République Algérienne Démocratique et Populaire
Ministère de L'Enseignement Supérieur et de la Recherche Scientifique
Université Mouloud Mammeri Tizi-Ouzou
Faculté de Génie Electrique & d'Informatique
Département d'Informatique

MEMOIRE

En vue de** l'obtention **de Diplôme de Master2 en informatique

Thème

Optimisation de la mémoire pour les systèmes embarqués

Dirigé et proposé par :

Mr HEMDANI CHABANE

Réalisé par :

M^elle^ MEHLEB LILA

Promotion 2010/2011

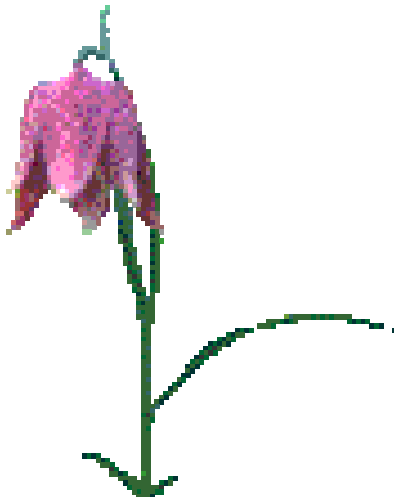
Remerciements

J’exprime mes remerciements et mes gratitudes à mon promoteur Mr HEMDANI CHABANE pour tous ses aides ainsi que ses conseils tout au long de ce travail.

Mes plus vifs remerciements vont aussi aux membres de jury pour m’avoir fait l’honneur de juger ce travail.

Je réserve ici une place particulière pour remercier vivement ma famille pour son affection et son soutien continu. Et à tous ceux qui, d’une manière ou d’une autre, n’ont *aidé et encouragé à la réalisation de ce modeste travail.*

Merci à tous et à toutes.



e *Dédicaces* f

A Mes chers parents,
A mes frères et sœurs ainsi que toute ma famille,
A tous mes amis(e),

Liste des figures

Chapitre III

Chapitre IV

Chapitre V

Liste des tableaux

Chapitre I

Chapitre II

Introduction général

Chapitre I
Etat de l'art

Chapitre II
Techniques de gestion de la mémoire SPM

Chapitre III
La solution d' Avissar et Barua

Chapitre IV
Conception

Chapitre V
Implémentation

Conclusion général

Introduction générale

Introduction

La conception de systèmes embarqués ou enfouis, appelés "*System on Chip – SoC*" utilise une méthode conjointe dans laquelle la mise en place de la plate-forme matérielle et celle des parties logicielles sont soumises à des interactions très fortes. En effet, ces systèmes (téléphones portables, agendas personnels électroniques ou les systèmes de guidage et de vidéo embarqués dans les engins mobiles) doivent répondre à des contraintes spécifiques de place, de consommation et de performance. De plus, près de la moitié de la surface des systèmes actuels est dédiée à la mémoire, il est donc extrêmement important de prendre en compte les optimisations possibles sur la mémoire et les communications dans les hiérarchies de mémoires ou bien vers les réseaux afin de contrôler la taille et la consommation de ces systèmes.

Les accès à la mémoire sont beaucoup plus consommateurs d'énergie que les opérations de contrôle ou les opérations arithmétiques. Ainsi, des expérimentations ont, par exemple, montré qu'un accès à la mémoire externe peut coûter jusqu'à 33 fois plus en consommation qu'une addition sur 16 bits. De plus, plus l'accès mémoire est lointain, plus le coût est important : l'accès à une mémoire interne coûte 3 fois moins que l'accès à une mémoire externe au circuit.

Dans ce cadre, les mémoires Scratch-Pad (SPM) sont de plus en plus utilisées et commencent à concurrencer les traditionnelles mémoires caches. La *scratch-pad memory*, est une petite zone mémoire rapide (SRAM) gérée directement par le compilateur, et non pas par le matériel. Ses avantages sont nombreux : taille plus faible de 34% par rapport à un cache, consommation énergétique moindre de 40%, bornes temps réel plus précises [Banakar *et autre*, 2002]. Le compilateur cherche donc à gérer cette zone en y plaçant les données les plus fréquemment accédées. Diverses approches existent pour gérer les *scratch-pads*. Les approches statiques [Avissar *et autre*, 2002], dont la gestion (choix de placement) est faite à la compilation, offrent de bonnes performance et de bonnes caractéristiques à la faible consommation d'énergie.

Pour ce thème de Master, je me suis plus particulièrement intéressée à l'étude de l'optimisation de la mémoire pour les systèmes embarqués. Dans ce cadre, les mémoires dites "Scratch-Pad" (SPM) présentent un intérêt considérable par la facilité et la certitude avec lesquelles leur comportement (temps de réponse notamment) peut être

prédit. Le travail que je dois effectuer dans le cadre de ce diplôme comprend globalement cinq chapitres :

Le premier chapitre; l'état de l'art, consiste à se familiariser avec ce domaine et à en approfondir l'état de l'art. Le deuxième chapitre; techniques de gestions de la mémoire SPM, doit porter sur l'étude de l'optimisation de codes et la gestion de la mémoire (quelles données allouer à quel type de mémoire) pour la gestion mémoire basse consommation. Le troisième chapitre, la méthode d'Oren Avissar, présente une gestion efficace de compilateur, on utilise la solution de programmation linéaire 0/1 (integer linair programming). Le quatrième chapitre, c'est la conception de notre travail qui se situe dans le cadre d'implémentation de méthode d'Avissar et Barua pour les variables globales. Le cinquième chapitre est la partie implémentation et simulation. Et on termine par une conclusion générale.

.

Chapitre I

Etat de l'art

Introduction

La taille et la consommation d'énergie du code dans les systèmes embarquées sont deux critères très importants pour une question de coût, d'autonomie et de dissipation d'énergie. L'optimisation de la taille des applications embarquées est importante parce que souvent, il n'y a pas de disque dur et toutes les applications doivent tenir dans la mémoire de taille limitée pour des raisons économiques. D'autre part, étant donné que les besoins en énergie des processeurs et des systèmes embarqués augmentent plus rapidement que la capacité des sources d'énergie portables, il est très important de minimiser la consommation d'énergie. De plus, les besoins en performance des systèmes embarqués augmentent sans cesse, et les processeurs embarqués deviennent de plus en plus sophistiqués. Il faut donc pouvoir optimiser simultanément la taille, la consommation d'énergie et la performance des codes embarqués.

1. Evolution mémoire et impact sur la consommation

La mémoire est un composant essentiel dans tous les circuits intégrés, et ce quelque soit le type : ROM, SRAM, DRAM, Flash et Ferro-Magnetique. Deux exemples décrivent bien cette nouvelle tendance : les baladeurs de type MP3 qui s'appuient sur le stockage d'un album musical non pas sur un disque comme auparavant, mais sur l'utilisation de mémoire Flash d'environ 32Mo actuellement. Ceci laisse prévoir une utilisation de ce principe pour des applications encore plus gourmandes en termes de capacité de stockage comme la mémorisation de séquences vidéo par exemple. L'autre exemple est celui de la nouvelle génération de processeurs qui gèrent la mémoire d'une manière complètement différente puisqu'elle est au cœur du système. C'est ce que proposent les projets IRAM (Intelligent RAM) et SSMP (Stanford Smart Memories Project) définis respectivement à Berkeley et Stanford. Dans ces projets, la mémoire est intégrée sur la même puce que le microprocesseur.

En effet, malgré les progrès de types de mémoires, celle-ci a toujours des contraintes de la puissance, d'énergie, de taille et le problème de mur de la mémoire (Memory Wall Problem) [Manish Verma et Peter.2007] par rapport aux exigences des systèmes embarqués. Les recours à des solutions de performance des systèmes embarqués occupent aujourd'hui une part importante. Aussi, réduire la consommation

des mémoires permettrait de réduire la consommation totale des circuits. L'accroissement de l'autonomie des systèmes portatifs passe par la diminution de la consommation des circuits intégrés, sachant qu'il y a pour le moment, peu à attendre des améliorations sur les batteries. Puisque les mémoires sur une puce en constituent l'essentiel de la surface et du temps d'activation des systèmes embarqués, on voit que si l'on parvient à réduire la consommation des mémoires, on pourra réduire la consommation totale du circuit.

2. Sources de consommation au sein des mémoires

On distingue deux types de consommation d'énergie :

- Consommation dynamique : c'est la puissance consommée par la mémoire, échanges de données entre un élément de calcul et l'élément de stockage (accès aux données mémoire on-chip et en mémoire off-chip). [Olivier, 2005]

- Consommation statique : elle représente les courants de fuite, ils deviennent critiques quand le circuit passe beaucoup de temps en mode repos ou bien lorsque le volume de la mémoire de plus en plus augmente. [Olivier, 2005]

3. Les hiérarchies mémoires

Dans un système embarqué une hiérarchie mémoire est l'ensemble des mécanismes automatiques de gestion de la mémoire qui permettent d'optimiser des paramètres comme le coût, l'efficacité du système ou encore sa consommation d'énergie.

Assez nombreux travaux de recherche se focalisent sur les techniques d'optimisation d'une hiérarchie mémoire [Stéphane et autre, 2005-2009]. L'objectif est d'automatiser l'optimisation du paramétrage d'algorithmes de suivi de séquences d'accès mémoire utilisés par une hiérarchie mémoire. Une stratégie de gestion des accès mémoire est basée sur l'analyse dynamique qui intègre des mécanismes de prédiction.

4. Critères et techniques d'optimisation des mémoires

Réduire la consommation d'énergie des systèmes embarqués est un sujet d'actualité dont les enjeux sont cruciaux, plusieurs options pour économiser l'énergie existent [Zendra, 2006]. Ces différentes approches peuvent être divisées en deux catégories : optimisations matérielles et optimisations logicielles. Les techniques

d'optimisations matérielles, hors du contexte de ce mémoire, ne seront pas traitées, mais une quantité importante de littérature les concernant est disponible [Egger *et autre.* 2006] [Hallnor et Reinhardt, 2000]. Notons que quelques travaux combinent de façon intéressante les techniques d'optimisations matérielles et logicielles, comme [Poletti *et autre*, 2004] (qui s'appuie sur un Direct Memory Access ou DMA pour réduire le coût de copie entre les mémoires SPM et DRAM), ou encore [Benini *et autre*, 2000].

Les techniques logicielles se focalisent sur l'optimisation des compilateurs. Ceux-ci permettent de dédier beaucoup de ressources (temps, CPU, mémoire) à des optimisations, ce qui est impossible dans le matériel. Ils peuvent ainsi "connaître le futur" d'un programme, via des analyses puissantes (analyses globales). Un compilateur optimisant intègre un survol de solutions qui existent dans le domaine de la compilation de code optimisé pour la basse consommation énergétique. Parmi ces optimisations on peut citer :

4.1. Les optimisations de bas niveau

Celles-ci sont anciennes, nombreuses et d'un usage commun dans les compilateurs. Certaines peuvent avoir un effet bénéfique, quoiqu'en général relativement limité, sur la consommation énergétique. Notons qu'elles sont souvent très dépendantes du processeur. Nous en présentons ici quelques une :

- ***Énergie de transition et commutation (switching activity)***

En réordonnant les instructions, il est possible de diminuer les commutations entre les instructions successives, ce qui diminue la puissance consommée [Graybill *et autre*, 2002]. De même, des encodages judicieux des données (voire des instructions) permettent de diminuer l'énergie consommée.

On peut par exemple économiser l'énergie due aux changements dans les champs registres des instructions, en effectuant après compilation et allocation de registres, une passe de renommage de registres [Kandemir *et autre*, 2000]. Ceci résulte en une réduction de 11% de l'activité de commutation sur le champ registre de l'instruction [Woo *et autre*, 2001].

- ***Dépliage de boucles***

Le *loop unrolling* permet de diminuer le surcoût dû au contrôle de boucle et donc d'exécuter globalement moins d'instructions. Cette optimisation classique pour la

vitesse se traduit donc aussi en des gains d'énergie, y compris au niveau du cœur d'exécution (ce qui est très positif pour y éviter les points chauds). Bien entendu, il convient d'être vigilant à l'accroissement de la taille du code, qui augmente la consommation et peut ne pas être acceptable dans un système embarqué.

4.2. Les optimisations de haut niveau

Elles ont souvent un impact bien plus fort que les optimisations de bas niveau. Les optimisations de compilation liées à la mémoire sont potentiellement très efficaces du point de vue de la consommation.

- ***Compactage***

La fusion de variables (*variable coalescing*), qui fait coexister dans un même emplacement mémoire plusieurs (petites) données permet d'utiliser moins de place, ce qui diminue la consommation énergétique [Zhuang *et autre*, 2003]. De façon semblable, l'analyse de la durée de vie des données permet le partage d'un même emplacement entre des données qui ne coexistent pas temporellement.

A plus large échelle, la compression des données [Zhang *et autre*, 2002], bien que coûteuse en énergie et en temps à l'exécution, permet également de diminuer la taille de la mémoire active et facilite donc la mise au repos de bancs mémoire non utilisés. Bien entendu, le surcoût de (dé)compression doit être limité et mis en balance avec le gain de place et d'énergie. Cette technique est particulièrement appropriée pour des données à longue durée de vie mais accédées peu fréquemment.

- ***Mise en hibernation dynamique des ressources***

La mise en hibernation dynamique des ressources matérielles (ou plus largement l'exploitation de leurs modes basse énergie) permet des gains en énergie considérables. Par exemple, [Hom *et autre*, 2001] montre que prédire les événements de *swapping* mémoire permet de n'activer la ressource pour le *swap* (dans leur cas, une carte réseau sans fil) que lorsque c'est nécessaire, et amène une économie d'énergie de 20%.

- ***Mémoires scratch-pad***

Les caches aident à améliorer la vitesse des programmes. Cependant, ils ne sont pas très adaptés aux systèmes embarqués: ils augmentent la taille du système et sa consommation énergétique (avec la zone cache et sa logique de gestion) et sont peu prédictibles, ce qui est gênant pour les systèmes temps réel. Une alternative intéressante est la *scratch-pad memory*, petite zone mémoire rapide (SRAM) gérée directement par

le compilateur, et non pas par le matériel. Ses avantages sont nombreux : taille plus faible de 34% par rapport à un cache, consommation énergétique moindre de 40%, bornes temps réel plus précises (Banakar *et autre,* 2002). Le compilateur cherche donc à gérer cette zone en y plaçant les données les plus fréquemment accédées. Diverses approches existent pour gérer les *scratch-pads*. Les approches statiques (Avissar *et autre,* 2002), dont la gestion (choix de placement) est faite à la compilation, offrent de bonnes performances et de bonnes caractéristiques temps réel.

Les approches dynamiques prennent en compte des situations plus complexes (par exemple deux boucles successives sur des tableaux trop gros pour être placés en même temps en *scratch-pad*, mais qui peuvent l'être successivement). En revanche, elles sont plus difficiles à prendre en compte dans le cadre du temps réel. Elles ont aussi un surcoût (temps et énergie) non négligeable à l'exécution, qui vient notamment des coûts de transfert des données entre *scratch-pad* et mémoire principale décidés par le compilateur. Ce dernier effectue ses choix en fonction du coût des transferts et de la fréquence future d'utilisation des données, que ne connait pas un cache matériel.

Le compilateur peut ainsi plus efficacement placer les bonnes données en mémoire rapide, et peut aussi réutiliser immédiatement la mémoire rapide occupée par une donnée qui ne sera plus utilisée. Pour éviter les surcoûts dus aux transferts de données entre mémoire principale et *scratch-pad,* il est intéressant que le compilateur alloue directement en SRAM certaines parties du tas. Les gains en énergie rapportés dans [Dominguez *et autre,* 2005] sont d'environ 40% par rapport à un placement en mémoire.

5. Estimation de la consommation d'énergie

Certains modèles d'estimation de la consommation d'énergie considèrent seulement les mémoires caches. C'est le cas de [Shiue et Chakrabarti, 1999]. Dans cet article, les auteurs comparent la pertinence des optimisations des performances (taille du cache, nombre de cycles processeur et consommation d'énergie) entre trois techniques, à savoir allocation mémoire *off-chip*, mosaïque (*tiling*) et associativité (*set associativity*). Ils déterminent la configuration optimale de la mémoire en fonction de la performance qu'ils cherchent à optimiser. Leurs résultats complètent ceux déjà obtenus par [Panda *et autre,* 1997a; Panda *et autre,* 1997] en considérant la consommation d'énergie comme performance à optimiser. Leur modèle d'énergie ne prend en compte

que les lectures car elles sont considérées comme étant celles qui consomment le plus d'énergie dans la mémoire cache d'après [Hennessy et Patterson, 2003]. Ne considérant que l'énergie de lecture que ce soit succès de lecture (*read hit*) ou que ce soit échec de lecture (*read miss*), ce modèle est très simple.

D'autres modèles considèrent à la fois les mémoires caches et SPMs. [Banakar *et autre*, 2002] en est un exemple. Cet article compare les performances des mémoires caches et SPMs en termes de consommation d'énergie et de surface mémoire occupée. Au delà du fait que ses résultats montrent un gain de surface et d'énergie dans le cas des mémoires SPMs, le modèle qu'il utilise calcule la consommation d'énergie par accès pour la mémoire cache ainsi que pour la mémoire SPM et cela pour différentes tailles. L'énergie totale consommée en mémoire SPM est égale au nombre d'accès à la mémoire SPM multiplié par l'énergie consommée par accès à la SPM. De même pour la mémoire cache. Que se soit pour la mémoire cache ou pour la mémoire SPM, l'énergie par accès consommée par ces mémoires est égale à la somme des énergies consommées par leurs composants respectifs. Cette énergie par accès peut être obtenue grâce au simulateur CACTI [Wilton et Jouppi, 1996]. Il ne reste plus qu'à déterminer le nombre d'accès à la mémoire SPM ou à la mémoire cache qui dépend du *benchmark* et des données utilisés.

