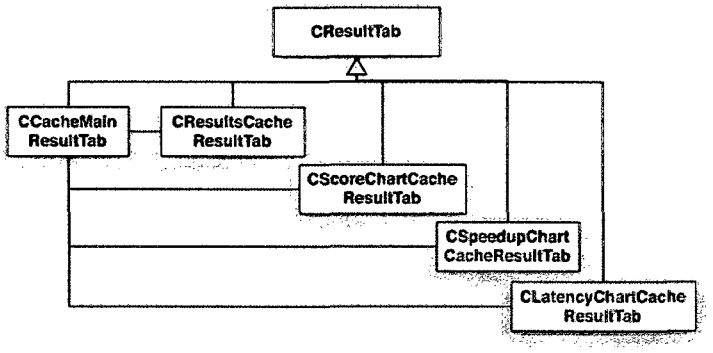

Figure 5.22 Diagramme de classe des sous-onglets d'optimisation de cache

ANNEXE C

RÉSULTATS DE PROFILAGE

C.1.Dhrystone

Tableau 5.5 Résultats de profilage pour Dhrystone

Symbol	Section	Taille (octets)	Temps (%)
dhrystone()	.text	3328	34.32
strcmp	.text	132	15.51
_stack_end	.stack	640	13.25
_heap_start	.heap	192	9.89
Func_2(char*, char*)	.text	184	5.78
Proc_8(int*, int (*) [50], int, int)	.text	124	5.78
Proc_7(int, int, int*)	.text	20	2.48
Func_1(char, char)	.text	44	2.31
Proc_6(Enumeration, Enumeration*)	.text	164	2.31
__rodata_start	.rodata	1522	2.30
Arr_2_Glob	.bss	10152	1.22
Ptr_Glob	.bss	4	1.02
Ch_2_Glob	.bss	4	1.01
Arr_1_Glob	.bss	204	0.98
Int_Glob	.bss	4	0.75
Ch_1_Glob	.bss	4	0.50
Bool_Glob	.bss	4	0.48
DisplayPrint	.text	220	0.04
L_start	.text	20	0.03
L_start	.text	20	0.03
_crtinit	.text	152	0.02
DisplayFormatItem	.text	708	0.01
cISSAdapterID	.data	4	0.00
DisplayPutNumber	.text	428	0.00
_malloc_r	.text	2680	0.00
__pack_d	.text	748	0.00
__unpack_d	.text	348	0.00
SpacePrint	.data	4	0.00
__muldf3	.text	1232	0.00
__divdf3	.text	628	0.00
sbrk	.text	116	0.00
_sbrk_r	.text	84	0.00

__fixsfsi	.text	200	0.00
__pack_f	.text	368	0.00
__unpack_f	.text	192	0.00
main	.text	112	0.00
__malloc_av_	.data	1048	0.00
__malloc_current_mallinfo	.bss	40	0.00
__fixdfsi	.text	212	0.00
__extendsfdf2	.text	84	0.00
malloc	.text	40	0.00
__do_global_ctors_aux	.text	72	0.00
__floatsisf	.text	184	0.00
Next_Ptr_Glob	.bss	4	0.00
__make_dp	.text	48	0.00
__init	.init	44	0.00
heap_ptr	.bss	4	0.00
_start1	.text	36	0.00
frame_dummy	.text	60	0.00
MicroBlaze_disable_dcache	.text	32	0.00
MicroBlaze_disable_icache	.text	32	0.00
MicroBlaze_enable_dcache	.text	32	0.00
MicroBlaze_enable_icache	.text	32	0.00
__malloc_max_sbrked_mem	.bss	4	0.00
__malloc_max_total_mem	.bss	4	0.00
__malloc_sbrk_base	.data	4	0.00
errno	.bss	4	0.00
__malloc_top_pad	.bss	4	0.00
Dhrystones_Per_Second	.bss	4	0.00
Microseconds	.bss	4	0.00
User_Time	.bss	4	0.00
__malloc_lock	.text	8	0.00
__malloc_unlock	.text	8	0.00
MicroBlaze_init_dcache_range	.text	16	0.00
MicroBlaze_init_icache_range	.text	16	0.00
_impure_ptr	.data	4	0.00
Begin_Time	.bss	4	0.00
End_Time	.bss	4	0.00

Tableau 5.6 Symboles en mémoire locale pour Dhrystone – 2K-1K/4-128/4

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
strcmp	.text	132	182	15.51	15.51
_stack_end	.stack	640	822	13.25	28.76
_heap_start	.heap	192	1014	9.89	38.64
Func_2(char*, char*)	.text	184	1322	5.78	50.19
Proc_8(int*, int (*) [50], int, int)	.text	124	1138	5.78	44.42
Proc_7(int, int, int*)	.text	20	1342	2.48	52.67
Func_1(char, char)	.text	44	1550	2.31	57.29
Proc_6(Enumeration, Enumeration*)	.text	164	1506	2.31	54.98
Ptr_Glob	.bss	4	1554	1.02	58.31
Ch_2_Glob	.bss	4	1558	1.01	59.31
Arr_1_Glob	.bss	204	1762	0.98	59.31
Int_Glob	.bss	4	1766	0.75	59.31
Ch_1_Glob	.bss	4	1770	0.50	59.31
Bool_Glob	.bss	4	1774	0.48	59.31
DisplayPrint	.text	220	1994	0.04	59.31
L_start	.text	20	2014	0.03	59.31
L_start	.text	20	2034	0.03	59.31
cISSAdapterID	.data	4	2038	0.00	59.31
SpacePrint	.data	4	2042	0.00	59.31
Next_Ptr_Glob	.bss	4	2046	0.00	59.31

Tableau 5.7 Symboles en mémoire locale pour Dhrystone – 4K-1K/4-64/4

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
strcmp	.text	132	182	15.51	15.51
_stack_end	.stack	640	822	13.25	28.76
_heap_start	.heap	192	1014	9.89	38.64
Proc_8(int*, int (*) [50], int, int)	.text	124	1138	5.78	44.42
Func_2(char*, char*)	.text	184	1322	5.78	50.19
Proc_7(int, int, int*)	.text	20	1342	2.48	52.67
Proc_6(Enumeration, Enumeration*)	.text	164	1506	2.31	54.98
Func_1(char, char)	.text	44	1550	2.31	57.29
__rodata_start	.rodata	1522	3072	2.30	59.59
Ptr_Glob	.bss	4	3076	1.02	60.61
Ch_2_Glob	.bss	4	3080	1.01	61.62
Arr_1_Glob	.bss	204	3284	0.98	61.62
Int_Glob	.bss	4	3288	0.75	61.62
Ch_1_Glob	.bss	4	3292	0.50	61.62
Bool_Glob	.bss	4	3296	0.48	61.62
DisplayPrint	.text	220	3516	0.04	61.62
L_start	.text	20	3536	0.03	61.62
L_start	.text	20	3556	0.03	61.62
_crtinit	.text	152	3708	0.02	61.62
cISSAdapterID	.data	4	3712	0.00	61.62