Référence	Mémoires considérées	Spécificités
[Banakar *et autre.*, 2002]	– cache – SPM	comparaison performances en énergie/surface
[Fradj *et autre*, 2005]	– cache – SPM – DRAM	comparaison consommation d'énergie de trois architectures mémoires : – DRAM + cache – DRAM + SPM – DRAM + cache + SPM
[Mamidipaka et Dutt, 2004]	– cache – SPM	eCACTI[14] : version augmentée de CACTI
[Shiue et Chakrabarti, 1999]	– cache	– taux de *miss* – nombre de cycles en énergie vs. allocation mémoire *off-chip* – *tiling* – associativité
[Wehmeyer *et autre*, 2004]	– cache – SPM – CPU	prise en compte des branchements mémoire dans la fonction à minimiser
[Wilton et Jouppi, 1996]	– cache – SPM	CACTI, un modèle analytique

TAB. 1 – Modèles d'estimation de la consommation d'énergie

D'autres articles prennent également en considération la mémoire principale (DRAM) dans leur modèle. Dans [Fradj *et autre*, 2005], les auteurs cherchent à exploiter l'espace mémoire *on-chip* efficacement afin de réduire la consommation d'énergie sans heurter la performance globale du système. Pour ce faire, ils comparent l'impact de trois architectures mémoires différentes sur la consommation d'énergie. Leurs résultats obtenus par simulation montrent un gain en énergie de 19% en utilisant une architecture mémoire composée d'une mémoire principale et d'une mémoire SPM par rapport à une architecture mémoire composée d'une mémoire principale et d'une mémoire cache. En considérant une architecture mémoire combinant mémoire cache et mémoire SPM avec la mémoire principale, les auteurs obtiennent un gain en énergie de 35% par rapport à la même architecture mémoire (une mémoire principale et une mémoire cache). Dans leur modèle de consommation d'énergie, les auteurs utilisent le même principe que [Banakar *et autre*, 2002] mais font la distinction entre énergie

statique et énergie dynamique et cela pour les trois types de mémoires. Dans le cas de la mémoire cache seulement, ils rajoutent l'énergie de transfert dans leur équation. L'énergie d'un accès à la mémoire cache ou à la mémoire SPM est obtenue grâce à l'outil CACTI. Pour la mémoire DRAM, les valeurs sont données par les spécifications du constructeur.

Notons que la consommation d'énergie du processeur est rajoutée au modèle d'estimation de la consommation d'énergie proposé par [Wehmeyer *et autre*, 2004]. Cette énergie est cependant omise dans les calculs car les auteurs considèrent qu'elle n'a pas d'influence sur les résultats.

Le tableau TAB 1 illustre en résumé les modèles d'estimation de la consommation d'énergie.

6. Conclusion

Dans ce chapitre on a fait une étude sur les contraintes des mémoires dans les systèmes embarquées. La conception de systèmes embarqués ou enfouis, appelés ''System on chip- SOC'' utilise une méthode conjointe dans laquelle la mise en place de la plate-forme matérielle et celle des parties logicielles sont soumises à des interactions très fortes. Il est donc extrêmement important de prendre en compte les optimisations possibles sur la mémoire et les communications dans les hiérarchies de mémoires afin de contrôler la taille et la consommation de ces systèmes. Une étude consacrée sur un type de mémoire très recommandé pour les systèmes on- chip, à savoir la SPM (Scrachpad Memory) ainsi que les techniques utilisées pour sa gestion sera l'objectif du prochain chapitre.

Chapitre II

Techniques de gestion de la mémoire SP M

Introduction

Les mémoires Scratch-Pad (SPMs) sont considérées comme étant efficaces dans la réduction de l'énergie consommée en mémoire, un problème sensible, particulièrement dans les systèmes embarqués. Cependant, la complexité du domaine et la grande variété de techniques de gestion des mémoires SPM compliquent le choix de l'algorithme adéquat.

1. Placement des données en mémoire

La première catégorie comprend les techniques de gestion SPM qui se focalisent sur le placement des données en mémoire en fonction du type de mémoire. Ces approches tentent de répondre à la question: quelles variables du programme devraient être allouées à quelle mémoire ou banc mémoire? Dans ces techniques, à cause de la taille réduite de la SRAM, les variables les moins utilisées sont d'abord allouées aux bancs mémoire lents, alors que les variables les plus fréquemment utilisées sont gardées en mémoire rapide le plus longtemps possible. Ces méthodes utilisent les données des profils d'exécution pour collecter des informations sur les fréquences d'accès (nombre de fois qu'une donnée est accédée) afin de placer les données fréquemment utilisées dans la mémoire rapide et les autres données dans la mémoire lente. Pour ce faire, la plupart des auteurs modélisent le problème comme étant une programmation linéaire d'entier 0/1 (ILP).

L'article [Avissar *et autre*, 2002] considère les variables globales et de pile et choisit entre SPM et cache alors que [Steinke *et autre*., 2002; Wehmeyer *et autre*, 2004] considèrent les variables globales, les fonctions et les blocs de base et choisissent entre les différents bancs SPM seulement. Le Tableau 1 résume les différentes caractéristiques des articles considérés dans cette section. Au lieu d'utiliser une seule grande SPM, les résultats par simulation obtenus par [Wehmeyer *et autre*, 2004] ont montrés qu'en utilisant une SPM partitionnée, des améliorations de la consommation d'énergie mémoire de plus de 22% peuvent être atteints.

Référence	Objets traités	Spécificités
[Avissar *et autre*, 2001] [Avissar *et autre*, 2002]	variables globales et variables de pile	– choix SPM/cache – pile distribuée
[Nguyen *et autre*, 2005]	variables globales, de pile et objets du code	– taille SPM pas connue à la compilation – allocation idem qu'en [Avissar *et autre*, 2002] – aucune mesure pour la consommation d'énergie
[Panda *et autre*, 1997]	Variables scalaires et tableaux	– allocation des données en mémoire en fonction des données les plus conflictuelles par rapport à la mémoire cache
[Steinke *et autre*, 2002b]	variables globales, fonctions et blocs de base	– que pour la SPM
[Wehmeyer *et autre*, 2004]	variables globales, fonctions et blocs de base	– que pour la SPM – prise en considération des branchements mémoire dans la fonction à maximiser

TAB.1 – Allocation des données selon le type de mémoire

Ces techniques se basent toutes sur la fréquence d'accès aux données. [Panda *et autre*, 1997] quant à lui, considère les tableaux et les variables scalaires et se focalise sur les données les plus conflictuelles dans la mémoire cache. Les auteurs se basent sur les données des profils d'exécution pour placer les données les plus conflictuelles en SRAM. Toutes ces approches nécessitent la connaissance de la taille de la SPM à la compilation mais [Nguyen *et autre*, 2005] présente une méthode dont l'exécutable produit est portable à travers les SPMs de n'importe quelle taille. La méthode consiste à découvrir d'abord la taille de la SPM, en faisant un appel système soit bas niveau, soit au niveau de l'OS s'il est disponible, ou en sondant les adresses en mémoire en utilisant un modèle de recherche binaire et en observant la latence pour trouver la rangée d'adresses appartenant à la SPM. Puis, l'algorithme d'allocation mémoire est le même que celui dans [Avissar *et autre*, 2002].

1.1 Localité des accès mémoire

Les techniques présentées dans cette section peuvent être considérées comme un raffinement de celles présentées en section présidente En effet, il serait intéressant de conjuguer le meilleur placement de données en respectant les types de mémoires et la localité des accès mémoire afin d'optimiser encore plus l'utilisation de l'énergie. Le principe de la localité consiste à diviser la mémoire SPM en bancs, à y disposer les données et à les allouer au même banc autant que possible, tout en mettant les autres bancs en mode de puissance bas. Il existe deux types de localités : la localité spatiale et la localité temporelle. Le Tableau 2 présente les principales différences de quelques uns des articles qui traitent de ce sujet.

Localité	Référence	Objets traités	Spécificités
Spatiale	[Athavale *et autre*, 2001]	tableaux en Java	tableaux multidimensionnels
	[Kandemir *et autre*, 2004]	tableaux et boucles	– SPM virtuelle – fonction de *Folding* – problème modélisé avec des matrices et des vecteurs
	[Kandemir *et autre*, 2005]	tableaux et boucles	[Kandemir *et autre*, 2004] mesures pour la consommation d'énergie
Temporelle	[Verma *et autre*, 2004]	variables globales (non)scalaires, locales non scalaires et non scalaires et	– gestion dynamique : les objets mémoire sont alloués/retirés de la SPM en fonction de leur cycle de vie

TAB.2 – Localité des accès mémoire

1.2 Localité spatiale

Certaines méthodes sont basées sur l'optimisation de la localité spatiale ; des accès SPM successifs utilisent le même banc SPM le plus longtemps possible. En effet, augmenter la localité spatiale d'un ensemble de bancs SPM augmente clairement la durée pendant laquelle les autres bancs SPM ne sont pas accédés, ce qui aide à amortir le coût de transition d'un banc en mode basse consommation et son retour en mode de

consommation normal. La méthode présentée dans [Athavale *et autre*, 2001] explore la consommation d'énergie des mécanismes d'allocations des tableaux en Java. En utilisant un ensemble de programmes test (*benchmarks*) dominés par des tableaux et une architecture mémoire partitionnée supportant des modes basse puissance, les auteurs étudient deux techniques d'optimisation des données : la modification de la disposition mémoire et l'entrelacement de tableaux. La modification de la disposition mémoire consiste à changer l'ordre de stockage des données à l'intérieur d'un tableau afin d'améliorer sa localité spatiale. L'entrelacement de tableaux regroupe ensemble dans le même module mémoire les éléments appartenant à différents tableaux multidimensionnels, augmentant ainsi l'intervalle d'inter-accès (temps entre deux références du même module) des modules non utilisés. Cela fournit une opportunité pour mettre les modules mémoire en mode basse puissance durant une période plus longue.

Les résultats expérimentaux obtenus par les auteurs montrent que la modification de la disposition mémoire et l'entrelacement de tableaux fournissent en moyenne 9,68% et 14,96% de gains en énergie, respectivement. La stratégie de compilation proposée dans [Kandemir *et autre*, 2004; Kandemir *et autre*, 2005] est également efficace dans la réduction du courant de fuite (*leakage energy*) des mémoires SPMs sur puce (*on-chip*). Cette méthode a l'avantage de considérer les tableaux et les boucles en général sans restrictions à un langage particulier. L'idée dans [Kandemir *et autre*, 2005] est de diviser la mémoire SPM en bancs tout en optimisant la disposition des données (guidé par le compilateur) et en faisant migrer des données afin de maximiser le ralentissement des bancs SPM, augmentant ainsi les chances de placer les bancs mémoires en mode basse puissance. Ce travail se focalise sur la réduction de la consommation d'énergie des SPMs *on-chip* sans heurter la performance globale du système.

1.3 Localité temporelle

D'autres méthodes se basent sur la localité temporelle ; les bancs SPM accédés récemment sont susceptibles de l'être encore dans un futur proche. [Verma *et autre*, 2004] présente une approche basée sur les profils qui, sur la base des cycles de vie des variables et des segments de code, remplit le contenu de la mémoire SPM. Ces variables et segments de code sont choisis de façon à minimiser le surcoût énergétique causé par les allers-retours des objets mémoire depuis la mémoire SPM vers la mémoire DRAM.

Cette technique calcule aussi les adresses mémoire appartenant à la SPM où les variables et les segments de code doivent être copiés. Ces adresses sont calculées de telle façon à ce qu'un grand nombre de variables et de segments de code partagent le même espace SPM.

2. Techniques considérant tous les objets mémoire

Les techniques mentionnées dans la section précédente ont l'inconvénient de ne pas prendre en considération tous les types d'objets. Dans la section courante, contrairement aux précédentes, je me focalise sur les travaux d'Udayakumaran et de Barua qui eux, considèrent tous les objets mémoire : tableaux, boucles, variables globales, de tas et de pile. Les travaux d'Udayakumaran et de Barua se basent sur ceux de [Kandemir *et autre*, 2001] et essaient de les améliorer. Ainsi, afin de mieux souligner les différences entre les deux approches, j'explique le principe de [Kandemir *et autre*, 2001] en premier. Le Tableau 3 résume cela. Les deux méthodes font aller et revenir les données entre DRAM et SPM sous contrôle du compilateur, mais deux améliorations sont apportées par [Udayakumaran et Barua, 2003] par rapport à [Kandemir *et autre*, 2001]. Tout d'abord, [Kandemir *et autre*, 2001] considère seulement les tableaux, avec en plus les trois restrictions suivantes :

- Les programmes doivent d'abord accéder aux tableaux des boucles les plus imbriquées.
- Les boucles doivent être bien structurées et ne doivent pas avoir d'instructions de contrôle comme *if-else*, *break* et *continue*.
- Les codes contenant ces instructions doivent être bien écrits, sans les optimisations faites à la main souvent trouvées dans de nombreux codes, car ces optimisations considèrent non seulement la boucle en question, mais aussi un contexte plus large.

Référence	Objets traités	Spécificités
[Kandemir *et autre*, 2001]	boucles et tableaux	– gestion dynamique – réordonner les instructions afin d'optimiser le nombre de transferts – considère chaque boucle imbriquée indépendamment
[Udayakumaran et Barua, 2003]	variables globales, de pile et tous les modèles d'accès à ces variables	– gestion dynamique – différences par rapport à [Kandemir *et autre*, 2001] : – objets traités plus généraux – considère le programme en entier – ne permet pas de diviser un tableau
[Udayakumaran *et autre*, 2006]	variables globales, de pile et tous les modèles d'accès à ces variables et objets du code	– même principe que [Udayakumaran et Barua, 2003] – mesures faites pour la consommation d'énergie sur plus de *benchmarks*
[Udayakumaran et Barua, 2006]	variables globales, de pile et tous les modèles d'accès à ces variables + tableaux	– même principe que [Udayakumaran et Barua, 2003] – aucune mesure pour la consommation d'énergie
[Dominguez *et autre*, 2005]	données du tas	– même principe que [Udayakumaran et Barua, 2003] – se base sur des déplacements fixes

TAB. 3 – Synthèse des travaux d'Udayakumaran et de Barua

En combinant ces trois restrictions, la méthode de Kandemir *et autre*, s'applique aux codes scientifiques et multimédia bien structurés. Cependant, comme l'ont souligné Udayakumaran et Barua, la plupart des programmes dans les systèmes embarqués ne remplissent pas ces conditions. [Udayakumaran et Barua, 2003] ont amélioré la généralité de la méthode et l'applique à toutes les variables globales et de pile ainsi qu'à tous les enchaînements des accès à ces variables. La méthode devient ainsi plus générale et est capable d'exploiter la localité de tous les codes, y compris ceux contenant des enchaînements d'accès irréguliers et des variables autres que les tableaux ainsi que du code avec pointeurs et flux de contrôle irrégulier.

La deuxième amélioration apportée par [Udayakumaran et Barua, 2003] est que [Kandemir *et autre*, 2001] considère chaque boucle imbriquée indépendamment alors que la méthode d'Udayakumaran et de Barua analyse le programme en entier et toutes

les structures de contrôle. Cela a plusieurs conséquences. L'une est que la méthode présentée par Kandemir *et autre*, est optimisée localement pour chaque boucle alors que la méthode d'Udayakumaran et de Barua est optimisée globalement pour le programme en entier. L'autre conséquence est qu'avec la méthode de Kandemir *et autre*, la mémoire SPM est disponible en entier pour chaque boucle imbriquée. En revanche, l'approche d'Udayakumaran et de Barua n'est pas dans l'obligation de rendre tout l'espace SPM disponible pour chaque boucle, elle peut ainsi choisir d'utiliser une partie de la SPM pour les données qui sont partagées entre des instructions de contrôle successives économisant de ce fait le temps et l'énergie de transfert à la DRAM.

Notons cependant que pour les tableaux, il est possible de diviser un tableau en plusieurs parties au lieu de le considérer en entier avec [Kandemir *et autre*, 2001], alors que cela est impossible avec [Udayakumaran et Barua, 2003]. Udayakumaran et Barua ont étendu leurs travaux dans d'autres articles. Par exemple, l'approche dans [Udayakumaran *et autre*, 2006] manipule aussi les objets du code et fournit quelques mesures pour la consommation d'énergie. Leurs résultats obtenus par simulation montrent que leur schéma réduit en moyenne le temps d'exécution de plus de 39,8% et la consommation d'énergie de plus de 31,3% pour leurs *benchmarks* (en fonction de la taille SRAM utilisée) par rapport à [Avissar *et autre*, 2002].