__unpack_d	.text	348	4060	0.00	61.62
SpacePrint	.data	4	4064	0.00	61.62
Next_Ptr_Glob	.bss	4	4068	0.00	61.62
heap_ptr	.bss	4	4072	0.00	61.62
__malloc_max_total_mem	.bss	4	4076	0.00	61.62
__malloc_max_sbrked_mem	.bss	4	4080	0.00	61.62
__malloc_sbrk_base	.data	4	4084	0.00	61.62
errno	.bss	4	4088	0.00	61.62
__malloc_top_pad	.bss	4	4092	0.00	61.62

Tableau 5.8 Symboles en mémoire locale pour Dhrystone – 8K-1K/4-64/4

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
dhrystone()	.text	3328	3378	34.32	34.32
strcmp	.text	132	3510	15.51	49.83
_stack_end	.stack	640	4150	13.25	63.08
_heap_start	.heap	192	4342	9.89	72.97
Proc_8(int*, int (*) [50], int, int)	.text	124	4466	5.78	78.74
Func_2(char*, char*)	.text	184	4650	5.78	84.52
Proc_7(int, int, int*)	.text	20	4670	2.48	86.99
Proc_6(Enumeration, Enumeration*)	.text	164	4834	2.31	89.30
Func_1(char, char)	.text	44	4878	2.31	91.61
__rodata_start	.rodata	1522	6400	2.30	93.91
Ptr_Glob	.bss	4	6404	1.02	94.93
Ch_2_Glob	.bss	4	6408	1.01	95.94
Arr_1_Glob	.bss	204	6612	0.98	95.94
Int_Glob	.bss	4	6616	0.75	95.94
Ch_1_Glob	.bss	4	6620	0.50	95.94
Bool_Glob	.bss	4	6624	0.48	95.94
DisplayPrint	.text	220	6844	0.04	95.94
L_start	.text	20	6864	0.03	95.94
L_start	.text	20	6884	0.03	95.94
_crtinit	.text	152	7036	0.02	95.94
DisplayFormatItem	.text	708	7744	0.01	95.94
clSSAdapterID	.data	4	7748	0.00	95.94
DisplayPutNumber	.text	428	8176	0.00	95.94
SpacePrint	.data	4	8180	0.00	95.94
Next_Ptr_Glob	.bss	4	8184	0.00	95.94
heap_ptr	.bss	4	8188	0.00	95.94

C.2.IDCT

Tableau 5.9 Résultats de profilage pour IDCT

Symbole	Section	Taille (octets)	Temps (%)
EntryClassPool::EntryClassPoolTbl	.bss	8356	23.37
IDCT::thread_inverse_dct()	.text	2600	16.73
OS_MemCopy	.text	52	8.19
OSQPost	.text	280	5.50
OSQAccept	.text	216	4.85
OSEnterCritical	.text	44	4.58
OSExitCritical	.text	32	3.33
OSQTbl	.bss	568	2.91
_stack_end	.stack	1344	2.74
SWBus::ModuleWrite(unsigned char, unsigned char, unsigned char, unsigned long, void*, unsigned long)	.text	1808	2.48
SWBus::ModuleRead(unsigned char, unsigned char, unsigned char, unsigned long, void*, unsigned long)	.text	1144	2.31
OSCtxSw	.text	348	2.13
OS_CPU_SR_Save	.text	20	2.08
SWBus::m_FreeQPool	.bss	2076	1.60
DisplayPrint	.text	220	1.32
OSEventTbl	.bss	2640	1.24
OS_Sched	.text	192	1.18
OSSemPend	.text	308	0.84
OS_CPU_SR_Restore	.text	8	0.83
OSQueueRead	.text	28	0.75
OSQueueWrite	.text	28	0.75
OSDisableInt	.text	28	0.74
OSEnableInt	.text	28	0.74
OS_EventTaskRdy	.text	236	0.73
OSQQuery	.text	236	0.69
OSSemPost	.text	212	0.53
OSSemAccept	.text	136	0.48
OSTCBTbl	.bss	6756	0.44
phash(unsigned int)	.text	44	0.41
IQTZ::thread_inverse_quantize()	.text	360	0.38
OS_EventTaskWait	.text	144	0.35
unsigned char SpaceBaseModule::ModuleRead<short>(unsigned char, unsigned char, unsigned long, short*, unsigned char)	.text	132	0.32
unsigned char SpaceBaseModule::ModuleWrite<short>(unsigned char, unsigned char, unsigned long, short*, unsigned char)	.text	132	0.32
OSSemTake	.text	148	0.32
SpaceBaseModule::ModuleReadInternal(unsigned char, unsigned char, unsigned long, void*, unsigned char)	.text	236	0.25
SpaceBaseModule::ModuleWriteInternal(unsigned char, unsigned char, unsigned long, void*, unsigned char)	.text	300	0.25
DisplayPutNumber	.text	428	0.22
__rodata_start	.rodata	3766	0.21
DirectLinkUtility::isDirectLinkTarget(unsigned char)	.text	32	0.20

OSSemBGive	.text	52	0.20
sc_module::GetVerbose()	.text	44	0.19
SWBus::m_FreeSemPool	.bss	64	0.16
OSTCBCur	.bss	4	0.16
SWBus::pEntryClassTbl	.bss	1036	0.16
OSInit	.text	568	0.15
clSSAdapterID	.data	4	0.14
DisplayFormatItem	.text	708	0.12
OSQueueGetBytes	.text	36	0.11
OSUnMapTbl	.rodata	260	0.11
OSRdyTbl	.bss	8	0.10
Q::GetFreeBytes()	.text	32	0.10
unsigned char SpaceBaseModule::ModuleRead<unsigned long>(unsigned char, unsigned char, unsigned long, unsigned long*, unsigned char)	.text	132	0.08
unsigned char SpaceBaseModule::ModuleWrite<unsigned long>(unsigned char, unsigned char, unsigned long, unsigned long*, unsigned char)	.text	132	0.08
OSRdyGrp	.bss	4	0.07
OSTCBHighRdy	.bss	4	0.07
OSPrioCur	.bss	4	0.07
OSPrioHighRdy	.bss	4	0.07
OSCtxSwCtr	.bss	4	0.07
OSIntNesting	.bss	4	0.06
OSTCBPrioTbl	.bss	260	0.05
g_pRcvQueueHashTable	.bss	4	0.05
L_start	.text	20	0.05
L_start	.text	20	0.05
OSTaskSwHook	.text	8	0.05
pDirectLinkCoprocTable	.bss	1040	0.04
OSMapTbl	.rodata	8	0.04
OSLockNesting	.bss	4	0.04
SpacePrint	.data	4	0.02
OS_MemClr	.text	44	0.01
OSTaskCreateExt	.text	472	0.01
__divdf3	.text	628	0.00
SWBus::Initialize()	.text	372	0.00
IQTZ::IQTZ(sc_module_name, double, sc_time_unit, unsigned char, unsigned char, bool)	.text	276	0.00
__muldf3	.text	1232	0.00
__unpack_d	.text	348	0.00
__pack_d	.text	748	0.00
strncpy	.text	336	0.00
OSTaskIdleStk	.bss	2080	0.00
memset	.text	256	0.00
__muldi3	.text	156	0.00
_fpadd_parts	.text	868	0.00
OSQCreate	.text	348	0.00
OS_TCBInit	.text	456	0.00
OSFlagTbl	.bss	224	0.00
__fpcmp_parts_d	.text	436	0.00
OSTaskStkInit	.text	396	0.00

OSSemCreate	.text	176	0.00
OS_QInit	.text	108	0.00
__gedf2	.text	172	0.00
__fixdfsi	.text	212	0.00
sc_module::SpaceThread(void (sc_module::*)())	.text	620	0.00
OSTaskCreate	.text	372	0.00
__fixunsdfdi	.text	280	0.00
sc_time::sc_time(double, sc_time_unit)	.text	276	0.00
OS_EventWaitListInit	.text	44	0.00
__fixunsdfsi	.text	140	0.00
zigzag_table	.rodata	64	0.00
SemPool::Initialize()	.text	132	0.00
__nedf2	.text	172	0.00
FreeQPool::Initialize()	.text	124	0.00
__subdf3	.text	148	0.00
SpaceBaseModule::SpaceBaseModule(sc_module_name, double, sc_time_unit, unsigned char, unsigned char, bool)	.text	140	0.00
__adddf3	.text	136	0.00
OS_FlagInit	.text	132	0.00
OSQueueCreate	.text	52	0.00
sc_main(int, char**)	.text	304	0.00
HALInitialize()	.text	204	0.00
OSSemBCreate	.text	60	0.00
IDCT::IDCT(sc_module_name, double, sc_time_unit, unsigned char, unsigned char, bool)	.text	188	0.00
sc_module::sc_module(sc_module_name const&, unsigned char, unsigned char, bool)	.text	96	0.00
OSStartHighRdy	.text	172	0.00
OSTaskSpawn	.text	80	0.00
__floatsidf	.text	256	0.00
__lshrdi3	.text	80	0.00
HALRegisterModule(unsigned char, unsigned char)	.text	72	0.00
OSStart	.text	144	0.00
OSEventFreeList	.bss	4	0.00
EntryClass::Initialize(sc_module*, void (sc_module::*)(), unsigned char, Sem*)	.text	56	0.00
sc_module_name::sc_module_name(sc_module_name const&)	.text	28	0.00
ConvertDeviceIDToAddress(unsigned char)	.text	12	0.00
main	.text	136	0.00
EntryFunc(void*)	.text	84	0.00
HALMainHook()	.text	76	0.00
SemPool::GetSem()	.text	32	0.00
OSQFreeList	.bss	4	0.00
DirectLinkUtility::Initialize()	.text	56	0.00
sc_module_name::sc_module_name(char const*)	.text	28	0.00
sc_module::iSystemThreadNum	.data	4	0.00
FreeQPool::GetQ()	.text	48	0.00
_start1	.text	48	0.00
TorInitialize(int, char**)	.text	44	0.00
EntryClassPool::GetEntryClass(int)	.text	20	0.00
HALStop(unsigned long)	.text	52	0.00
OSTaskCtr	.bss	4	0.00

OSTCBFreeList	.bss	4	0.00
OSTCBList	.bss	4	0.00
OSRunning	.bss	4	0.00
DirectLinkInitializationHook()	.text	32	0.00
MicroBlaze_enable_dcache	.text	32	0.00
MicroBlaze_enable_icache	.text	32	0.00
OS_MBOXInit	.text	32	0.00
OSCtxSw_SavedByISR	.text	28	0.00
OS_CPU_Get_MSR	.text	8	0.00
OSTaskCreateHook	.text	8	0.00
OSTCBInitHook	.text	8	0.00
sc_start()	.text	36	0.00
sc_time_params::time_resolution	.data	8	0.00
time_values	.data	48	0.00
bAckPICBeforeService	.bss	4	0.00
ucOSIsInitialized	.data	4	0.00
cTimerID	.data	4	0.00
XTmrCtr_Offsets	.data	4	0.00
DirectLinkUtility::GetNumberOfDirectLinks()	.text	16	0.00
MicroBlaze_init_dcache_range	.text	16	0.00
MicroBlaze_init_icache_range	.text	16	0.00
OSStartScheduler	.text	28	0.00
sc_module::Initialize()	.text	16	0.00
sc_stop()	.text	28	0.00
cPICID	.data	4	0.00
uiNumberOfDirectLinks	.data	4	0.00
sc_time_params::time_resolution_fixed	.data	4	0.00
L_done	.text	8	0.00
L_done	.text	8	0.00
OSDebugInit	.text	8	0.00
OSInitHookBegin	.text	8	0.00
OSInitHookEnd	.text	8	0.00
SpaceBaseModule::SynchronizeWithOtherPartition()	.text	8	0.00
g_uiRcvQueueHashTableLength	.data	4	0.00
OSFlagFreeList	.bss	4	0.00
OSIdleCtr	.bss	4	0.00
OSMBOXFreeList	.bss	4	0.00
OSMBOXTbl	.bss	12	0.00
OSTime	.bss	4	0.00

Tableau 5.10 Symboles en mémoire locale pour IDCT – 2K-2K/8-1K/8

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
OS_MemCopy	.text	52	102	8.19	8.19
OSQPost	.text	280	382	5.5	13.69
OSQAccept	.text	216	598	4.85	18.54
OSEnterCritical	.text	44	642	4.58	23.12
OSExitCritical	.text	32	674	3.33	26.45
OSQTbl	.bss	568	1242	2.91	29.35
OSCtxSw	.text	348	1590	2.13	31.48

OS_CPU_SR_Save	.text	20	1610	2.08	33.56
DisplayPrint	.text	220	1830	1.32	34.89
OS_Sched	.text	192	2022	1.18	36.06
OS_CPU_SR_Restore	.text	8	2030	0.83	36.06
OSTCBCur	.bss	4	2034	0.16	36.06
cISSAdapterID	.data	4	2038	0.14	36.06
OSRdyTbl	.bss	8	2046	0.1	36.06

Tableau 5.11 Symboles en mémoire locale pour IDCT – 4K-2K/8-1K/8

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
IDCT::thread_inverse_dct()	.text	2600	2650	16.73	16.73
OS_MemCopy	.text	52	2702	8.19	24.92
OSQPost	.text	280	2982	5.5	30.42
OSQAccept	.text	216	3198	4.85	35.27
OSEnterCritical	.text	44	3242	4.58	39.85
OSExitCritical	.text	32	3274	3.33	43.18
OSQTbl	.bss	568	3842	2.91	46.08
OS_CPU_SR_Save	.text	20	3862	2.08	48.16
DisplayPrint	.text	220	4082	1.32	49.49
OS_CPU_SR_Restore	.text	8	4090	0.83	49.49
OSTCBCur	.bss	4	4094	0.16	49.49