L'article [Dominguez *et autre*, 2005], souvent cité dans la littérature, est très intéressant car il est, à notre connaissance, le seul travail qui considère les données du tas et a une politique de gestion SPM à l'exécution qui permet des déplacements fixes (comme montré dans le Tableau 4). Cette politique consiste à découper le programme en régions puis à déterminer, pour chacune de ces régions, les sites d'allocation les plus souvent utilisés et à allouer leurs données en mémoire SPM. Entre une région r et une région r +i, si le site d'allocation n'est pas fréquemment utilisé, ses données peuvent être placées en DRAM afin de libérer leurs places en SPM en faveur des données d'un autre site utilisé fréquemment entre r et r + i. Ce placement est tel que les mêmes données d'un même site d'allocation sont toujours placés au même endroit en mémoire SPM et cela pour différentes régions. Cela permet d'éviter d'avoir à mettre à jour en r + i les pointeurs utilisés auparavant pour les données du même site en région r. Les résultats obtenus par simulation dans cet article montrent que cette méthode réduit la moyenne du temps d'exécution de 34,6% et la moyenne de la consommation de puissance de 39,9% pour la même taille de la SPM fixée à 5% de la taille totale des

données, quand les auteurs la comparent à une technique où toutes les variables de tas sont placées en DRAM et seulement les données globales et de pile sont placées en SPM.

Enfin, [Udayakumaran et Barua, 2006] étend le travail effectué en explorant l'allocation SPM pour les tableaux. Ce dernier article modifie l'algorithme déjà proposé en ajoutant un code pour identifier les variables partielles (colonne, ligne, collection d'éléments d'un tableau accédée par boucle imbriquée), en utilisant une passe d'analyse affine.

Le but de cette passe est de permettre l'allocation de parties d'un tableau, par exemple lorsque le tableau en entier ne peut pas être alloué en SPM. Cependant, cet article présente des résultats pour le temps d'exécution seulement, la consommation d'énergie n'y est pas abordée.

3. Nouvelle classification des politiques de gestion de la SPM

Dans cette section, on porte un regard neuf sur les techniques de gestion de la mémoire SPM. Le premier critère traite de façon dont l'information sur les fréquences des accès est recueillie et le second critère fait référence à l'algorithme de gestion de la mémoire SPM utilisé à l'exécution. Pour le critère Recueil d'informations, Compilation veut dire que l'information est recueillie en analysant le code à la compilation, alors que *Profile* indique que le recueil d'informations se fait en utilisant des profils d'exécution (traces). L'algorithme de gestion SPM *à* l'exécution peut être statique ce qui veut dire que les données peuvent être écrasées mais pas déplacées. Avec des déplacements, les données existantes en mémoire SPM peuvent être expulsées pour libérer la place aux données entrantes. De cette manière, les données ne sont jamais perdues. Le déplacement fixe place toujours les données au même endroit dans la mémoire SPM ou dans la mémoire DRAM. En revanche, avec le déplacement libre l'allocation SPM peut être adaptée dynamiquement à l'exécution en plaçant les données les plus fréquemment utilisées à n'importe quelle place libre de la mémoire SPM ou de la mémoire DRAM. A noter qu'avec les déplacements libres, il risque d'y avoir des problèmes de fragmentation.

Diverses méthodes sont statiques et sont basées sur les informations de compilation seulement [Absaret Catthoor, 2005; Steinke *et autre*, 2002b; Athavale *et autre*, 2001]. Cependant, en utilisant les Profils d'exécution, il est possible d'obtenir une vue plus précise des enchaînements des accès aux données du programme car les profils contiennent les informations permettant de savoir quelles variables sont accédées

durant quelle partie du programme. Dans ce contexte, [Egger *et autre*, 2006; Avissar *et autre*, 2002, Kandemir *et autre*, 2001] ont fourni quelques résultats expérimentaux qui améliorent en général leurs précédents résultats obtenus avec l'approche statique. De plus, les algorithmes de gestion SPM basés sur les déplacements [Dominguez *et autre*, 2005; Verma *et autre*, 2004; Udayakumaran et Barua, 2003] sont plus efficaces que les approches statiques car ils optimisent l'utilisation de l'espace SPM et permettent de changer le contenu de la mémoire SPM à l'exécution. En d'autres termes, comme dans la mémoire cache, les données sont bougées entre les mémoires DRAM et SPM, mais sous contrôle du compilateur. Le surcoût d'énergie causé par un déplacement est compensé par le meilleur placement.

Comme on peut le constater dans le Tableau 4. Il existe différentes combinaisons de Recueil d'informations et d'Algorithme de gestion SPM *à* l'exécution.

Référence	Recueil d'informations	Algorithme de gestion SPM à l'exécution
[Absar et Catthoor, 2005]	Compilation	Statique
[Athavale *et autre*, 2001]	Compilation	Statique
[Avissar *et autre*, 2002]	Compilation + Profils	Statique
[Dominguez *et autre*, 2005]	Compilation	Déplacement fixe
[Egger *et autre*, 2006]	Compilation + Profils	Statique
[Hiser et Davidson, 2004]	Compilation + Profils	Statique
[Kandemir *et autre*, 2001]	Compilation + Profils	Statique
[Kandemir *et autre*, 2005]	Compilation + Profils	Statique
[Nguyen *et autre*, 2005]	Compilation + Profils	Statique
[Panda *et autre*, 1997c]	Compilation + Profils	Statique
[Poletti *et autre*, 2004]	Compilation	Statique
[Steinke *et autre*, 2002a]	Compilation	Statique
[Steinke *et autre*, 2002b]	Compilation	Statique
[Udayakumaran et Barua, 2003]	Compilation	Déplacement libre
[Udayakumaran *et autre*, 2006]	Compilation	Déplacement libre
[Udayakumaran et Barua, 2006]	Compilation	Déplacement libre
[Verma *et autre*, 2003]	Compilation	Statique
[Verma *et autre*, 2004]	Compilation	Déplacement libre
[Wehmeyer *et autre*, 2004]	Compilation + Profils	Statique

Tab.4- Classification des politiques de gestion de la SPM

Cette étude montre qu'il n'y a aucune méthode qui fusionne la compilation et les profils d'exécution avec un algorithme de gestion SPM basé sur les déplacements.

4. Conclusion

Ce chapitre présente une étude ainsi qu'une classification de technique de gestion de la mémoire SPM. Afin de gérer l'espace de la mémoire SPM, certaines approches tentent de répondre à la question : quelles données allouer à quel type de banc mémoire SPM.

Dans le chapitre suivant on décrira la solution de [l'Avissar et Barua, 2001]. Cette solution et une gestion automatique de compilateur pour assigner des données de programme parmi les unités hétérogènes de mémoires.

Chapitre III

La solution d'Avissar et Barua

Introduction

Dans ce chapitre, nous allons présenter en détail la solution proposée par O. AVISSAR et R. BARUA [Avissar et Bruna, 2001] puisque celle-ci constitue la principale source de notre étude.

Vu la taille réduite de la mémoire SPM, les auteurs tentent à répondre à la question: quelles variables du programme devraient être placées dans quelle mémoire ou banc mémoire ? La méthode présentée dans l'article en question, utilise les données des profils d'exécution pour collecter des informations sur les fréquences d'accès afin de placer les données les plus fréquemment utilisées dans la mémoire rapide, à savoir la SPM, et les autres données dans la mémoire plus lente, à savoir la DRAM. Pour ce faire, les auteurs modélisent le problème dans le cadre de la programmation linéaire sur les entiers 0 et 1.

1. Motivation et approche

La méthode présentée est motivée par un besoin d'améliorer la qualité du code compilé. Le code généré par les compilateurs d'aujourd'hui reste toujours moins efficace comparativement à son équivalent généré à partir de programmes codés en assembleur [Bhattacharyya et autres, 2000], [Paulin et autres, 1997].Cela a d'ailleurs comme conséquence, le fait que les systèmes dont la performance du noyau est critique, sont directement écrits en assembleur. Malheureusement, la programmation en assembleur présente des inconvénients bien connus, à savoir qu'elle est plus pénible, plus cher, sujette à des erreurs et non portable. Une conséquence directe sur l'industrie, est que cela demande plus de temps pour la mise sur le marché de nouveaux produits.

L'un des principaux problèmes auxquels est confrontée la réalisation de compilateurs pour la production de code efficace, reste la présence simultanée dans un système d'une multitude d'unités mémoires, souvent hétérogènes, dont chacune se voit assigner une certaine portion de l'espace d'adressage. Plusieurs processeurs embarqués, parmi lesquels on peut citer Motorola MCore, Texas Instruments TMS370Cx et Motorola 68HC12, ont de telles unités mémoire à savoir, une scratch-pad SRAM et une DRAM internes ainsi qu'une DRAM et une ROM externes. Ces différentes unités ne forment pas une hiérarchie mémoire unifiée. Les mémoires caches sont écartées des systèmes embarqués pour des raisons de contraintes temps-réel, de coût et de

dissipation d'énergie. Notons que dans le cas des ordinateurs de bureau, les unités mémoires sont plutôt unifiées à travers l'utilisation de mémoires caches. Dans le cas des processeurs embarqués, l'accès direct aux unités mémoires nécessite une assignation par logiciel des données aux différentes unités.

Ce travail propose une méthode pour assigner automatiquement des données d'un programme aux unités de mémoire hétérogènes dans les processeurs embarqués sans caches. Une technique couramment utilisée, consiste à laisser l'assignation des données au programmeur. Une solution automatisée pour un genre différent de processeur embarqué, à savoir, un processeur pour lequel la mémoire externe dispose d'un cache matériel interne, a été proposée dans [Panda et autres 2000].Nous montrons, cependant, que les critères d'optimalité sont très différents dans le cas d'absence de mémoires caches.

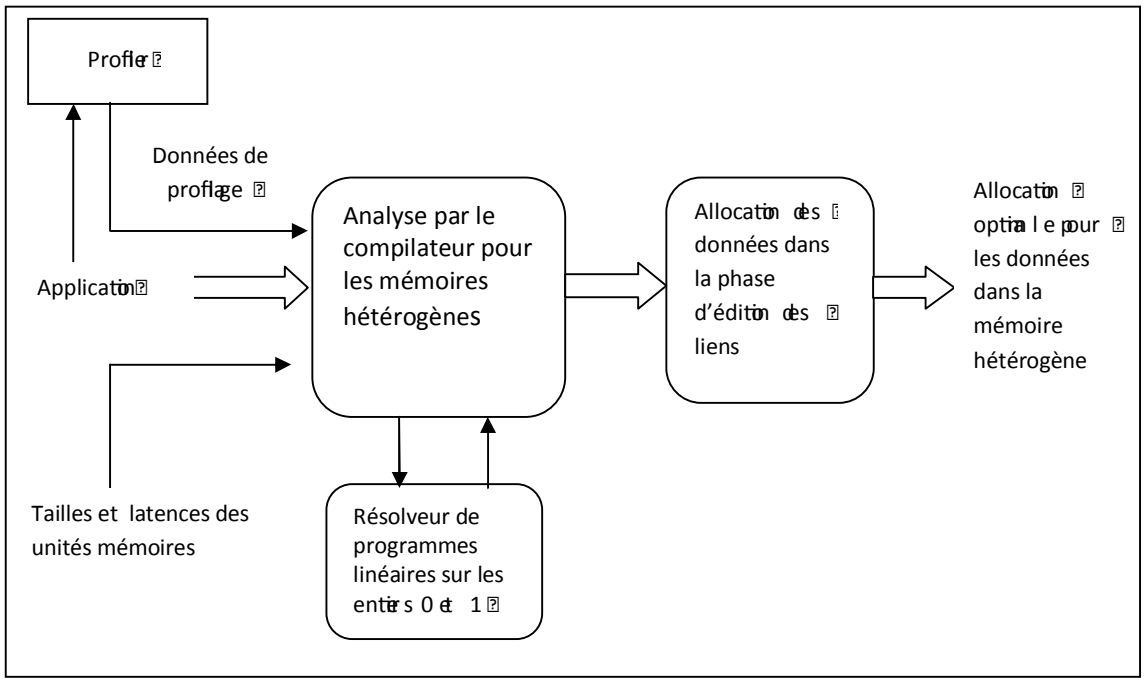


Figure 1 : Organigramme pour la gestion de mémoire hétérogène dans les systèmes embarqués.

La figure 1 décrit la méthode proposée par O. AVISSAR et R. BARUA [Avissar et autres, 2001] pour la gestion de la mémoire hétérogène sur les systèmes embarqués. A gauche, le programme d'application ainsi que les données du profilage sont soumis à une phase d'analyse du compilateur pour déduire une allocation statique optimale. Cette

phase reçoit également en entrée des informations sur les tailles et les temps de latences des différentes unités mémoires disponibles sur le système cible, et modélise le problème d'allocation mémoire pour les données globales et les données de pile comme un problème de programmation linéaire sur les entiers 0 et 1 que les auteurs proposent de résoudre en utilisant Matlab. Parmi les méthodes d'allocation statique de la mémoire aux données globales et aux données de pile, il est toujours possible de prouver que la technique que nous décrivons ici donne une solution optimale.

Le résultat de la phase d'analyse précédente, à savoir une spécification de l'allocation de la mémoire aux données, est ensuite fournie comme entrée au module de liaison qui ajoute des directives d'assemblage à son code de sortie pour implémenter l'allocation désirée.

Le code résultant améliore, non seulement la performance, mais réduit la consommation d'énergie puisqu'il est bien admis que les optimisations réduisant le temps d'exécution réduisent également la consommation d'énergie [Lee et autres, 1997].

2. Travaux connexes

Pour les processeurs embarqués avec des unités de mémoire hétérogènes et sans caches, il y a peu de travaux permettant l'assignation automatique des données aux différents bancs mémoires tout en augmentant les performances. L'approche habituelle consiste à laisser la tâche au programmeur. Cependant, les méthodes d'allocation intégrées dans les compilateurs sont préférables aux directives des programmeurs pour trois raisons :

- elles n'exigent pas d'effort au programmeur,
- elles sont portables sur différentes plateformes, et
- elles susceptibles de prendre de meilleures décisions, particulièrement pour les programmes longs et complexes.

À notre connaissance, les seules méthodes publiées permettant l'allocation de la mémoire aux données sur des systèmes ayant une mémoire interne et une autre externe disposant chacune de son propre espace d'adressage sont ceux présentées dans [Panda et autres, 2000], [Sjodin et autres, 1998] et[Sjodin et vonPlaten, 2001]. Cependant, la classe d'architecture visée par [Panda et autres,2000] est différente de celle considérée dans le travail que nous décrivons puisqu'elle cible des processeurs embarqués

disposant, en plus de la SPM, d'un cache matériel pour la mémoire externe. La présence de la mémoire cache change complètement l'objectif de la stratégie d'allocation. Au lieu de viser à réduire le nombre d'accès aux données dans la mémoire externe, il devient bien plus important de diminuer le nombre de défauts de cache. L'objectif de cette méthode n'est donc pas de maximiser le nombre d'accès à la SPM et elle n'est donc pas adaptée à notre modèle architectural.

La méthode proposée dans [Sjodin et autres, 1998] diffère également de celle que nous considérons dans plusieurs aspects. Même si cette méthode essaye également de maintenir des variables avec le nombre le plus élevé d'accès par octet dans la mémoire SRAM et assigne les variables moins critiques à une RAM plus lente et externe, il n'en demeure pas moins qu'à la différence de la méthode que nous décrivons, les auteurs de celle présentée dans [Sjodin et autres, 1998] ne considèrent que deux niveaux de mémoire (la SRAM interne et la RAM externe) et ne propose aucune façon de l'étendre pour traiter les variables de la pile. La formulation de la méthode que nous étudions [Avissar et all,2001] manipule automatiquement *N* niveaux de mémoires avec des temps de latences et des tailles variables. En outre, la formulation est étendue pour inclure les variables (locales) de pile. Une autre différence est que Sjodin et autres, utilisent un schéma de profilage statique pouvant être imprécis particulièrement pour les gros programmes. Notre allocation emploie toujours le profilage dynamique qui tient compte de chaque chargement (load) et enregistrement (store) dans tout le programme. Les travaux récents de Sjodin et vonPlaten [Sjodin et vonPlaten, 2001] utilisent une formulation linéaire comme celle utilisée dans la solution que nous décrivons. Les principales différences entre les deux méthodes se résument comme suit : tout d'abord, Sjodin et vonPlatenn utilisent pas de pile distribuée alors que les résultats de [Avissar et Bruna, 2001] montrent que cela permet d'obtenir des gains de performance de l'ordre de 44%. De plus, ils n'ont aucune analyse optimisée des variables de pile.

Comme décrit plus tôt, notre méthode utilise une allocation statique même si une stratégie dynamique reste possible. Dans une stratégie dynamique, des données peuvent être déplacées d'un emplacement à l'autre pendant l'exécution du programme. Une classe de stratégies dynamiques comporte les méthodes utilisant un cache logiciel[Moritz et autres, 2000], [Hallnor et Reinhardt, 2000]. Ces méthodes émulent le cache dans la mémoire rapide en utilisant le logiciel. A chaque accès d'un programme aux données, les étiquettes, les données et les bits de validation sont gérés par du code

inséré par le compilateur. Cela induit donc une augmentation de la taille du code généré et de son temps d'exécution. [Moritz et autres, 2000] cible le cache primaire(L1), alors que [Hallnor et Reinhardt,2000]cible plutôt le cache secondaire(L2). Pour cette raison la méthode proposée par [Moritz et autres,2000] est plus applicable pour le problème que nous considérons. Des défis significatifs devront être surmontés en concevant une technique par cache logiciel pour les processeurs embarqués. Il faudra entre autres fournir des garanties de temps réel lorsque le cache a un comportement imprévisible, et réduire la taille et le temps supplémentaires induits par la gestion du cache.