Tableau 5.12 Symboles en mémoire locale pour IDCT – 8K-2K/8-2K/8

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
IDCT::thread_inverse_dct()	.text	2600	2650	16.73	16.73
OS_MemCopy	.text	52	2702	8.19	24.92
OSQPost	.text	280	2982	5.50	30.42
OSQAccept	.text	216	3198	4.85	35.27
OSEnterCritical	.text	44	3242	4.58	39.85
OSExitCritical	.text	32	3274	3.33	43.18
OSQTbl	.bss	568	3842	2.91	46.08
_stack_end	.stack	1344	5186	2.74	48.82
SWBus::ModuleWrite(unsigned char, unsigned char, unsigned char, unsigned long, void*, unsigned long)	.text	1808	6994	2.48	51.30
SWBus::ModuleRead(unsigned char, unsigned char, unsigned char, unsigned long, void*, unsigned long)	.text	1144	8138	2.31	53.61
OS_CPU_SR_Save	.text	20	8158	2.08	55.69
OS_CPU_SR_Restore	.text	8	8166	0.83	55.69
OSTCBCur	.bss	4	8170	0.16	55.69
cISSAdapterID	.data	4	8174	0.14	55.69
OSRdyTbl	.bss	8	8182	0.10	55.69
OSRdyGrp	.bss	4	8186	0.07	55.69
OSTCBHighRdy	.bss	4	8190	0.07	55.69

Tableau 5.13 Symboles en mémoire locale pour IDCT – 16K-2K/8-2K/8

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
EntryClassPool::EntryClassPoolTbl	.bss	8356	8406	23.37	23.37
IDCT::thread_inverse_dct()	.text	2600	11006	16.73	40.10
OS_MemCopy	.text	52	11058	8.19	48.29
OSQPost	.text	280	11338	5.50	53.79
OSQAccept	.text	216	11554	4.85	58.64
OSEnterCritical	.text	44	11598	4.58	63.22
OSExitCritical	.text	32	11630	3.33	66.55
OSQTbl	.bss	568	12198	2.91	69.45
_stack_end	.stack	1344	13542	2.74	72.19
SWBus::ModuleWrite(unsigned char, unsigned char, unsigned char, unsigned long, void*, unsigned long)	.text	1808	15350	2.48	74.67
OSCtxSw	.text	348	15698	2.13	76.80
OS_CPU_SR_Save	.text	20	15718	2.08	78.88
DisplayPrint	.text	220	15938	1.32	80.20
OS_Sched	.text	192	16130	1.18	81.38
OS_CPU_SR_Restore	.text	8	16138	0.83	81.38
OSQueueRead	.text	28	16166	0.75	81.38
OSQueueWrite	.text	28	16194	0.75	81.38
OSDisableInt	.text	28	16222	0.74	81.38
OSEnableInt	.text	28	16250	0.74	81.38
phash(unsigned int)	.text	44	16294	0.41	81.38
DirectLinkUtility::isDirectLinkTarget(unsigned char)	.text	32	16326	0.20	81.38
OSSemBGive	.text	52	16378	0.20	81.38
OSTCBCur	.bss	4	16382	0.16	81.38

Tableau 5.14 Symboles en mémoire locale pour IDCT – 32K-2K/8-1K/8

Symbole	Section	Taille (octets)	Taille cumulative (octets)	Temps (%)	Temps cumulatif (%)
EntryClassPool::EntryClassPoolTbl	.bss	8356	8406	23.37	23.37
IDCT::thread_inverse_dct()	.text	2600	11006	16.73	40.10
OS_MemCopy	.text	52	11058	8.19	48.29
OSQPost	.text	280	11338	5.50	53.79
OSQAccept	.text	216	11554	4.85	58.64
OSEnterCritical	.text	44	11598	4.58	63.22
OSExitCritical	.text	32	11630	3.33	66.55
OSQTbl	.bss	568	12198	2.91	69.45
_stack_end	.stack	1344	13542	2.74	72.19
SWBus::ModuleWrite(unsigned char, unsigned char, unsigned char, unsigned long, void*, unsigned long)	.text	1808	15350	2.48	74.67
SWBus::ModuleRead(unsigned char, unsigned char, unsigned char, unsigned long, void*,	.text	1144	16494	2.31	76.98

unsigned long)					
OSCtxSw	.text	348	16842	2.13	79.11
OS_CPU_SR_Save	.text	20	16862	2.08	81.19
SWBus::m_FreeQPool	.bss	2076	18938	1.60	82.79
DisplayPrint	.text	220	19158	1.32	84.11
OSEventTbl	.bss	2640	21798	1.24	85.35
OS_Sched	.text	192	21990	1.18	86.53
OSSemPend	.text	308	22298	0.84	86.53
OS_CPU_SR_Restore	.text	8	22306	0.83	86.53
OSQueueRead	.text	28	22334	0.75	86.53
OSQueueWrite	.text	28	22362	0.75	86.53
OSDisableInt	.text	28	22390	0.74	86.53
OSEnableInt	.text	28	22418	0.74	86.53
OS_EventTaskRdy	.text	236	22654	0.73	86.53
OSQQuery	.text	236	22890	0.69	86.53
OSSemPost	.text	212	23102	0.53	86.53
OSSemAccept	.text	136	23238	0.48	86.53
OSTCBTbl	.bss	6756	29994	0.44	86.53
phash(unsigned int)	.text	44	30038	0.41	86.53
IQTZ::thread_inverse_quantize()	.text	360	30398	0.38	86.53
OS_EventTaskWait	.text	144	30542	0.35	86.53
unsigned char SpaceBaseModule::ModuleRead<short>(unsigned char, unsigned char, unsigned long, short*, unsigned char)	.text	132	30674	0.32	86.53
unsigned char SpaceBaseModule::ModuleWrite<short>(unsigned char, unsigned char, unsigned long, short*, unsigned char)	.text	132	30806	0.32	86.53
OSSemTake	.text	148	30954	0.32	86.53
SpaceBaseModule::ModuleReadInternal(unsigned char, unsigned char, unsigned long, void*, unsigned char)	.text	236	31190	0.25	86.53
SpaceBaseModule::ModuleWriteInternal(unsigned char, unsigned char, unsigned long, void*, unsigned char)	.text	300	31490	0.25	86.53
DisplayPutNumber	.text	428	31918	0.22	86.53
DirectLinkUtility::isDirectLinkTarget(unsigned char)	.text	32	31950	0.20	86.53
OSSemBGive	.text	52	32002	0.20	86.53
sc_module::GetVerbose()	.text	44	32046	0.19	86.53
SWBus::m_FreeSemPool	.bss	64	32110	0.16	86.53
OSTCBCur	.bss	4	32114	0.16	86.53
OSInit	.text	568	32682	0.15	86.53
cISSAdapterID	.data	4	32686	0.14	86.53
OSQueueGetBytes	.text	36	32722	0.11	86.53
OSRdyTbl	.bss	8	32730	0.10	86.53
Q::GetFreeBytes()	.text	32	32762	0.10	86.53
OSRdyGrp	.bss	4	32766	0.07	86.53