Heureusement, les données de profilage permettent à notre méthode statique d'incorporer l'information sur le temps d'exécution dans une certaine mesure.

3. Exemple

La figure 2 montre un exemple simple qui illustre comment un programmeur ou un compilateur pourrait prendre des décisions au sujet de l'allocation de la mémoire aux données. La figure 2(a) est le programme d'application pour lequel nous souhaitons allouer la mémoire aux données. Deux tableaux de type octet *A* [100] et *B* [1000] sont passés comme arguments à la procédure *foo* qui les manipule à travers les paramètres *x* [ ] et *y* []. Supposons que le programme soit compilé pour un processeur embarqué muni d'une unité mémoire rapide interne de type SRAM (scratch-pad) de 1 Ko et d'une unité mémoire externe, plus lente, de type DRAM de 8 Ko. On suppose également que la puce n'intègre pas de DRAM interne. Le problème que nous essayons de résoudre est posé comme suit: quelles unités mémoires allouer à quelles variables du programme? Il est clair que chacun des deux tableaux peut être placé dans la SRAM mais pas les deux simultanément. Pour simplifier l'illustration, supposons qu'il n'y ait aucun autre accès à *A* et *B*, bien qu'il n'y ait aucune telle condition dans notre méthode.

Pour le code de la figure 2(a), le compilateur doit choisir entre les deux allocations possibles illustrées par les figures 2(b) et (c). Dans la figure 2(b), A [100]est dans la SRAM alors que dans 2(c) c'est plutôt *B* [1000]qui occupe cette unité mémoire. Le choix entre les deux dépend des fréquences d'accès.

Deux cas illustrent le choix. Dans le premier cas, supposons qu'à l'exécution, la boucle *while* est exécutée réellement pour plus d'itérations que la boucle *for*; alors l'allocation de la figure 2(b) est préférée car elle fait plus d'accès à la SRAM plus rapide. Dans le deuxième cas, supposons que la boucle *for* s'exécute pour plus

d'itérations; dans ce cas, l'allocation de la figure 2(c) est meilleure puisqu'elle fait plus d'accès à la SRAM. Cependant, faire des évaluations des fréquences d'accès relatives est difficile puisque beaucoup de boucles ont des bornes inconnues à la compilation. Heureusement, les données du profilage donnent de bonnes évaluations que les méthodes statiques de prévision de fréquence; il suffit juste que l'ensemble des données choisi soit représentatif. Notre méthode d'allocation utilise les données de profilage pour trouver les fréquences d'accès relatives aux adresses mémoire.

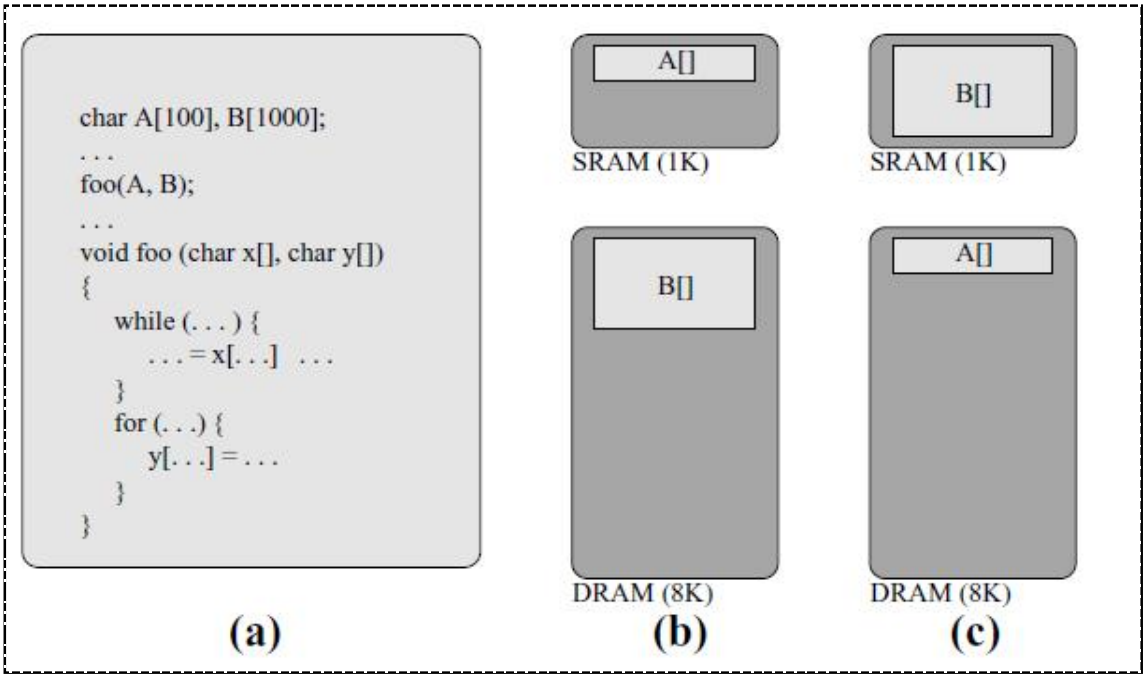


Figure 2 : Exemple montrant le choix dans l'allocation de mémoire hétérogène :(a) le code source de l'application;(b) *A* [ ] assigné à la SRAM;(c) *B* [ ] assigné à la SRAM.

L'exemple ci-dessus illustre que pour prendre de bonnes décisions d'allocation, le compilateur doit intégrer au moins trois technologies. D'abord, le problème général de l'allocation optimale de la mémoire, qui est NP-complèt, doit être résolu exactement ou approximativement en utilisant des heuristiques. La méthode que nous décrivons ici calcule une solution statique optimale dans le cadre de la programmation linéaire sur les nombres entiers. La programmation sur les entiers 0 et 1 fournit rapidement une solution même pour les gros programmes utilisant des milliers de variables [Appel et George 2001]. Heureusement, le nombre de variables dans notre formulation est proportionnel au nombre de variables dans le programme source original, qui n'est

habituellement pas plus de quelques milliers même pour les gros programmes. Des solutions rapides sont confirmées par les résultats des auteurs dans lesquels, dans tous les cas, la solution a été retournée en une minute. En second lieu, le compilateur doit rassembler des évaluations précises de fréquences en utilisant le profil de l'application, et les utiliser pour réaliser l'allocation. Troisièmement, pour rassembler des calculs de fréquences pour les variables, le profiler doit corréler les accès avec les variables qu'ils accèdent. Par exemple, sur le schéma 2(a), le profiler doit savoir que *x* [ ] est *A* [ ] et *y* [ ] est *B* [ ], ceci exige une analyse inter-procédurale de pointeurs, très couteuse en temps de calcul. De plus, l'analyse de pointeurs peut renvoyer des données ambiguës comme, par exemple, dans le cas où il y aurait plusieurs appels à la fonction *foo()* avec différents arguments. Pour cette raison, nous évitons l'analyse de pointeur en adoptant une approche différente à savoir, le contrôle d'adresse pendant le profilage où chaque adresse accédée est examinée dans une table des plages d'adresses pour les différentes variables. Le fait de retrouver la plage d'adresses indique que la variable accédée a été trouvée. L'approche basée sur le profil d'exécution donne des statistiques exactes, à la différence de l'information inexacte donnée par utilisant l'analyse de pointer.

4. Les formules pour les variables globales

Pour présenter la formulation de l'allocation mémoire pour les variables globales, on utilisera les symboles suivants:

U = Nombre d'unités mémoires hétérogènes;

T_{rj} = temps, en cycles, pour lire l'unité mémoire$j \in [1, U]$;

T_{wj} =temps, en cycles, pour écrire dans l'unité mémoire$j \in [1, U]$;

M_j =Taille, en octets, de l'unité mémoire $j \in [1, U]$;

G=Nombre de variables globales dans l'application ;

V_i= la i^{ème} variable globale, $i \in [1, G]$;

$N_r(v_i)$=Le nombre de fois que v_i est lue (obtenu à partir du profil d'exécution);

$N_w(v_i)$= Le nombre de fois que v_i est écrite (obtenu à partir du profil d'exécution) ;

$S(v_i)$= Taille, en octets, de la variable v_i.

Le problème d'optimisation est formulé comme un programme linéaire sur les nombres entiers 0 et 1. Pour chaque unité mémoire $j \in [1, U]$ et pour chaque variable globale v_j ($i \in [1, G]$), on définit la variable $I_j(v_i)$à valeurs entières 0 ou 1 comme suit:

$$I_j(v_i) = \begin{cases} 1 & \text{si la variable vi est allouée dans l'unité mémoire j} \\ 0 & \text{si non} \end{cases}$$

La fonction objective à minimiser est le temps d'accès total de tous les accès mémoires de l'application. Pour des architectures permettant au plus un accès mémoire par cycle, le temps total est :

$$\sum_{j=1}^{U} \sum_{i=1}^{G} I_j(v_i)[T_{rj} N_r(v_i) + T_{wj} N_w(v_i)] \qquad (1)$$

$T_{rj}N_r(v_i)$ est le temps pour tous les accès en lecture à la variable v_i si elle était allouée dans l'unité mémoire *j*. Un terme similaire est ajouté pour tous les accès en écriture. Une fois multiplié par la variable $I_j(v_i)$ de valeur 0 ou 1, on obtient le temps d'accès mémoire si v_i était effectivement allouée dans l'unité mémoire *j*, ou bien 0 dans le cas contraire. En sommant ce terme sur toutes les variables (le sigma interne), on obtient le temps d'accès total pour une unité mémoire donnée. Le sigma externe donne le temps d'accès total à toutes les unités mémoires.

Pour les machines qui permettent un accès mémoire par cycle, la formule en (1) est précise. La plupart des processeurs embarqués visés et certaines architectures VLIW (Very Long Instruction Word) permettent seulement un accès mémoire par cycle; par conséquent, la formule en (1) est précise pour la plupart des processeurs visées. Pour les architectures VLIW qui permettent plus d'un accès mémoire par cycle, la formule en (1) ne tient pas compte du chevauchement des latences des mémoires. Pour en tenir compte, la formule doit inclure le maximum des latences des différents accès mémoires dans le même cycle; malheureusement, la fonction objective ne reste pas linéaire, puisque la fonction maximum n'est pas linéaire et ce problème n'est pas pris en considération dans le travail que nous présentons.

Comme dans n'importe quel problème d'optimisation linéaire, un ensemble de contraintes est également défini. La première est une contrainte d'exclusion qui impose pour chaque variable *vi* d'être alloué dans une seule unité mémoire :

$$\sum_{j=1}^{U} I_j(v_i) = 1 \quad (\forall i \in [1, G]) \qquad (2)$$

Une autre contrainte est que la somme des tailles de toutes les variables allouées dans une unité mémoire ne doit pas dépasser la taille de cette unité:

$$\sum_{i=1}^{G} I_j(v_i) * S(v_i) \leq M_j \quad (\forall j \in [1, U]) \qquad (3)$$

La fonction objective en (1) combinée avec les contraintes (2) et (3) définit le problème d'optimisation dont la solution, donnant des valeurs aux variables $I_j(v_i)$, constitue l'allocation statique optimale.

5. Extension aux variables de la pile

La pile est utilisée pour les appels de fonctions. Lorsqu'une fonction est invoquée, le système construit un *bloc d'activation* pour cette fonction. Il s'agit d'une région de mémoire allouée en *bas* de la pile (adresses basses). Le pointeur de pile (qui cible les adresses basses) se déplace vers le bas. Le bloc d'activation est suffisamment grand pour contenir les arguments et les variables locales de la fonction invoquée, les registres qui doivent être sauvegardés pendant l'exécution de la fonction et certaines informations de contrôle. Lorsque l'exécution de la fonction est terminée, le pointeur de pile est restauré à sa valeur précédente (adresse plus haute) et le contrôle revient à la fonction appelante. Le plus important est de ne pas oublier qu'après le retour de la fonction, le contenu de son bloc d'activation n'est plus valide.

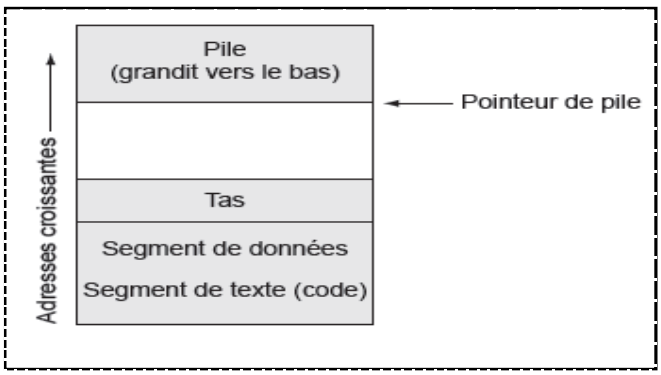


Figure 3. Organisation de la mémoire d'un programme C.

Pour obtenir de bonnes performances, les variables de la pile (paramètres des procédures, variables locales, …) doivent être distribuées parmi les différentes unités

mémoires hétérogènes. La distribution de la pile est un objectif complexe puisque la pile est une abstraction allouée de manière séquentielle.

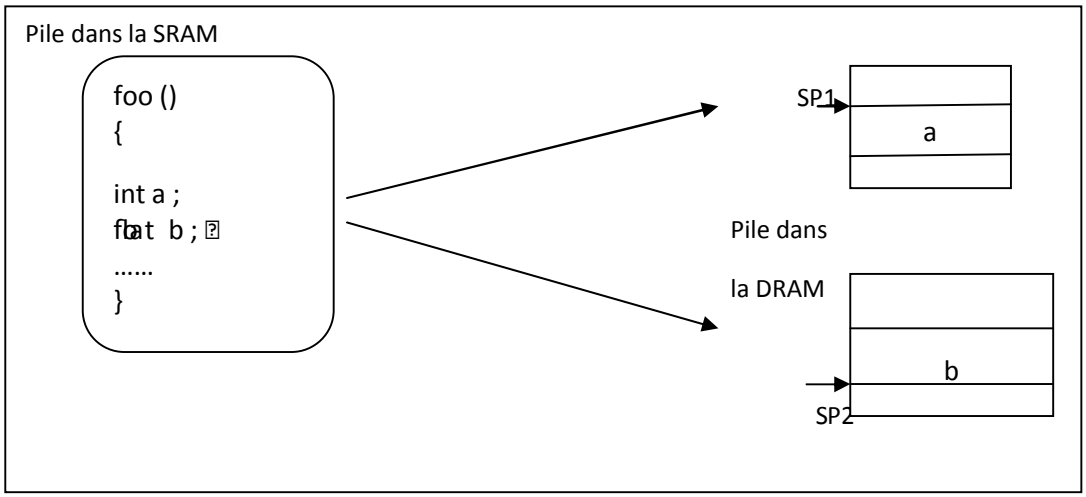


Figure 4. Exemple d'une pile fragmentée sur deux unités mémoires séparées. Les variables *a* et *b* sont placées, respectivement, dans la SRAM et dans la DRAM. Un appel à *foo()* exige des pointeurs de pile d'être incrémentés dans les deux mémoires.

La figure 4 présente une stratégie pour les piles distribuées appliquée à des unités mémoires hétérogènes. Les piles distribuées ont été proposées pour la première fois par l'un des auteurs de [Barua et autres, 2001] ; mais c'était pour un autre objectif. Avissar et Barua les adaptent pour la première fois aux unités mémoires hétérogènes.

L'approche est mieux expliquée par la figure 4. A gauche, on donne un exemple de fragment de code contenant une procédure *foo()* avec deux variables locales *a* et b. A droite, on montre comment la pile est distribuée dans deux unités de mémoire ; la variable *a* est assignée à la SRAM, et *b* à la DRAM. Ceci a comme conséquence la nécessité de disposer de deux pointeurs de pile, *SP1* en SRAM et *SP2*en DRAM. Les deux pointeurs de pile doivent être incrémentés à l'entrée de la procédure et décrémentés sur sa sortie. Pour faciliter l'utilisation, l'implémentation des piles distribuées est faite automatiquement par le compilateur; l'apparence séquentielle d'une pile non répartie est maintenue pour le programmeur.

Pour surmonter le coût supplémentaire induit par la gestion simultanée des deux pointeurs, deux solutions sont présentées ci-dessous dans les alternatives 1 et 2.

L'alternative 1 élimine le cout supplémentaire en faisant en sorte que toutes les variables d'une procédure soient assignées au même niveau de mémoire, de cette façon seul un pointeur de pile doit être mis à jour par procédure. La pile est toujours divisée : différents blocs d'activation peuvent être assignés à différentes unités mémoires, et on a plusieurs pointeurs de pile.

La deuxième solution est présentée dans l'alternative 2. Elle consiste à tolérer le surcoût et à distribuer chaque variable individuelle dans un bloc d'activation dans différents bancs mémoires tel que illustré à la figure 4.

La meilleure solution consiste à utiliser une alternative hybride des deux précédentes: tolérer le surcoût pour les procédures ayant une longue durée d'exécution. Pour les procédures lentes, identifiées par le profilage, l'impact de quelques instructions supplémentaires sera insignifiant; le surcoût sera donc toléré. Pour des procédures courtes, chaque bloc d'activation est assigné à une seule unité mémoire; et le surcoût sera donc éliminé.