ANNEXE D

RÉSULTATS DE SIMULATION DE CACHE

D.1.Dhrystone

Tableau 5.15 Résultats de la cache d'instructions pour Dhrystone - 0K-2K/8-1K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.05	4.10	111603353	75.944	24.056
128/4	2.16	4.32	106002492	77.868	22.132
256/4	2.23	4.46	102499636	79.072	20.928
512/4	2.34	4.69	97595394	80.756	19.244
1024/4	2.99	5.97	76591187	87.973	12.027
2048/4	3.33	9.99	45786896	98.555	1.445
4096/4	1.83	11.00	41586413	99.999	0.001
8192/4	0.92	11.00	41585909	99.999	0.001
16384/4	0.48	11.00	41585902	99.999	0.001
32768/4	0.24	11.00	41585902	99.999	0.001
65536/4	0.12	11.00	41585902	99.999	0.001
64/8	2.02	4.04	113103710	84.364	15.636
128/8	2.29	4.58	99902731	87.250	12.750
256/8	2.42	4.85	94399926	88.453	11.547
512/8	2.51	5.02	91096384	89.175	10.825
1024/8	3.16	6.32	72391742	93.264	6.736
2048/8	4.97	9.95	45986627	99.037	0.963
4096/8	2.75	11.00	41586033	99.999	0.001
8192/8	1.37	11.00	41585439	99.999	0.001
16384/8	0.73	11.00	41585428	99.999	0.001
32768/8	0.37	11.00	41585428	99.999	0.001
65536/8	0.18	11.00	41585428	99.999	0.001

Tableau 5.16 Résultats de la cache de données pour Dhrystone - 0K-2K/8-1K8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	3.37	6.74	31025610	67.463	32.537
128/4	4.12	8.23	25421732	79.364	20.636
256/4	4.63	9.25	22615544	87.302	12.698
512/4	5.28	10.56	19814830	95.234	4.766
1024/4	5.68	11.36	18414354	98.407	1.593
2048/4	3.79	11.36	18414081	98.407	1.593
4096/4	1.97	11.81	17713759	99.994	0.006
8192/4	0.98	11.81	17713759	99.994	0.006
16384/4	0.51	11.81	17713724	99.994	0.006
32768/4	0.26	11.81	17713724	99.994	0.006
65536/4	0.13	11.81	17713724	99.994	0.006
64/8	2.79	5.57	37541542	69.045	30.955

128/8	3.51	7.01	29831609	80.948	19.052
256/8	4.51	9.01	23219773	88.096	11.904
512/8	4.73	9.46	22117287	92.064	7.936
1024/8	4.98	9.96	21015637	95.237	4.763
2048/8	5.56	11.12	18814559	98.410	1.590
4096/8	2.95	11.81	17713668	99.997	0.003
8192/8	1.48	11.81	17713668	99.997	0.003
16384/8	0.79	11.81	17713624	99.997	0.003
32768/8	0.39	11.81	17713624	99.997	0.003
65536/8	0.20	11.81	17713624	99.997	0.003

Tableau 5.17 Résultats de la cache d'instructions pour Dhrystone - 2K-1K/4-128/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.08	4.15	55130117	76.454	23.546
128/4	2.16	4.32	53030012	77.895	22.105
256/4	2.22	4.44	51628836	78.857	21.143
512/4	2.83	5.66	40426925	86.544	13.456
1024/4	4.33	8.67	26422368	96.154	3.846
2048/4	2.89	8.67	26421787	96.154	3.846
4096/4	1.83	11.00	20821262	99.998	0.002
8192/4	0.92	11.00	20821185	99.998	0.002
16384/4	0.48	11.00	20821157	99.998	0.002
32768/4	0.24	11.00	20821157	99.998	0.002
65536/4	0.12	11.00	20821157	99.998	0.002
64/8	2.00	4.01	57130454	84.143	15.857
128/8	2.32	4.63	49430278	87.505	12.495
256/8	2.42	4.85	47229475	88.466	11.534
512/8	2.98	5.96	38427759	92.310	7.690
1024/8	4.18	8.35	27422435	97.116	2.884
2048/8	4.18	8.35	27421698	97.116	2.884
4096/8	2.75	11.00	20820906	99.999	0.001
8192/8	1.37	11.00	20820829	99.999	0.001
16384/8	0.73	11.00	20820774	99.999	0.001
32768/8	0.37	11.00	20820774	99.999	0.001
65536/8	0.18	11.00	20820774	99.999	0.001

Tableau 5.18 Résultats de la cache de données pour Dhrystone - 2K-1K/4-128/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	3.66	7.32	3209607	71.493	28.507
128/4	6.49	12.99	1809551	99.944	0.056
256/4	6.49	12.99	1809376	99.946	0.054
512/4	6.49	12.99	1809243	99.947	0.053
1024/4	6.49	12.99	1809222	99.947	0.053
2048/4	4.33	12.99	1809201	99.948	0.052
4096/4	2.16	12.99	1809194	99.948	0.052
8192/4	1.08	12.99	1809194	99.948	0.052
16384/4	0.56	12.99	1809173	99.948	0.052
32768/4	0.29	12.99	1809173	99.948	0.052
65536/4	0.14	12.99	1809173	99.948	0.052

64/8	2.93	5.86	4009627	71.519	28.481
128/8	4.04	8.08	2909506	85.745	14.255
256/8	6.49	12.99	1809352	99.971	0.029
512/8	6.49	12.99	1809143	99.973	0.027
1024/8	6.49	12.99	1809121	99.973	0.027
2048/8	6.49	12.99	1809088	99.974	0.026
4096/8	3.25	12.99	1809077	99.974	0.026
8192/8	1.62	12.99	1809077	99.974	0.026
16384/8	0.87	12.99	1809044	99.974	0.026
32768/8	0.43	12.99	1809044	99.974	0.026
65536/8	0.22	12.99	1809044	99.974	0.026

Tableau 5.19 Résultats de la cache d'instructions pour Dhrystone - 4K-1K/4-64/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.19	4.37	52319298	78.365	21.635
128/4	2.19	4.37	52319130	78.365	21.635
256/4	2.31	4.62	49518038	80.288	19.712
512/4	2.88	5.76	39715791	87.018	12.982
1024/4	4.58	9.15	25011738	97.113	2.887
2048/4	3.05	9.15	25011248	97.114	2.886
4096/4	1.83	11.00	20810765	99.998	0.002
8192/4	0.92	11.00	20810611	99.998	0.002
16384/4	0.48	11.00	20810597	99.998	0.002
32768/4	0.24	11.00	20810597	99.998	0.002
65536/4	0.12	11.00	20810597	99.998	0.002
64/8	2.22	4.43	51620615	86.538	13.462
128/8	2.37	4.74	48319306	87.980	12.020
256/8	2.48	4.96	46118822	88.941	11.059
512/8	2.98	5.96	38416435	92.307	7.693
1024/8	4.54	9.08	25211661	98.076	1.924
2048/8	4.54	9.08	25211023	98.076	1.924
4096/8	2.75	11.00	20810330	99.999	0.001
8192/8	1.37	11.00	20810187	99.999	0.001
16384/8	0.73	11.00	20810165	99.999	0.001
32768/8	0.37	11.00	20810165	99.999	0.001
65536/8	0.18	11.00	20810165	99.999	0.001

Tableau 5.20 Résultats de la cache de données pour Dhrystone - 4K-1K/4-64/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	4.50	8.99	905177	99.871	0.129
128/4	4.50	8.99	905156	99.872	0.128
256/4	4.50	8.99	905156	99.872	0.128
512/4	4.50	8.99	905156	99.872	0.128
1024/4	4.50	8.99	905156	99.872	0.128
2048/4	3.00	8.99	905142	99.872	0.128
4096/4	1.50	8.99	905142	99.872	0.128
8192/4	0.75	8.99	905142	99.872	0.128
16384/4	0.39	8.99	905128	99.873	0.127
32768/4	0.20	8.99	905128	99.873	0.127

65536/4	0.10	8.99	905128	99.873	0.127
64/8	4.50	8.99	905255	99.933	0.067
128/8	4.50	8.99	905222	99.934	0.066
256/8	4.50	8.99	905222	99.934	0.066
512/8	4.50	8.99	905189	99.935	0.065
1024/8	4.50	8.99	905189	99.935	0.065
2048/8	4.50	8.99	905167	99.935	0.065
4096/8	2.25	8.99	905167	99.935	0.065
8192/8	1.12	8.99	905167	99.935	0.065
16384/8	0.60	8.99	905145	99.936	0.064
32768/8	0.30	8.99	905145	99.936	0.064
65536/8	0.15	8.99	905145	99.936	0.064

Tableau 5.21 Résultats de la cache d'instructionss pour Dhrystone - 8K-1K/4-64/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	1.69	3.37	3998	67.700	32.300
128/4	1.70	3.41	3956	68.189	31.811
256/4	1.77	3.54	3809	69.902	30.098
512/4	1.86	3.73	3620	72.104	27.896
1024/4	2.15	4.30	3137	77.732	22.268
2048/4	1.52	4.55	2962	79.772	20.228
4096/4	0.78	4.67	2885	80.669	19.331
8192/4	0.40	4.79	2815	81.485	18.515
16384/4	0.21	4.80	2808	81.566	18.434
32768/4	0.11	4.80	2808	81.566	18.434
65536/4	0.05	4.80	2808	81.566	18.434
64/8	1.65	3.30	4086	78.793	21.207
128/8	1.69	3.38	3987	79.527	20.473
256/8	1.77	3.54	3811	80.832	19.168
512/8	1.86	3.72	3624	82.219	17.781
1024/8	2.16	4.31	3129	85.889	14.111
2048/8	2.34	4.69	2876	87.765	12.235
4096/8	1.23	4.91	2744	88.744	11.256
8192/8	0.63	5.04	2678	89.233	10.767
16384/8	0.34	5.06	2667	89.315	10.685
32768/8	0.17	5.06	2667	89.315	10.685
65536/8	0.08	5.06	2667	89.315	10.685

Tableau 5.22 Résultats de la cache de données pour Dhrystone - 8K-1K/4-64/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	4.50	8.99	905266	99.869	0.131
128/4	4.50	8.99	905231	99.870	0.130
256/4	4.50	8.99	905231	99.871	0.129
512/4	4.50	8.99	905203	99.871	0.129
1024/4	4.50	8.99	905203	99.871	0.129
2048/4	3.00	8.99	905189	99.872	0.128
4096/4	1.50	8.99	905189	99.872	0.128
8192/4	0.75	8.99	905189	99.872	0.128
16384/4	0.39	8.99	905161	99.873	0.127

32768/4	0.20	8.99	905161	99.873	0.127
65536/4	0.10	8.99	905161	99.873	0.127
64/8	4.50	8.99	905310	99.932	0.068
128/8	4.50	8.99	905244	99.934	0.066
256/8	4.50	8.99	905244	99.935	0.065
512/8	4.50	8.99	905211	99.935	0.065
1024/8	4.50	8.99	905211	99.935	0.065
2048/8	4.50	8.99	905189	99.936	0.064
4096/8	2.25	8.99	905189	99.936	0.064
8192/8	1.12	8.99	905189	99.936	0.064
16384/8	0.60	8.99	905156	99.937	0.063
32768/8	0.30	8.99	905156	99.937	0.063
65536/8	0.15	8.99	905156	99.937	0.063

D.2.IDCT

Tableau 5.23 Résultats de la cache d'instructions pour IDCT - 0K-1K/4-1K/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	1.95	3.90	81049663	70.386	29.614
128/4	2.02	4.04	78275598	71.888	28.112
256/4	2.32	4.63	68350172	77.264	22.736
512/4	2.45	4.90	64610842	79.290	20.710
1024/4	2.91	5.81	54469529	84.783	15.217
2048/4	2.08	6.24	50721330	86.813	13.187
4096/4	1.29	7.72	41007010	92.075	7.925
8192/4	0.69	8.28	38238664	93.574	6.426
16384/4	0.44	10.10	31330882	97.316	2.684
32768/4	0.24	10.76	29410887	98.356	1.644
65536/4	0.13	12.00	26383856	99.995	0.005
64/8	1.78	3.55	89113750	78.375	21.625
128/8	1.84	3.67	86212951	79.375	20.625
256/8	2.13	4.27	74158337	83.530	16.470
512/8	2.27	4.53	69846568	85.016	14.984
1024/8	2.67	5.35	59174676	88.694	11.306
2048/8	2.89	5.78	54794300	90.204	9.796
4096/8	1.93	7.71	41049492	94.942	5.058
8192/8	1.03	8.28	38228982	95.914	4.086
16384/8	0.68	10.17	31128515	98.361	1.639
32768/8	0.36	10.72	29531678	98.912	1.088
65536/8	0.20	12.00	26382499	99.997	0.003

Tableau 5.24 Résultats de la cache de données pour IDCT - 0K-1K/4-1K/4

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.61	5.22	31163704	60.996	39.004
128/4	3.49	6.97	23341582	77.817	22.183
256/4	4.27	8.54	19055909	86.792	13.208
512/4	4.53	9.06	17973807	90.283	9.717
1024/4	4.82	9.64	16889829	92.699	7.301
2048/4	3.39	10.18	15986108	94.246	5.754
4096/4	1.70	10.22	15932089	94.356	5.644
8192/4	0.93	11.14	14607500	97.396	2.604
16384/4	0.49	11.33	14362227	97.805	2.195
32768/4	0.27	12.15	13393707	99.907	0.093
65536/4	0.14	12.20	13337672	99.989	0.011
64/8	1.90	3.79	42942129	57.691	42.309
128/8	2.63	5.26	30967430	75.173	24.827
256/8	3.42	6.84	23807376	85.117	14.883
512/8	3.88	7.76	20981795	90.018	9.982
1024/8	4.15	8.31	19591538	92.141	7.859
2048/8	4.49	8.98	18135853	93.780	6.220
4096/8	2.34	9.38	17357394	94.683	5.317
8192/8	1.33	10.66	15275930	97.528	2.472
16384/8	0.73	10.93	14890512	97.938	2.062
32768/8	0.40	12.14	13403774	99.933	0.067
65536/8	0.20	12.20	13337675	99.994	0.006