6. Modification de la formulation globale

Pour voir comment modifier la formulation des variables globales pour des variables de pile, considérons que la différence fondamentale entre elles est que les variables de pile ont une durée de vie limitée. Les variables de pile pour des procédures non récursives peuvent être traitées comme des variables globales en leur allouant des emplacements mémoires permanents; cependant, il en résulterait une allocation mémoire non optimale pour variables de pile. La raison est que l'exécution avec des variables de pile peut être améliorée d'avantage en tirant profit de leurs durées de vie limitées. Les variables de pile sont assignées à l'entrée d'une procédure et libérées à sa sortie. Ainsi, la contrainte (3) n'est plus valide :la taille totale des variables assignées à un banc mémoire peut dépasser la taille de ce dernier puisque les variables ne sont pas toutes actives au même moment. Les variables dont les durées de vie ne se chevauchent pas peuvent partager le même espace mémoire.

Une manière d'incorporer des variables de pile à notre formulation consiste à traiter chaque variable de pile comme une variable globale tout en relaxant quelque peu la contrainte (3) sur la taille maximale. Au lieu d'exiger que toutes les variables soient simultanément en mémoire, le graphe d'appel du programme est analysé pour construire un nouvel ensemble des contraintes qui exigent que seulement les variables pouvant être simultanément actives, coexistent en mémoire. L'intuition est qu'une nouvelle

contrainte est introduite pour chaque chemin dans le graphe d'appel, allant de la fonction *main()* à chaque feuille du graphe .Les détails découlent directement de l'intuition et sont présentés ci-dessous pour les deux méthodes alternatives 1 et 2.

6.1 Alternative 1

Dans cette première alternative, pour chaque procédure, la pile est combinée dans la formulation, en une variable d'agrégation unique. Ceci assure que le bloc d'activation pour une procédure est assigné à une seule unité mémoire, permet une simplicité dans la formulation et l'implémentation et ne demande aucun coût supplémentaire pour la gestion de plusieurs pointeurs. La pile est distribuée puisque l'espace occupé pourrait être assignée à différentes unités mémoires. Pour décrire cette formulation, en plus des symboles utilisés pour les variables globales, on introduit les nouveaux symboles suivants:

F= Nombre de variables d'agrégats dans le programme d'application (nombre de fonctions) ;

f_i=i^{ème} fonction, $j \in [1, F]$;

$NP(f_i)$ = Nombre total des chemins uniques à la fonction f_i dans le graphe d'appel ;

$P_j(f_i)$ = j^{ème} unique chemin dans le graphe d'appel vers f_i $j \in [1, NP(fi)]$;

L=L'ensemble de toutes les feuilles dans le graphe d'appel.

En outre, $N_r(f_i)$, $N_w(f_i)$, $S(f_i)$, $I_j(f_i)$ désignant, respectivement, le nombre de lectures, le nombre d'écritures, la taille et la variable 0/1 du bloc d'activation pour la fonction f_i, sont définis de façon analogue à $N_r(v_i)$, $N_w(v_i)$, $S(v_i)$, $I_j(v_i)$.Les valeurs des variables $I_j(f_i)$ donnent l'allocation désirée.

La fonction objective pour la formulation étendue à la pile donne la durée totale de tous les accès mémoire aux variables globales et aux variables de pile. Elle est exprimée par:

$$\sum_{j=1}^{U}\sum_{i=1}^{G} I_j(v_i)[T_{rj}N_r(v_i) + T_{wj}N_w(v_i)] \quad (4)$$

$$+\sum_{j=1}^{U}\sum_{i=1}^{F} I_j(f_i)[T_{rj}N_r(f_i)+T_{wj}N_w(f_i)].$$

Le premier terme dans ce qui précède est la fonction objective originale (1) pour les variables globales ; il représente le temps total nécessaire pour accéder aux variables globales. Le deuxième terme est le temps total nécessaire pour accéder aux variables de pile. Concernant les contraintes, la contrainte d'exclusion pour des variables globales présentée en (2), reste nécessaire et inchangée. Une contrainte semblable est ajoutée pour les variables de pile :

$$\sum_{j=1}^{U} I_j(f_i)=1 \quad (\forall i \in [1,F]). \tag{5}$$

Comme mentionné précédemment, des changements sont nécessaires pour tenir compte du fait que les variables de pile peuvent avoir des durées de vie disjointes. Des changements substantiels sont apportés à la contrainte(3) sur la taille de la mémoire pour l'adapter aux durées de vie limitées des variables de pile. La nouvelle contrainte sur la taille de la mémoire est :

$$\begin{aligned}&\forall j \in [1,U], \forall f_l \in \mathcal{L}, \ \forall t \in [1, NP(f_l)],:\\ &\sum_{i=1}^{G} I_j(v_i)S(v_i)+\sum_{\forall f_p \in P_t(f_l)} I_j(f_p)S(f_p) \le M_j.\end{aligned} \tag{6}$$

La première ligne de la formule ci-dessus stipule que la contrainte de la seconde ligne porte sur toutes les combinaisons de bancs mémoire(j), de feuilles(f_l) dans le graphe d'appel, et de chemins à cette feuille(t). La contrainte dans la deuxième ligne stipule que les variables globales et toutes les variables de pile dans un chemin donné à une feuille donnée, doivent pouvoir être placées dans la mémoire. Le premier terme représente la taille des variables globales alors que le deuxième désigne la taille des variables de pile pour chaque chemin du graphe d'appel vers une fonction représentée par une feuille. La pile a la taille maximale quand une feuille du graphe d'appel est atteinte; par conséquent, en assurant que tous les chemins vers toutes les feuilles puissent tenir en mémoire, on garantit qu'à tout moment, l'allocation pour le programme dans son ensemble puisse tenir en mémoire.

L'ensemble de contraintes en (6) est considérable puisqu'il porte sur toutes les valeurs de j, *fl, et t.* Cependant, on ne s'attend pas à ce que cela affecte défavorablement le temps d'exécution de la résolution du programme linéaire puisque ce temps dépend

plus du nombre de variables que du nombre de contraintes. En fait, l'augmentation du nombre de contraintes peut diminuer le temps d'exécution en diminuant la plage des solutions faisables.

6.2 Alternative 2

Dans cette alternative, on permet à des variables de pile de la même procédure d'être assignées à différentes unités mémoires. La formulation est modifiée comme suit. Les variables de pile sont traitées juste comme des variables globales dans la formulation ;ce qui mène à une fonction objective semblable à (1). La contrainte d'exclusion est semblable à (2). La contrainte de taille mémoire est semblable à (6) avec la seconde fonction $\sum$ convertie en $\sum\sum$; la sommation externe reste la même que la deuxième sommation en (6) et celle interne réalise la sommation à travers toutes les différentes variables de la procédure f_p.

7. Conclusion

Dans ce chapitre, nous avons présenté l'une des méthodes les plus élaborées d'allocation mémoire pour les processeurs embarqués non démunis de caches matériels. En plus de permettre une allocation statique optimale pour les variables globales, cette méthode donne également deux alternatives pour l'allocation de la mémoire pour les variables de pile.

Notre implémentation personnelle, limitée au traitement des variables globales, de cette méthode dans le compilateur du domaine publique GCC réglé à la cible du compilateur trimaran pour les systèmes embarqués, ainsi que l'analyse des résultats, feront l'objet des deux prochains chapitres.

Chapitre IV

Conception

Objectif

Avant toute réalisation d'une application informatique, il convient de suivre une démarche méthodologique et rigoureuse pour planifier et concevoir l'application, en mettant en évidence tous les objectifs tracés pour la bonne élaboration du projet souhaité.

L'objectif principal de notre travail en l'étude de l'optimisation de la mémoire pour les systèmes embarqués sans cache matériel. et d'implémenter la méthode d'Oren Avissar présenté précédemment. Et pour le faire on exécute d'abords l'application par le profilage pour récupérer les information sur les fréquences d'accès afin de placer les variables globale fréquemment utilisées dans la mémoire rapide SPM et les autres données dans la mémoire lente DRA M.

La mise en application de la méthode implémentée s'effectue en deux étapes :

La 1^{ére} étape: consiste à exécuter l'application par le profiler et collecter les informations sur les fréquences d'accès a chaque variable.

La 2^{éme} étape: consiste à placer les variables dans la mémoire SPM selon le nombre d'accès pour réduire la consommation d'énergie.

1. Analyse et conception

Les étapes d'évaluations de notre travail sont présentées ci-dessous :

1.1 Hiérarchie de la mémoire

La formulation de la méthode que nous étudions [Avissar et Barua, 2001] manipule automatiquement *N* niveaux de mémoires hétérogènes.

Dans notre architecture a discret, on utilise un processeur embarqué muni deux unité s mémoire hétérogènes, une rapide et interne de type SRAM (scratch-pad) et l'autre externe de type DRAM, avec des temps de latences et des tailles différentes.

1.2 Le processus de profilage

Le *profiler* est un programme capable d'analyser un exécutable. Le résultat de l'analyse est appelé un *profil*.

Le profiler exécute les taches suivantes:

- Analyse l'exécution de programme.
- rassembler des calculs de fréquences pour les variables
- corréler les accès avec les variables qu'ils accèdent par le contrôle d'adresse où chaque adresse accédée est examinée dans une table des plages d'adresses pour les différentes variables.

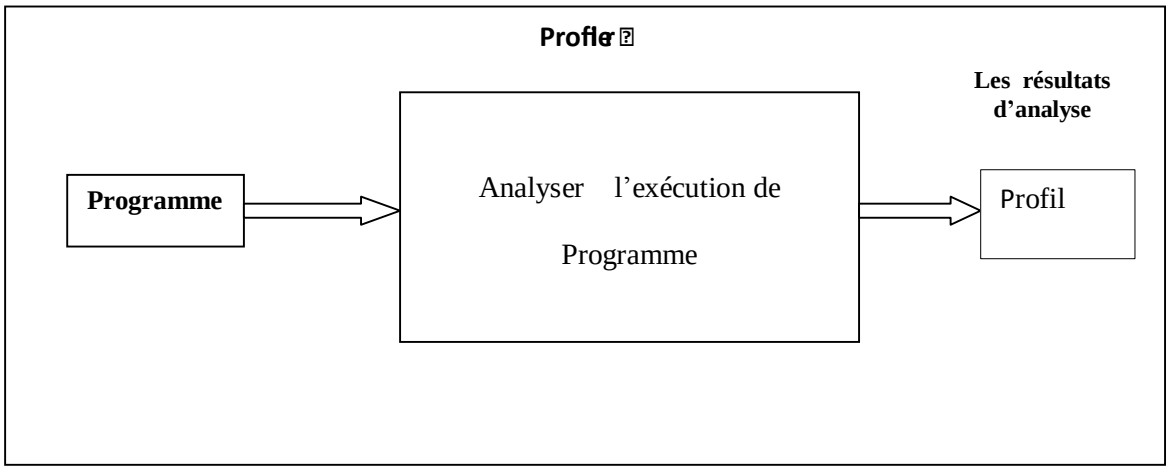


Figure 1 : le processus profilage.

1.3 Le processus de compilation

A la compilation, le compilateur exécute les taches suivantes :

- rassembler des évaluations précises de fréquences en utilisant le profil de l'application, et les utiliser pour réaliser l'allocation.
- Récupérer la taille et latence des unités mémoires hétérogènes.
- Trier la table du nombre d'accès a une variable par ordre ascendant.
- Insérer la variable dans la mémoire rapide SPM.
- Tester si la mémoire SPM est pleine.
- Si la mémoire SPM est saturée, insérer le reste dans la mémoire lente DRAM.

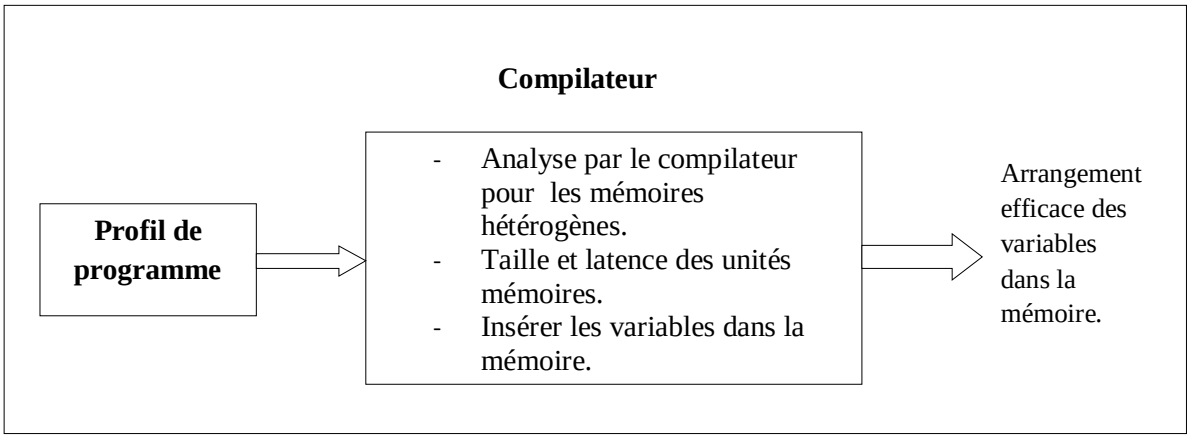


Figure 2 : le processus de compilation

2. Structure de la table des variables

Le compilateur utilise une table pour garder trace des informations concernant les variables. Chaque fois qu'une variable est rencontrée dans le programme, une recherche s'effectue.

Le mécanisme de la table créée doit nous permettre de créer de nouvelles entrées et de retrouver des sorties existantes de façon efficace, la structure la plus utilisée est la table de hachage.

La table de hachage comporte les noms et le nombre de cycle des variables qui sont associés. La structure est constituée de :

- Une table d'adressage dispersé consistant en un tableau de n pointeurs vers des entrées dans la table.
- Les entrées sont organisées en n listes chainées.

Pour déterminer si une entrée existe pour la variable C dans la table, nous appliquant à C une fonction de hachage hash telle que hash(C) retourne un entier entre 0 et n-1, puis on cherche dans la liste correspondante hash(C), si la chaîne n'appartienne pas on l'insère on créant un enregistrement en tête de la liste numérotée par hash(C), cet enregistrement comporte le nom de la variable, le nombre d'accès à la variable et un pointeur vers le prochain enregistrement.

3. Algorithme d'insertion des variables globale dans la mémoire

Cet algorithme est une implémentation de la méthode statique d'Oren Avissar et Rajeev Barua. Elle montre comment procéder pour insérer les variables globales d'un programme dans une mémoire SPM plus rapide et plus petite.

tant que (SPM non pleine)

pour i=1 jusqu'à n faire

lire ('tab[i]') ;

écrire ('tab[i] dans la mémoire SPM') ;

Fin

3. Conclusion

A la fin de notre conception, le problème a été analysé en profondeur et nous avons défini une conception appropriée à l'architecture de notre application. Nous pouvons alors entreprendre l'étape d'implémentation qui est l'objet d'étude du prochain chapitre.

.

Chapitre V

Implémentation

Introduction

Après avoir explicité dans le chapitre précédent la conception de notre application, nous allons présenter dans ce chapitre son implémentation dans le compilateur trimaran.

Tout au long de ce chapitre, nous allons, tout d'abord, commencer par la description des langages et les outils de programmations, l'environnement de développement et l'évaluation de notre optimisation, puis nous nous focaliserons sur la simulation des résultats.

1. Outils et langages

Pour implémenter la méthode, nous avons utilise le système d'exploitation Ubuntu 9.10, et les langages de programmation C et C++.

Ø *Gprof*

Le *profilage* d'un programme permet d'identifier les endroits où celui ci passe le plus de temps, mais également quelles sont les fonctions qui sont exécutées et combien de fois. Un *profiler* est un programme capable d'analyser un exécutable. Le résultat de l'analyse est appelé un *profile*. Le profiler standard dans le monde *GNU* est le programme *gprof*. Il fait partie du *package binutils* du projet *GNU* et est capable d'analyser des programmes écrits en *C*, *C++*, *Pascal* ou *Fortran77*.

Ø Compilation d'un programme pour le *profilage*

On compile un programme avec l'option *pg* qui permet d'altérer à la fois le comportement du compilateur et celui de l'éditeur de liens.

Exemple : **gcc prog.c oprog -pg**

2. Présentation de trimaran

Trimaran est un environnement intégré de compilation et de simulation utilisé dans la recherche en architecture des ordinateurs pour l'optimisation de code. Trimaran est fortement paramétrable, et peut viser un éventail d'architectures incluant les processeurs embarqués, les processeurs VLIW (Very Long Instruction Wold), et les architectures multi-cluster (multi-clustered architectures). Trimaran facilite l'exploration de model d'architecture désigné par la machine de description, et approprié bien à la synthèse automatique des architectures programmables pour les applications. Il tient compte de la personnalisation de tous les aspects d'une architecture, tel que : le chemin

d'accès, l'ensemble d'opérations, l'interconnexion, et les instructions mémoire entrée/sortie.

La nature modulaire du compilateur trimaran et la hiérarchie de représentation intermédiaire des programmes, rend l'insertion de nouveaux modules de compilation et d'optimisation facile. Trimaran est déjà mis à jour avec un grand nombre de modules existants en compilation. Il constitue un puissant outil pour de nouvelles recherches ainsi que pour l'enseignement. L'Utilisation de L'interface Graphique de Trimaran (GUI) fait la configuration et utilisation du système étonnamment facile.

De plus, trimaran exploite :

- Algorithmes d'ordonnancement et les mécanismes de pipelining.
- parallélisme de niveau hiérarchique de données en utilisant les instructions SIMD (Single Instruction Multiple Data).
- Diverse heuristique d'attribution de registre.