Tableau 5.25 Résultats de la cache d'instructions pour IDCT - 2K-2K/8-1K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.02	4.03	38096414	71.764	28.236
128/4	2.19	4.38	35058939	75.154	24.846
256/4	2.46	4.93	31168297	79.497	20.503
512/4	2.59	5.19	29597329	81.250	18.750
1024/4	3.27	6.53	23512768	88.041	11.959
2048/4	2.35	7.05	21793820	89.960	10.040
4096/4	1.29	7.73	19858761	92.120	7.880
8192/4	0.74	8.91	17229785	95.054	4.946
16384/4	0.47	10.92	14069621	98.582	1.418
32768/4	0.26	11.75	13074263	99.693	0.307
65536/4	0.13	11.99	12807395	99.990	0.010
64/8	2.00	4.01	38332898	81.863	18.137
128/8	2.21	4.42	34729936	84.423	15.577
256/8	2.58	5.15	29806380	87.920	12.080
512/8	2.75	5.51	27888827	89.282	10.718
1024/8	3.35	6.70	22909358	92.819	7.181
2048/8	3.59	7.17	21409475	93.884	6.116
4096/8	1.94	7.77	19756758	95.058	4.942
8192/8	1.12	8.95	17156523	96.905	3.095
16384/8	0.72	10.75	14293476	98.938	1.062
32768/8	0.39	11.63	13203057	99.713	0.287
65536/8	0.20	11.99	12806100	99.995	0.005

Tableau 5.26 Résultats de la cache de données pour IDCT - 2K-2K/8-1K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.74	5.49	26764838	63.455	36.545
128/4	3.78	7.55	19447829	80.884	19.116
256/4	4.38	8.75	16787101	87.944	12.056
512/4	4.66	9.32	15764898	91.412	8.588
1024/4	4.97	9.94	14780425	94.137	5.863
2048/4	3.55	10.65	13793075	96.077	3.923
4096/4	1.80	10.80	13606735	96.665	3.335
8192/4	0.98	11.76	12492461	99.384	0.616
16384/4	0.52	11.93	12317202	99.724	0.276
32768/4	0.27	12.04	12200442	99.988	0.012
65536/4	0.13	12.04	12200442	99.988	0.012
64/8	1.93	3.87	38000457	57.525	42.475
128/8	2.74	5.48	26831387	76.654	23.346
256/8	3.69	7.37	19930284	87.671	12.329
512/8	4.17	8.33	17633869	92.446	7.554
1024/8	4.55	9.10	16144161	94.824	5.176
2048/8	4.96	9.93	14801127	96.312	3.688
4096/8	2.60	10.42	14101593	97.404	2.596
8192/8	1.43	11.48	12802086	99.251	0.749
16384/8	0.79	11.83	12416591	99.704	0.296
32768/8	0.40	12.04	12200452	99.994	0.006
65536/8	0.20	12.04	12200452	99.994	0.006

Tableau 5.27 Résultats de la cache d'instructions pour IDCT - 4K-2K/8-1K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	1.95	3.89	22918961	70.264	29.736
128/4	2.10	4.20	21275004	73.421	26.579
256/4	2.12	4.25	21013890	73.923	26.077
512/4	2.24	4.49	19883908	76.093	23.907
1024/4	2.36	4.73	18879933	78.022	21.978
2048/4	1.75	5.25	16985908	81.660	18.340
4096/4	0.99	5.95	15008562	85.458	14.542
8192/4	0.53	6.39	13969076	87.454	12.546
16384/4	0.40	9.13	9778274	95.504	4.496
32768/4	0.24	10.99	8117881	98.693	1.307
65536/4	0.13	11.99	7445167	99.985	0.015
64/8	1.80	3.61	24750696	78.838	21.162
128/8	2.02	4.04	22090049	82.090	17.910
256/8	2.12	4.23	21086376	83.317	16.683
512/8	2.31	4.61	19344394	85.446	14.554
1024/8	2.46	4.92	18131127	86.929	13.071
2048/8	2.71	5.43	16444376	88.991	11.009
4096/8	1.53	6.13	14549131	91.307	8.693
8192/8	0.83	6.63	13456633	92.643	7.357
16384/8	0.62	9.36	9536068	97.435	2.565
32768/8	0.37	11.19	7972715	99.346	0.654
65536/8	0.20	11.99	7444099	99.992	0.008

Tableau 5.28 Résultats de la cache de données pour IDCT - 4K-2K/8-1K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.74	5.47	27069888	63.198	36.802
128/4	3.76	7.52	19694604	80.538	19.462
256/4	4.36	8.72	16983462	87.575	12.425
512/4	4.63	9.26	15995027	91.187	8.813
1024/4	4.99	9.97	14852599	94.230	5.770
2048/4	3.56	10.68	13865242	96.155	3.845
4096/4	1.81	10.83	13672084	96.707	3.293
8192/4	0.98	11.77	12585789	99.360	0.640
16384/4	0.52	11.93	12410516	99.698	0.302
32768/4	0.27	12.06	12280512	99.988	0.012
65536/4	0.13	12.06	12280512	99.988	0.012
64/8	1.84	3.67	40336374	55.602	44.398
128/8	2.56	5.11	28961373	72.892	27.108
256/8	3.59	7.18	20637156	86.648	13.352
512/8	4.12	8.24	17965058	91.865	8.135
1024/8	4.56	9.13	16229819	94.765	5.235
2048/8	5.03	10.05	14731784	96.597	3.403
4096/8	2.59	10.36	14301024	97.297	2.703
8192/8	1.45	11.61	12759242	99.354	0.646
16384/8	0.79	11.86	12483835	99.692	0.308
32768/8	0.40	12.06	12280544	99.994	0.006
65536/8	0.20	12.06	12280544	99.994	0.006

Tableau 5.29 Résultats de la cache d'instructions pour IDCT - 8K-2K/8-2K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.18	4.37	16622505	75.055	24.945
128/4	2.21	4.43	16407976	75.561	24.439
256/4	2.25	4.49	16167911	76.128	23.872
512/4	2.31	4.61	15745447	77.125	22.875
1024/4	2.46	4.93	14734514	79.511	20.489
2048/4	2.01	6.04	12027929	85.899	14.101
4096/4	1.27	7.62	9532093	91.789	8.211
8192/4	0.74	8.93	8134907	95.086	4.914
16384/4	0.45	10.32	7041087	97.668	2.332
32768/4	0.25	11.26	6452520	99.057	0.943
65536/4	0.13	11.99	6060065	99.983	0.017
64/8	2.17	4.35	16704654	84.002	15.998
128/8	2.29	4.58	15871679	85.253	14.747
256/8	2.33	4.65	15605765	85.653	14.347
512/8	2.38	4.76	15262070	86.169	13.831
1024/8	2.52	5.04	14401221	87.462	12.538
2048/8	2.99	5.98	12153008	90.839	9.162
4096/8	1.94	7.76	9366015	95.024	4.976
8192/8	1.14	9.13	7953912	97.145	2.855
16384/8	0.69	10.37	7006163	98.568	1.432
32768/8	0.38	11.25	6455580	99.395	0.605
65536/8	0.20	11.99	6059063	99.991	0.009

Tableau 5.30 Résultats de la cache de données pour IDCT - 8K-2K/8-2K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	3.21	6.42	20736572	71.161	28.839
128/4	3.98	7.95	16743772	82.511	17.489
256/4	4.38	8.76	15205109	87.895	12.105
512/4	4.64	9.28	14352957	91.654	8.346
1024/4	4.88	9.77	13628632	94.076	5.924
2048/4	3.53	10.60	12554482	96.236	3.764
4096/4	1.79	10.73	12402568	96.595	3.405
8192/4	0.98	11.76	11316357	99.346	0.654
16384/4	0.52	11.95	11141161	99.721	0.279
32768/4	0.27	12.07	11031401	99.988	0.012
65536/4	0.13	12.07	11031401	99.988	0.012
64/8	2.28	4.56	29179303	66.595	33.405
128/8	3.07	6.15	21655435	79.823	20.177
256/8	3.84	7.68	17333557	88.737	11.263
512/8	4.25	8.51	15650458	93.004	6.996
1024/8	4.54	9.07	14677464	94.881	5.119
2048/8	5.02	10.05	13249521	96.764	3.236
4096/8	2.56	10.22	13020567	97.110	2.890
8192/8	1.45	11.59	11489873	99.349	0.651
16384/8	0.79	11.87	11214554	99.724	0.276
32768/8	0.40	12.07	11031404	99.994	0.006
65536/8	0.20	12.07	11031404	99.994	0.006

Tableau 5.31 Résultats de la cache d'instructions pour IDCT - 16K-2K/8-2K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.01	4.02	10776569	71.671	28.329
128/4	2.04	4.09	10604187	72.353	27.647
256/4	2.07	4.13	10490367	72.803	27.197
512/4	2.11	4.21	10285757	73.612	26.388
1024/4	2.18	4.35	9961272	74.895	25.105
2048/4	1.77	5.30	8179324	81.942	18.058
4096/4	1.09	6.52	6650377	87.988	12.012
8192/4	0.61	7.31	5932912	90.825	9.175
16384/4	0.42	9.72	4460567	96.647	3.353
32768/4	0.27	11.98	3619776	99.972	0.028
65536/4	0.13	11.98	3619566	99.973	0.027
64/8	1.99	3.98	10889884	81.687	18.313
128/8	2.05	4.09	10590409	82.441	17.559
256/8	2.09	4.19	10346572	83.055	16.945
512/8	2.15	4.30	10080680	83.724	16.276
1024/8	2.25	4.51	9616018	84.893	15.107
2048/8	2.67	5.33	8127256	88.639	11.361
4096/8	1.61	6.44	6727528	92.162	7.838
8192/8	0.92	7.35	5898964	94.247	5.753
16384/8	0.64	9.59	4521753	97.712	2.288
32768/8	0.40	11.98	3618840	99.984	0.016
65536/8	0.20	11.98	3618620	99.985	0.015

Tableau 5.32 Résultats de la cache de données pour IDCT - 16K-2K/8-2K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.57	5.13	4475181	64.537	35.463
128/4	3.07	6.14	3739642	73.686	26.314
256/4	3.76	7.53	3052473	81.699	18.301
512/4	4.57	9.15	2511793	89.109	10.891
1024/4	5.09	10.18	2258092	92.795	7.205
2048/4	3.91	11.72	1959969	96.119	3.881
4096/4	2.16	12.95	1774420	98.912	1.088
8192/4	1.08	12.95	1774259	98.921	1.079
16384/4	0.58	13.35	1721276	99.932	0.068
32768/4	0.30	13.35	1721262	99.932	0.068
65536/4	0.15	13.35	1721262	99.932	0.068
64/8	1.79	3.58	6424425	62.250	37.750
128/8	2.26	4.53	5075495	73.396	26.604
256/8	2.85	5.71	4024214	81.358	18.642
512/8	3.45	6.90	3328255	87.031	12.969
1024/8	4.11	8.22	2794854	90.960	9.040
2048/8	5.17	10.33	2223723	94.947	5.053
4096/8	3.22	12.87	1785494	99.109	0.891
8192/8	1.61	12.87	1785274	99.118	0.882
16384/8	0.89	13.35	1721298	99.965	0.035
32768/8	0.44	13.35	1721276	99.965	0.035
65536/8	0.22	13.35	1721276	99.965	0.035

Tableau 5.33 Résultats de la cache d'instructions pour IDCT - 32K-2K/8-2K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.41	4.83	485250	78.788	21.212
128/4	2.47	4.94	473945	79.615	20.385
256/4	2.72	5.45	430125	82.820	17.180
512/4	3.62	7.24	323788	90.599	9.401
1024/4	3.81	7.63	307142	91.817	8.183
2048/4	3.77	11.32	206965	99.145	0.855
4096/4	1.90	11.43	205103	99.282	0.718
8192/4	0.96	11.57	202583	99.466	0.534
16384/4	0.51	11.64	201358	99.556	0.444
32768/4	0.26	11.67	200861	99.592	0.408
65536/4	0.13	11.68	200581	99.612	0.388
64/8	2.54	5.08	460932	87.633	12.367
128/8	2.57	5.14	456334	87.847	12.153
256/8	2.85	5.71	410508	89.981	10.019
512/8	3.65	7.30	320979	94.148	5.852
1024/8	3.95	7.91	296405	95.292	4.708
2048/8	5.68	11.36	206205	99.492	0.508
4096/8	2.87	11.47	204324	99.579	0.421
8192/8	1.45	11.61	201783	99.697	0.303
16384/8	0.78	11.68	200606	99.752	0.248
32768/8	0.39	11.71	200122	99.775	0.225
65536/8	0.20	11.72	199880	99.786	0.214

Tableau 5.34 Résultats de la cache de données pour IDCT - 32K-2K/8-2K/8

Cache	Résultat	Accélération	Temps (cycles)	Succès (%)	Échecs (%)
64/4	2.14	4.29	1165057	54.926	45.074
128/4	2.56	5.12	975840	64.054	35.946
256/4	4.18	8.36	597462	82.307	17.693
512/4	5.46	10.92	457301	92.448	7.552
1024/4	5.63	11.27	443259	93.126	6.874
2048/4	4.92	14.77	338175	99.885	0.115
4096/4	2.46	14.77	338098	99.890	0.110
8192/4	1.23	14.77	338063	99.891	0.109
16384/4	0.64	14.77	338056	99.892	0.108
32768/4	0.33	14.77	338056	99.892	0.108
65536/4	0.16	14.77	338056	99.892	0.108
64/8	1.31	2.63	1902086	46.861	53.139
128/8	1.42	2.84	1758899	51.258	48.742
256/8	2.10	4.21	1186250	68.838	31.162
512/8	2.92	5.83	855909	78.978	21.022
1024/8	2.99	5.99	833722	79.659	20.341
2048/8	3.73	7.47	668590	86.418	13.582
4096/8	3.69	14.77	338128	99.942	0.058
8192/8	1.85	14.77	338084	99.944	0.056
16384/8	0.98	14.77	338073	99.944	0.056
32768/8	0.49	14.77	338073	99.944	0.056
65536/8	0.25	14.77	338073	99.944	0.056