Bien qu'il y ait plusieurs infrastructures de compilation disponibles à la communauté des chercheurs, Trimaran est unique parce qu'il intègre des modules de compilation et de la recherche d'architecture. Trimaran est employé pour concevoir des applications, et des nouvelles optimisations, aussi bien que l'évaluation de la diverse innovation architecturale. Trimaran est largement répandu pour l'enseignement à travers plusieurs universités dans le monde entier.

3. Organisation De Trimaran

Comme présenté sur le schéma 1, Trimaran est composé de trois composants : le compilateur OpenIMPACT, le compilateur Elcor, et le simulateur. Trimaran emploie OpenIMPACT pour compiler le code source original et génère une représentation intermédiaire appelée Lcode. Le Lcode produit est optimisé pour ILP optimizations (optimisation de déroulement de boucles, le parallélisme), mais pas pour une machine spécifique. Ce code est alors passé au compilateur d'Elcor, avec une description de machine (MDES) qui décrit la machine cible. Elcor compile le code pour la machine cible, produisant une autre représentation intermédiaire (IR) appelée rebel. Le simulateur de Trimaran connu sous le nom de Simu exécute le code rebel et recueille des statistiques d'exécution.

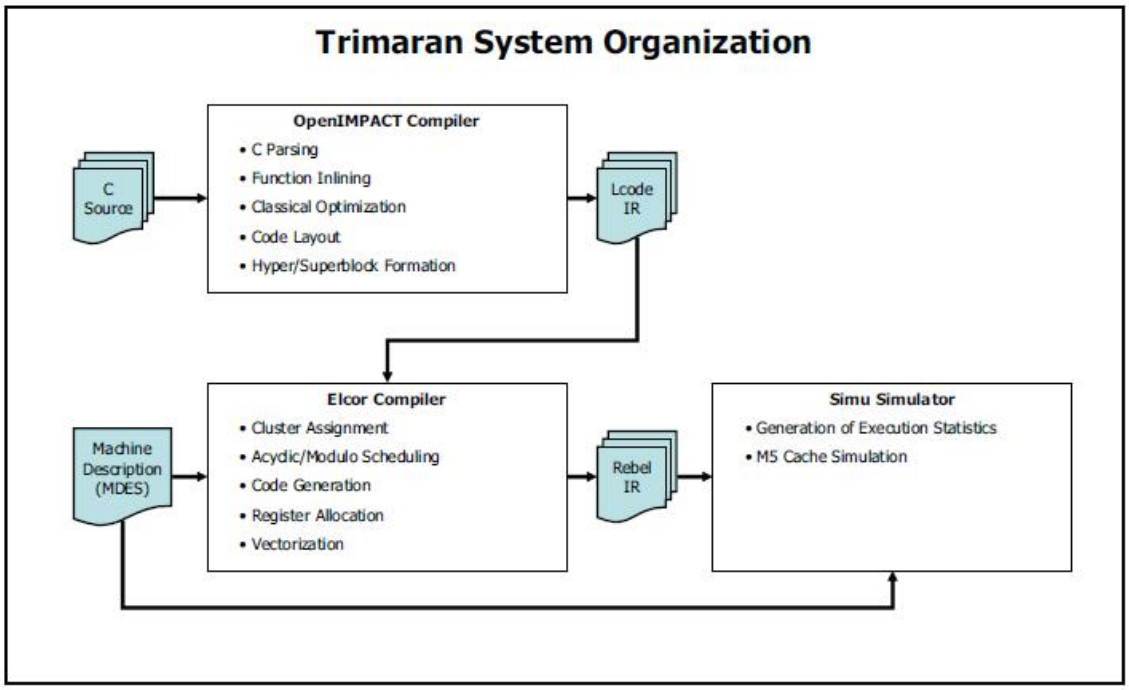


Figure1. Vue d'ensemble des étapes de compilation dans Trimaran

3.1 OpenIMPACT

Le compilateur d'OpenIMPACT est maintenu par le groupe d'IMPACT à l'université d'Illinois. Dans Trimaran, une version légèrement modifiée d'OpenIMPACT est utilisée pour compiler le code source original. Les passages principaux d'OpenIMPACT sont détaillés dans la figure 2.

3.2 Elcor

Le compilateur d'Elcor est un compilateur de VLIW (Very Large Instruction Wold) qui prend le Lcode de Openimpact non lié à un type de machine, le compile et l'optimise pour une machine spécifique décrite dans une description donnée de la machine (MDES). Les passages principaux d'Elcor sont détaillés dans la figure 2.

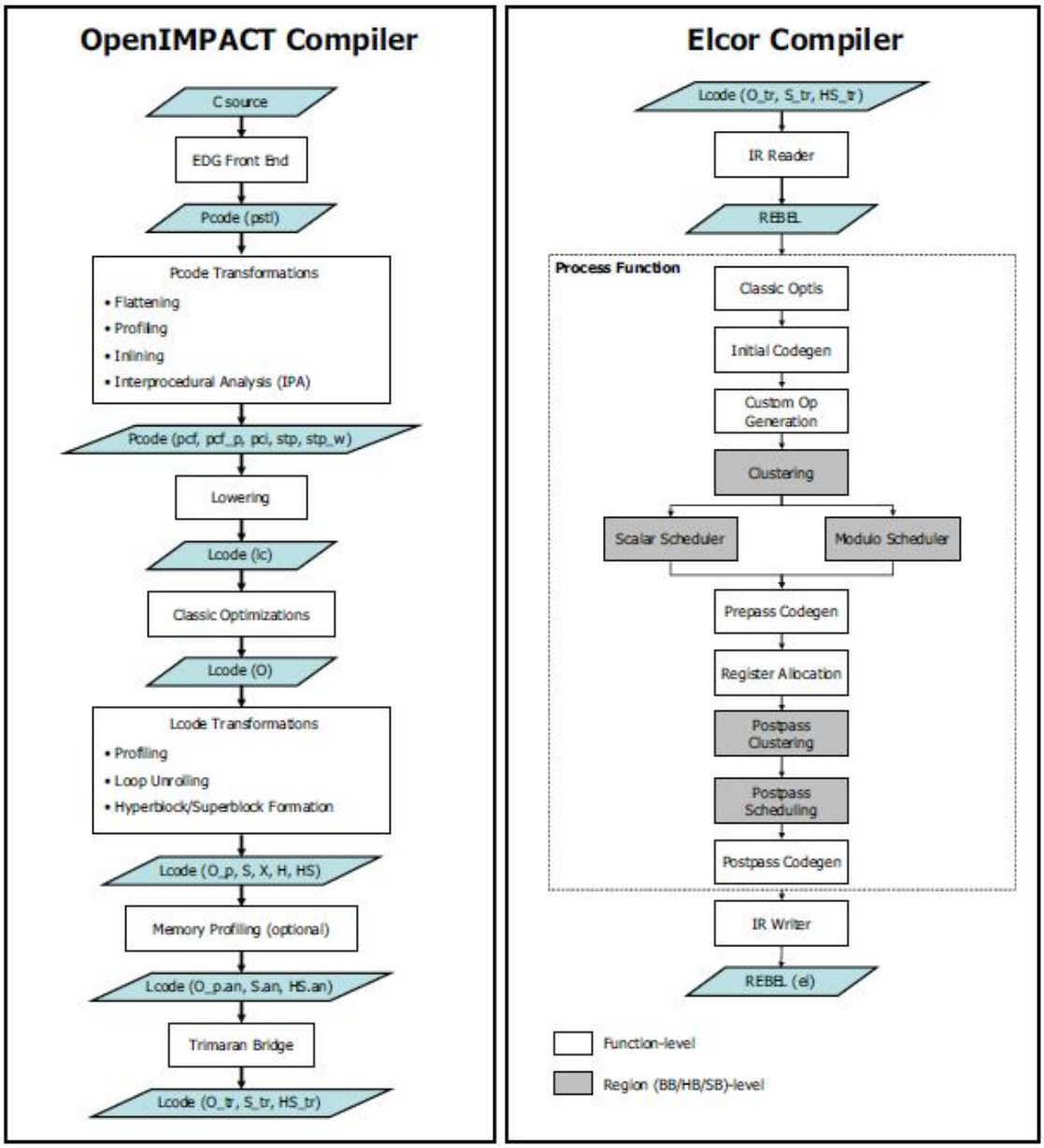


Figure 2 : Les étapes de compilation dans OpenIMPACT et Elcor.

3.3 Le simulateur

Le simulateur de Trimaran (Simu) supporte l'architecture Hpl-PD. Il prend la représentation intermédiaire (rebel) d'elcor comme entrée et crée un programme C qui fonctionne sur la machine cible, et émule comment le programme s'exécuterait sur l'architecture HPL_PD. Le simulateur produit des statistiques multiples décrivant l'exécution du programme.

Le simulateur a un composant statique et un composant dynamique. Le composant statique s'appelle Codegen, et la partie dynamique est Emulib. Le premier produit un code de bas niveau en C semblable à la représentation intermédiaire de rebel, Le code est alors compilé en utilisant le compilateur C lié à la bibliothèque d'émulation (Emulib) pour simuler l'application. Ceci est illustré sur la figure 3. Le langage C est employé car il est indépendant de la plateforme, de sorte que la simulation puisse fonctionner sur n'importe quelle plateforme, sans modification.

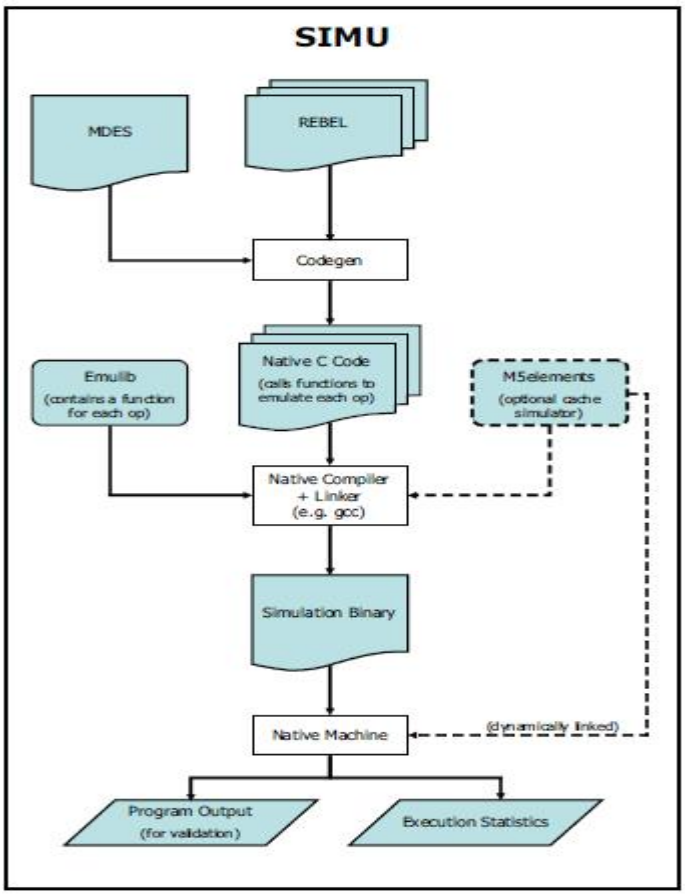


Figure 3. L'organigramme de simulateur

3.3.1 Codegen

Pour chaque fichier rebel d'entrée, Codegen crée quatre fichiers de sortie:

Un fichier avec l'extension **.c** contenant une série de routines (stub).
Un fichier avec l'extension **.inc** contenant les déclarations de variables externes, des constantes, et la structure de code source original C. En outre ce fichier est utilisé par

la bibliothèque Emulib pour le cheminement des statistiques d'exécution et l'information de profilage.

Un fichier avec l'extension **.tbls** contenant la collection des tables d'émulation. Une table d'émulation est maintenue par une procédure d'analyse. Chaque tableau d'émulation présente un ordre des instructions et leurs opérandes.

Un fichier appelé **benchmark data init.simu.c** qui initialise les structures de données globales du programme.

3.3.2 Emulib

La bibliothèque d'émulation fournit un interprète et un ensemble de routines d'émulation pour la machine virtuelle HPL-PD. La bibliothèque supporte la prédiction, le pipelining et l'ordonnancement.

L'émulation établit une multitude de statistiques pour résumer l'exécution du programme, Elle fournit également une interface pour simuler la hiérarchie mémoire. Le simulateur facultatif M5elements permet de modéliser le nombre de cycle précis du système de mémoire.

4. Description de machine MDES

Description de machine (MDES) dans TRIMARAN permet à l'utilisateur de développer une machine pour la famille HPL-PD, dans un langage de haut niveau, qui est traduit en représentation de bas niveau pour l'usage efficace par le compilateur. Le niveau élevé de langage permet les spécifications des contraintes d'exécution détaillées dans une façon compréhensive et maintenable. La représentation de bas niveau est conçue pour permettre au compilateur de vérifier les contraintes d'exécution avec l'efficacité élevée d'espace et de temps. Nous décrirons également l'interface du compilateur aux modules de description de machine MDES et, brièvement, on donne la structure interne au compilateur correspondant aux descriptions de machine.

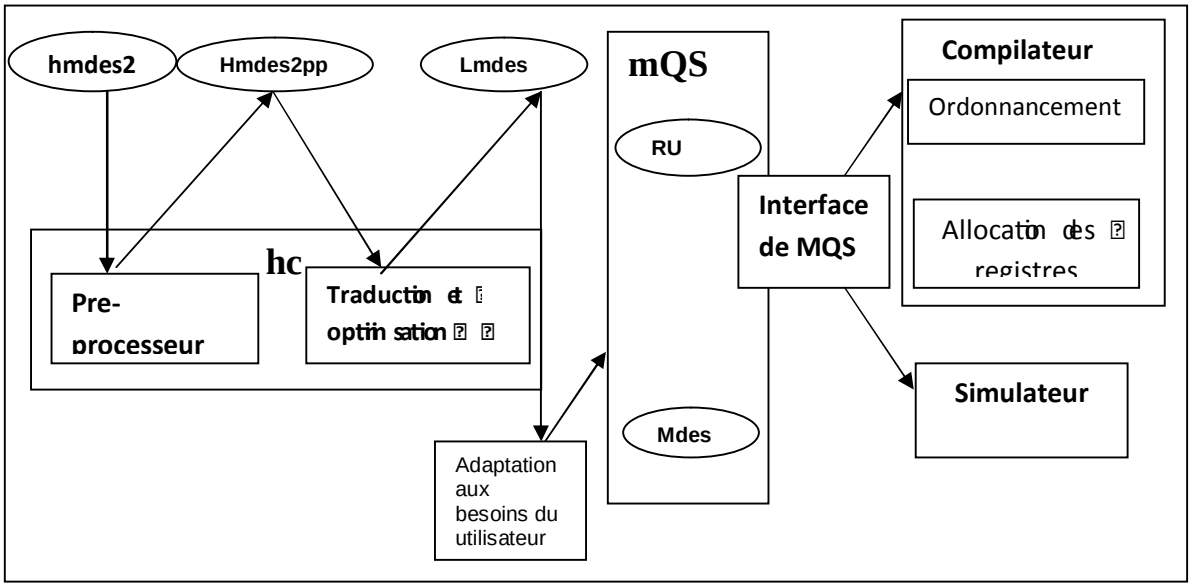


Figure 4. MDES dans trimaran

L'infrastructure de MDES dans TRIMARAN est montrée sur la figure 4. La cible (processeur Hpl-pd) est décrite par un langage appelée MD (langage de description de base de données relationnelle). Bien que le langage de MD tienne compte d'une variété de représentations, la machine de descriptions Hpl-pd est décrite par une de ces représentation de format particulier appelé HMDES ((High-level Machine Description). Après la compilation du niveau élevé, une spécification de bas niveau est présentée dans (*P.hmdes).* Les spécifications textuelles de LMDES établies sont chargées dans le compilateur par l'interface mQS.

4.1 Les répertoires relatifs à MDES dans TRIMARAN

L'organisation de description de machine MDES dans le compilateur trimaran est représenté comme suit :

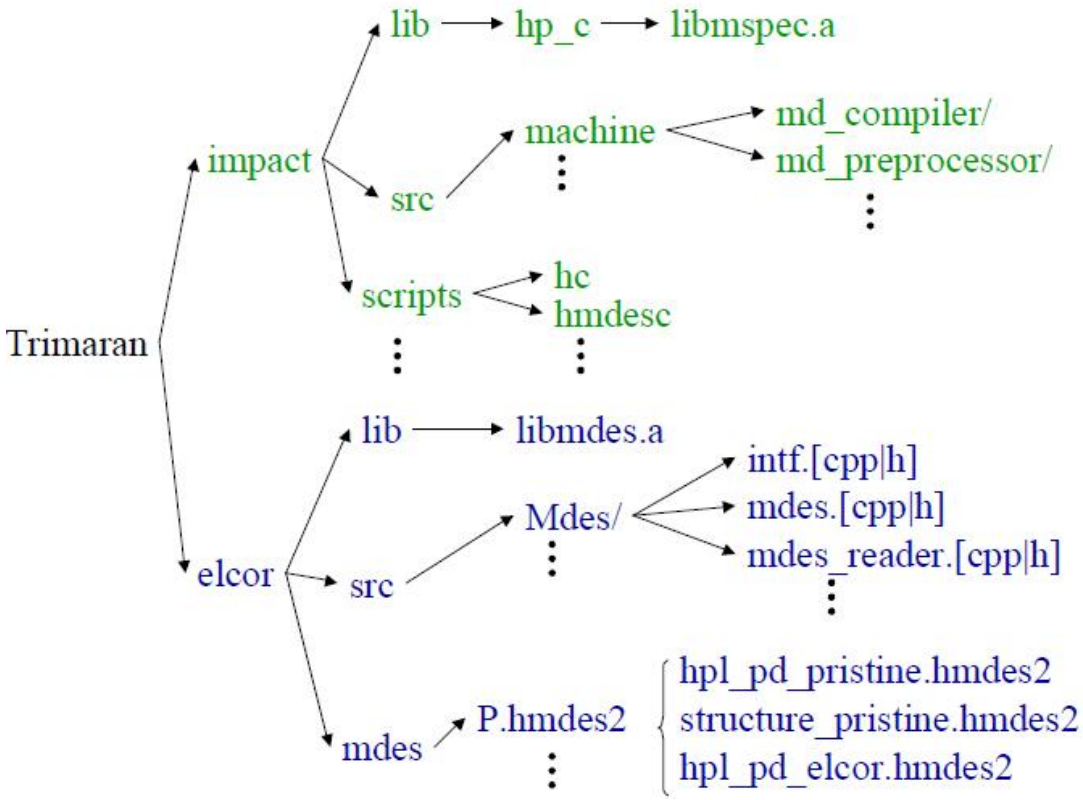


Figure 5 : les répertoires relatifs à MDES dans trimaran

Les fichiers de description de machine

Chaque exemple du processeur Hpl-pd est décrit en utilisant quatre fichiers dans un format textuel. Ces quatre fichiers correspondant à un processeur (p) de Hpl-pd sont : **P.hmdes2** et les trois fichiers : **structure_pristine.hmdes2, hpl_pd_pristine.hmdes2**, et **hpl_pd_elcor.hmdes2.** Actuellement, l'utilisateur peut changer la machine en modifiant des paramètres dans le fichier **P.hmdes2** et le reste du compilateur localise ces changements.

5. Le port ARM

En plus de l'architecture HPL-PD traditionnelle, la version 4.0 de Trimaran a été également mise en communication avec le port ARM.

5.1 Installation du port ARM

Il faut d'abords s'assurer que l'environnement est correctement positionné, et qu'elcor est déjà compilé depuis **arm-trans** qui exige les bibliothèques d'Elcor.

Le processus d'installation est :

% cd $TRIMARAN_ROOT/arm

% make

La deuxième étape exige la création d'un compilateur en travers, Nous employons une vieille version de gcc parce qu'il le soutient une norme désapprouvée de virgule flottante qui est employée dans simulateur SimpleScalar pour exécuter les benchmarks par le port ARM.

On crée un répertoire où on installe les compilateurs et aller à ce répertoire. On l'appelle <CROSSDIR>.

% mkdir <CROSSDIR>

% cd <CROSSDIR>

Puis on télécharge le compilateur et les bibliothèques :

% wget http://cccp.eecs.umich.edu/trimaran/binutils-2.10.tar.gz

% wget http://cccp.eecs.umich.edu/trimaran/gcc-2.95.2.tar.gz

% wget http://cccp.eecs.umich.edu/trimaran/glibc-2.1.3-armlinux.tar.gz

On extraire chaque fichier avec la commande **tar**

- **Installez la bibliothéque binutils**

  % cd binutils-2.10

  % ./configure --target=arm-linux --prefix=<CROSSDIR>

  % make

  % make install

  % cd ..

On ajoute **<CROSSDIR>/bin** ou variable d'environnement, si on utilise **csh** ou **tcsh** on doit taper la commande **rehash** dans la console. Si on utilise, bash on n'a pas besoin de la commande **rehash.**

- **Installation de compilateur gcc-2.95**

  % cd ./gcc-2.95.2

  % ./configure --target=arm-linux --prefix=<CROSSDIR>

  % make LANGUAGES=c

  % make LANGUAGES=c install

  % cd ..

On modifie en suite le fichier **./lib/gcc-lib/arm-linux/2.95.2/specs** comme suit **:** on remplace toutes occurrences de **"elf32arm"** avec **"armelf linux".** Ceci fixe une incompatibilité entre gcc et des bibliothèques de GLIBC.

En fin, on tape la commande **rehash,** pour que le shell trouve les nouveaux exécutables.

Pour télécharger SimpleScalar ARM, on emplois la commande suivante :

% wget http://www.eecs.umich.edu/~taustin/code/arm/simplesim-arm-0.2.tar.gz

- Extraire SimpleScalar et l'intaller

  % tar xzf simplesim-arm-0.2.tar.gz

  % cd simplesim-4.0

  % make config-arm

  % make

On ajoute **simplesim-4.0** à la variable Path pour que le simulateur puise le trouver a travers le shell.

6. Installation de trimaran

Il y a deux méthodes d'installation de compilateur Trimaran

- Installation manuelle
- Installation de LivePC

Nous décrivons ici l'installation manuelle de Trimaran. Elle exige les packages suivants **:**

- Gcc
- Autoconf version 2.54
- automake version 1.7

6.1 Installation de l'environnement :

Plusieurs variables d'environnement sont exigées pour construire ou employer le compilateur Trimaran :

- Télécharger le compilateur trimaran dans : http://www.trimaran.org
- Extraire l'archive, et met le dans un répertoire ou en vue l'installe par exemple : /home/lila/trimaran

Les fichies envrc et envrc.bash placé dans /trimaran/scripts/ fournissant une manière de charger les variables dans notre shell d'environnement par l'intermédiaire de l'invité de commande. On modifie le fichier envrc[.bash] pour remplacer TRIMARAN ROOT avec le chemin a notre installation de Trimaran : /home/lila/trimaran.

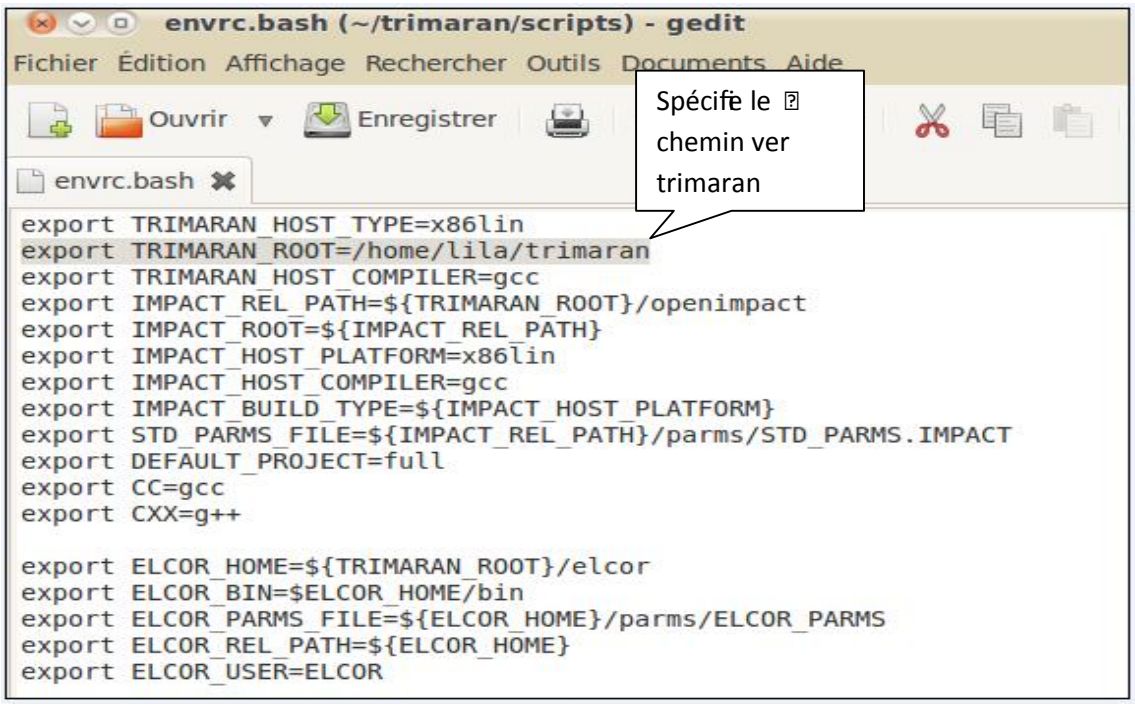


Figure 6 : le contenu de fichier envrc.bach

6.2 Installation

On commence par l'installation et compilation de deux compilateurs Openimpact et elcor. Pour cela on ouvre une console et on tape :

% source /home/lila/trimaran/scripts/envrc.bash

% cd openimpact

./install_openimpact

On installe le compilateur Elcor et le simulateur (simu).

% cd ../elcor

make

% cd ../simu

make

7. Ajouter notre Propre Code

Pour ajouter de nouvelles techniques d'analyse ou d'optimisations à Elcor, nous encapsulons notre code dans un répertoire. Ceci aide à préserver la modularité du compilateur et à coder l'organisation. Ici nous décrirons comment ajouter un nouveau répertoire source a Elcor de sorte qu'il se conforme au compilateur.

- On crée un répertoire :

% cd $ELCOR_HOME/src

% mkdir MyOpti

et entrer les fichiers appropriés dans ce répertoire.

On modifie le fichier **$ELCOR HOME/include/make** pour créer une entrée pour MyOpti comme les autres répertoires.

```
if [ -d ../src/MyOpti ] ; then
echo "Linking files from MyOpti dir"
ln -s ../src/MyOpti/*.h.;
fi;
```

Ceci assurera que tous les dossiers d'en-tête ont été créés dans **$ELCOR_HOME/src/MyOpti** et automatiquement incorporés dans **$ELCOR HOME/include** en analysant des dépendances avec **make depend**. Ceci permet également de trouver les fichiers d'en-tête nécessaires pendant la compilation dans d'autres répertoires.

- On crée un fichier **$ELCOR _HOME/src/MyOpti/MyOpti_all.cpp** et inclut tout les fichiers d'extensions *.cpp trouvés dans **$ELCOR HOME/src/MyOpti.** Par example

% cat MyOpti_all.cpp

#include "foo.cpp"

#include "bar.cpp"

#include "baz.cpp"

La raison de ce fichier est qu'elle accélère le temps d'exécution à la compilation. À savoir groupant tous les fichiers source dans un fichier simple évite d'inclure à plusieurs reprises les mêmes fichiers d'en-tête.

- On ouvre **$ELCOR HOME/src/Makefile** pour le modifier

7.1 Configuration de la machine MDES

Pour décrire notre description de machine MDES on aura besoin de changer la configuration du fichier hpl_pd_elcor .hmdes2 situé dans **/home/lila/trimaran/elcor/mdes**. On précise le nombre de clusters, le nombre d'unités fonctionnelles, la taille et latence des niveaux mémoires (SPM et DRAM).

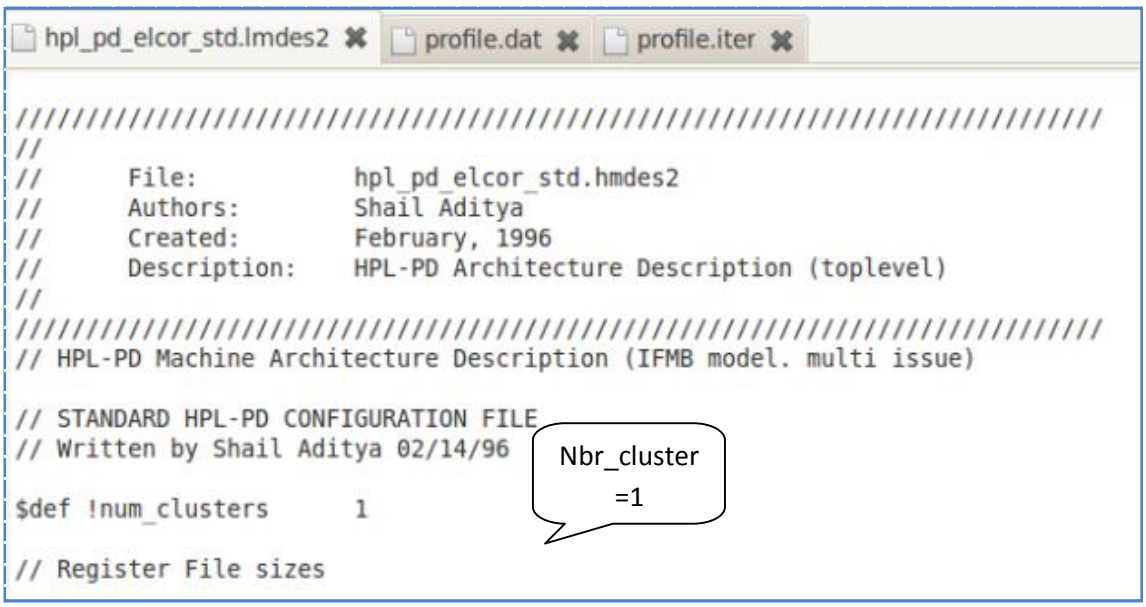


Figure 7 : spécification du nombre de cluster

Le nombre des unités fonctionnelles est égal à 1 pour chaque unité.

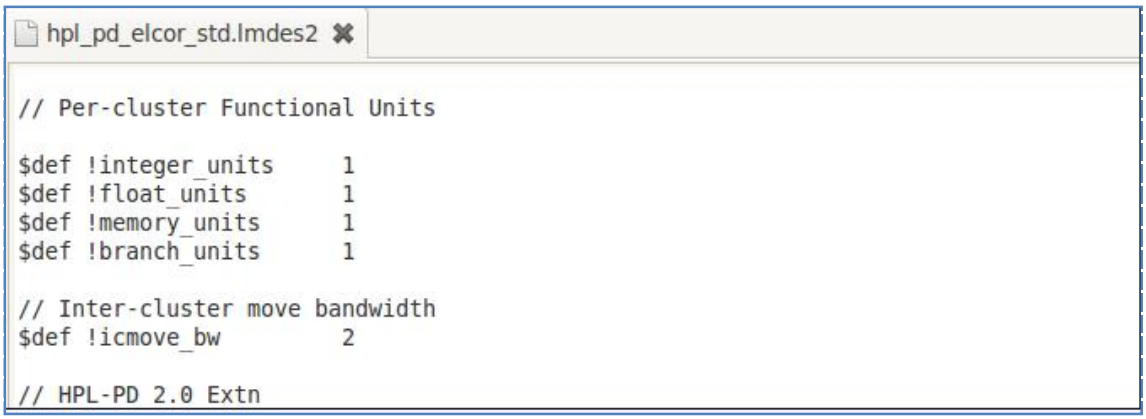


Figure 8 : spécification du nombre des unités fonctionnelles

La figure 9 montre la taille et latence des bancs mémoires.

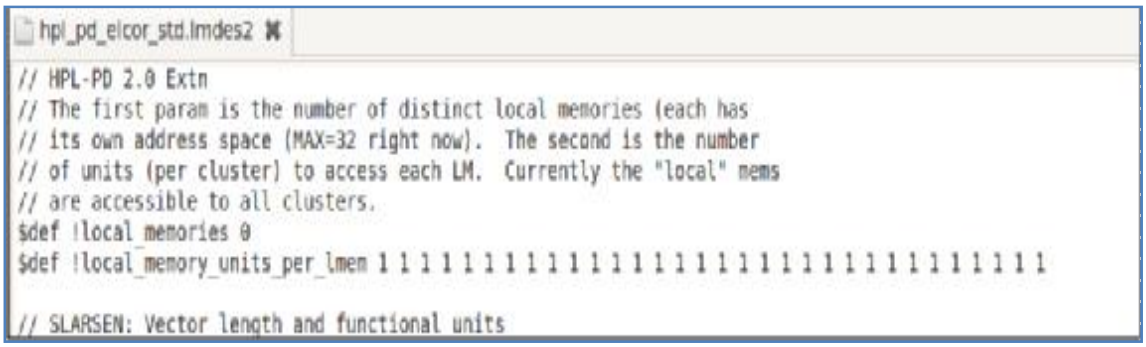


Figure 9 : spécification de taille de la mémoire

Apres avoir ajouté notre code d'optimisation dans /trimran/elcor/src/Myopti, on le compile et on l'exécute.

8. Exécution de trimaran

Le script principal tcc localisé dans /home/lila/trimaran/scripts exécutera la chaine d'outils de Trimaran. Le script tcc cherche le dossier des benchmarks dans le répertoire /home/lila/trimaran/benchmarks. Nous montrons comment exécuter Trimaran sur le benchmark appelé fir_int. Les commandes suivantes compilent et simulent fir_int :

% cd /tmp

% tcc -bench fir_int

Un dossier /tmp/fir_int_O est créé, et contiendra les dossiers intermédiaires correspondant à la compilation et simulation du benchmark fir_int. Ces dossiers sont utiles pour contrôler la correction, et l'analyse l'exécution du compilateur.

Le fichier /tmp/fir_int_o/elcor_intermediate/ELCOR STATS récapitulera quelques statistiques principales au sujet du benchmark.

% Sumstat -total -i ELCOR_STATS

Ces statistiques, qui incluent des évaluations d'exécution, sont basées sur l'information de profilage. Des résultats réels de simulation sont enregistrés dans : /tmp/fir_int_O/simu_intermediate/PD_STATS.

Les statistiques sont agrégées pour chaque fonction, puis totalisées pour le programme entier. Le simulateur localise un nombre des statistiques comprenant :

total cycles : simule le temps totale d'exécution du benchmark en cycles.

compute cycles : calcule le nombre de cycles de tout les opérations de branchement prévus, et toutes les opérations de mémoire (loads and stores). Les latences sont inférieures ou égales aux latences indiquées par MDES.

stall cycles : calcule le nombre de cycles ou le processeur perd de la vitesse dus ou opérations de branchement que le processeur peut pas la prédis et au système de mémoire retardé.

9. Les résultats de simulation pour les benchmarks.

Nous présentons ici une comparaison du nombre de cycles de benchmark avant l'optimisation et après l'optimisation :

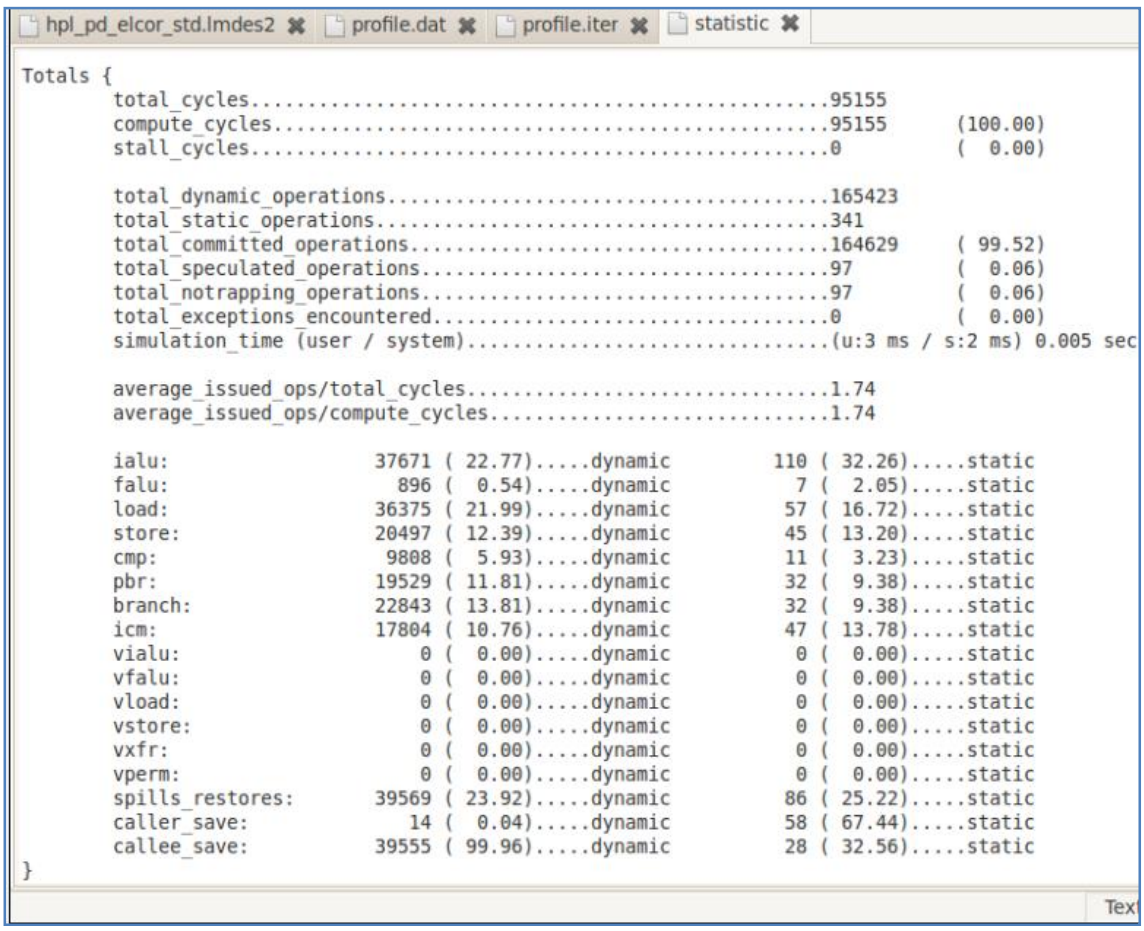


Figure 10. Les résultats de simulation avant l'optimisation

La figure 11 présente le nombre de cycles nécessaires pour l'exécution du benchmark après notre optimisation.

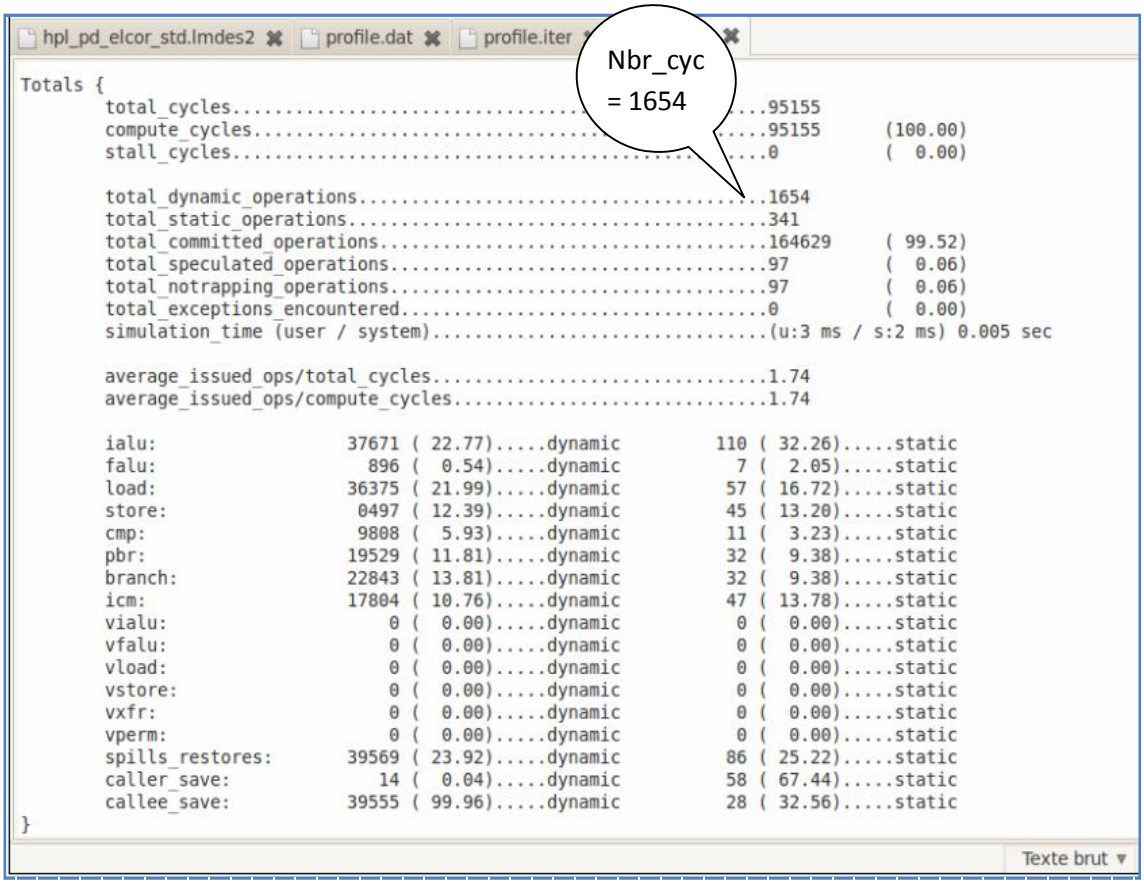


Figure 11 : Le nombre de cycle total d'évaluation.

La figure 11 montre que le nombre de cycles totaux de l'instruction total_dynamic _opiration =1654 (qui calcule le nombre de cycles totaux du programme lors d'exécution) après l'optimisation alors que avant l'optimisation de la mémoire total_dynamic_opiration=165423.

On dis que si on essaye de diminuer le nombre de cycle total d'exécution d'un programme, on diminue par la même l'énergie consommée par le système.

10. conclusion

Dans ce chapitre, nous avons présenté l'environnement de développement et l'implémentation et la simulation de notre application. Nous sommes encouragés par les résultats obtenus. Ils montrent les avantages d'une attribution optimale des variables globales sur les mémoires hétérogènes (SPM et DRAM) et son rôle dans l'optimisation de la consommation d'énergie.

Conclusion générale

Conclusion générale

Notre travail ce situe dans le cadre de la recherche d'optimisations pour la mémoire dans les systèmes embarqués, qui est un domaine vaste, complexe et passionnant.

Dans le chapitre I, j'ai présenté un état de l'art sur les mémoires et la consommation d'énergie, et quelque d'optimisations de bas niveau.

Dans le chapitre II, j'ai présenté la plupart des techniques d'optimisation logicielles et de gestion de la mémoire SPM existantes.

Dans le chapitre III, on a d'écrit la solution d'Oren Avissar et Rajeev Barua, pour une bonne attribution des données sur les mémoires hétérogènes d'une hiérarchie mémoire.

L'objectif de cette étude est d'implémenter la méthode d'Oren Avissar et Rajeev Barua, pour bien initialiser dans ce domaine.

L'approche présentée dans le chapitre IV, apporte une certaine technique pour le processus d'optimisation, et cela par l'utilisation de données de profilage pour collecter des informations sur les fréquences d'accès afin de placer les variables fréquemment utilisées dans la mémoire rapide SPM.

Un prototype concrétisant cette approche est développé. Son implémentation a fait l'objet du chapitre V. L'environnement de développement assure les différentes tâches relatives à l'implémentation de la méthode et la simulation des résultats sur des benchmarks installé dans le compilateur trimaran.

Le travail que nous avons accompli nous a permis :

- D'avoir les notions de bases d'une nouvelle discipline en pleine expansion si bien applicable dans l'optimisation de la mémoire pour les systèmes embarqués, et découvrir un grand nombre de techniques et de solutions de gestion des mémoires SPM.
- De nous initier aux différentes étapes à suivre pour la conduite d'un vrai projet de recherche en incluant la définition des besoins.

- Les techniques d'optimisation intégrés dans un compilateur dédié pour les systèmes embarqués.

- D'acquérir de nouvelles connaissances sur les langages C, C++, des connaissances dans le domaine de compilation, et aussi d'utiliser et de maîtriser plusieurs logiciels tels que le compilateur Trimaran, les benchmarks, port ARM et simulateur.

Les perspectives

Avec un peu de temps que nous avions pour mener ce travail, nous pouvons admettre qu'il y a beaucoup de perspectives pour notre travail :

- nous aimerions comparer la consommation d'énergie de l'architecture décrite par la description de machine MDES.

- Implémenter les variables de pile, et de tas.

- Implémenter la méthode dans une architecture multi-clusters.

Bibliographie

[Absar et Catthoor, 2005] M. J. Absar et F. Catthoor. Compiler-based approach for exploiting scratchpad in presence of irregular array access. In *DATE*, 2005.

[Athavale *et al.*, 2001] R. Athavale, N. Vijaykrishnan, M. T. Kandemir, et M. J. Irwin. Influence of array allocation mechanisms on memory system energy. In *IPDPS*, page 3, 2001.

[Avissar *et al.*, 2001] O. Avissar, R. Barua, et D. Stewart. Heterogeneous memory management for embedded systems. In *CASES*, 2001.

[Avissar *et al.*, 2002] O. Avissar, R. Barua, et D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transaction. on Embedded Computing Systems.*, 1(1) :6–26, 2002.

[Banakar *et al.*, 2001] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, et P. Marwedel. *Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption.* Technical Report 762. University of Dortmund, Sept. 2001.

[Banakar *et al.*, 2002] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, et P. Marwedel. Scratchpad memory : design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[Benini *et al.*, 2000] L. Benini, A. Macii, E. Macii, et M. Poncino. Increasing Energy Efficiency of Embedded Systems by Application Specific Memory Hierarchy Generation. *IEEE Design and Test*, 17(2) :74–85, 2000.

[Benini et Micheli, 1999] L. Benini et G. De Micheli. System-level power optimization : Techniques and tools. In *ISLPED-99 :ACM/IEEE*, pages 288–293, 1999.

[Benini et Micheli, 2000] L. Benini et G. DeMicheli. System-level power optimization : techniques and tools. *IEEE Design and Test*, 17(2) :74–85, 2000.

[Dominguez *et al.*, 2005] A. Dominguez, S. Udayakumaran, et R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(2) :115–192, 2005.

[Egger *et al.*, 2006] B. Egger, J. Lee, et H. Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *EMSOFT*, 2006.

[Fradj *et al.*, 2005] H. Ben Fradj, A. El Ouardighi, C. Belleudy, et M. Auguin. Energy aware memory architecture configuration. 33(3) :3–9, 2005.

[Graybill et all, 2002] Graybill R., Melhem R., *Power aware computing*, Kluwer Academic Publishers, Norwell, MA,
USA, 2002.

[Hallnor et Reinhardt, 2000] G. Hallnor et S. K. Reinhardt. A fully associative software-managed cache design. In *27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[H.Changet et al .1999] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Boston : Kluwer Academic Publisher, 1999. 235 p. ISBN 0–7923–8679–5.

[Hennessy et Patterson, 2003] J-L. Hennessy et D. Patterson. *Architectures des ordinateurs : Une approche quantitative, 3e édition.* Vuibert, 2003.

[Hom et all, 2001] Hom J., Kremer U., « Energy Management of Virtual Memory on Diskless Devices. », *Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, Barcelone, Espagne, septembre, 2001.

[Idrissi Aouad et Zendra, 2007] M. Idrissi Aouad et O. Zendra. A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, Berlin, Germany. July 2007. A paraître.

[Kandemir *et al.*, 2000] M. T. Kandemir, I. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, et A. Parikh. Dynamic Management of Scratch-pad Memory Space. In *Pmc. DAC*, 2001.

[Kandemir *et al.*, 2004] M. T. Kandemir, M. J. Irwin, G. Chen, et I. Kolcu. Banked scratch-pad memory management for reducing leakage energy consumption. In *ICCAD*, pages 120–124, 2004.

[Kandemir *et al.*, 2005] M. T. Kandemir, M. J. Irwin, G. Chen, et I. Kolcu. Compiler-guided leakage optimization for banked scratch-pad memories. *IEEE Trans. VLSI Syst.*, 13(10) :1136–1146, 2005.

[Manish Verma et Peter.2007] Manish Verma, Peter Marwedel in Advanced Memory Optimization Techniques for Low-Power Embedded Processors, *Technology Center, High Wycombe, UK Altera european*

[Nguyen *et al.*, 2005] N. Nguyen, A. Dominguez, et R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES*, 2005.

[Olivier, 2005] *Olivier rochecouste , Architecture et bits signi_catifs, l'universit_e de Rennes1, 2005*

[Panda *et al.*, 1997] P. R. Panda, N. Dutt, et A. Nicolau. *Data Cache Sizing for Embedded Processor Applications.* Technical Report ICS-TR-97-31. University of California, Irvine, June 1997 and Architectural exploration and optimization of local memory in embedded systems. In *International Symposium on System Synthesis (ISSS 97), Antwerp*, Sept. 1997.

[Poletti *et al.*, 2004] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, et J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.

[Shiue et Chakrabarti, 1999] W-T. Shiue et C. Chakrabarti. Memory exploration for low power, embedded systems. In *DAC*, 1999.

[Steinke *et al.*, 2001] S. Steinke, M. Knauer, L. Wehmeyer, et P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc. of PATMOS*, 2001.

[Steinke *et al.*, 2002] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, et P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory.In *ISSS*, 2002.

[Steinke *et al.*, 2002] S. Steinke, L. Wehmeyer, B. Lee, et P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, page 409. IEEE Computer Society, 2002.

[Stéphane et al, 2005-2009] *Stéphane Mancini, Vincent Fristot, Dominique Houzet, Sylvain Huet, Adéquation Algorithme Architecture,2005-2009*

[Udayakumaran *et al.*, 2006] S. Udayakumaran, A. Dominguez, et R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Embedded Comput. Syst*, 5(2) :472–511, 2006.

[Udayakumaran et Barua, 2003] S. Udayakumaran et R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES*, pages 276–286. ACM Press, 2003.

[Udayakumaran et Barua, 2006] S. Udayakumaran et R. Barua. An integrated scratch-pad allocator for affine and non-affine code. In *DATE*, pages 925–930, 2006.

[Verma *et al.*, 2003] M. Verma, S. Steinke, et P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *ASPDAC*, 2003.

[Verma *et al.*, 2004] M. Verma, L.Wehmeyer, et P.Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *CODES+ISSS*, 2004.

[Verma *et al.*, 2006] M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, et L. Benini. Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations. In *SAMOS*, pages 279–288, 2006.

[Wehmeyer *et al.*, 2004] L. Wehmeyer, U. Helmig, et P. Marwedel. Compiler-optimized usage of partitioned memories. In *WMPI*, 2004.

[Wilton et Jouppi, 1996] S.J.E. Wilton et N.P. Jouppi. Cacti : An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 1996.

[Woo et all, 2001] Woo S., Yoon J., Kim J., « Low-Power Instruction Encoding Techniques », *SOC Design Conference*,2001.

[Zendra, 2006] O. Zendra. Memory and compiler optimizations for low-power and -energy. In *ICOOOLPS*, 2006.

[Zhang et all, 2003] Zhuang X., Lau C., Pande S., « Storage assignment optimizations through variable coalescence for embedded processors », *LCTES '03 : Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, ACM Press, New York, NY,USA, p. 220-231, 2003.