RENPAR'14 / ASF / SYMPA
Hamamet, Tunisie, 10 – 13 avril 2002

Une méthode d'amélioration de la localité basée sur des estimations asymptotiques du trafic

Cédric Bastoul

Université de Versailles Saint-Quentin-en-Yvelines,
PRiSM, Bâtiment Descartes, 45 avenue de États-Unis,
78035 VERSAILLES - France
cedric.bastoul@prism.uvsq.fr

Résumé
Les mémoires cache ont été introduites pour réduire l'influence de la lenteur des mémoires principales sur les performances des processeurs. Elles n'apportent cependant qu'une solution partielle, et de nombreuses recherches ont été menées dans le but de transformer les programmes afin d'optimiser leur utilisation. La difficulté de modéliser précisément les échanges entre les différents niveaux de mémoire a conduit à l'utilisation de méthodes majoritairement heuristiques. Nous proposons dans cet article un modèle dans lequel une évaluation réelle du trafic de la mémoire est possible. Nous exploitons cette information pour trouver un meilleur ordonnancement des opérations et générer des codes optimisés. L'implémentation, puis l'expérimentation sur des problèmes non triviaux, nous a permis de juger de l'intérêt de notre approche pour l'amélioration de la localité et des performances.

Mots-clés : localité temporelle, mémoire cache, transformation de programmes, compilation.

---

1. Introduction

L'amélioration de la finesse de gravure des circuits intégrés se traduit en augmentation de la vitesse pour les processeurs, mais en augmentation de la capacité pour les mémoires. Si rien n'était fait, les gains de puissance des processeurs ne pourraient être exploités, faute de mémoire suffisamment rapide pour les alimenter. Les mémoires hiérarchiques sont une solution très répandue, peu coûteuse et souvent efficace. Pour autant, elles ne sont exemptes ni de limitations ni de défauts. D'une part, on sait que leur mécanisme convient mal aux programmes de calcul numérique qui utilisent de grandes structures de données de façon régulière. D'autre part, leur comportement difficilement prédictible les rend inadaptées aux systèmes temps réel. Enfin, la forte consommation en énergie que requièrent les transferts de données entre les différents niveaux de mémoire freine leur exploitation au sein des systèmes embarqués.

Le nombre de travaux ayant pour objectif d'améliorer la gestion de la hiérarchie des mémoires est en rapport avec l'importance des enjeux. Deux approches sont à distinguer. La première consiste à développer et utiliser des bibliothèques extrêmement optimisées, telles LAPACK [2]. Pour les architectures et les problèmes supportés, les solutions proposées sont alors souvent les plus intéressantes. La seconde approche vise à construire des compilateurs capables d'optimiser automatiquement les codes soumis par les programmeurs. Cette solution a l'avantage de l'adaptabilité tant au niveau des programmes que des architectures cibles. C'est dans cette seconde approche que s'incrit notre travail.

Les compilateurs optimiseurs ont parmi leurs principaux objectifs de transformer les programmes afin d'utiliser au mieux la hiérarchie des mémoires. Il s'agit pour eux d'exploiter au maximum la réutilisation des données. Pour cela, le principe classique est d'appliquer des transformations de boucles [17, 12] guidées par un modèle de coût [14] puis d'effectuer un pavage [18] de taille convenablement choisie [5]. Cette démarche a en particulier pour défaut de ne pouvoir s'appliquer que sur des nids de boucles parfaits. On est ainsi souvent amené à convertir des nids de boucles. Du résultat de cette conversion dépend la qualité du pavage final ; or il n'y a pas de méthode systématique pour l'effectuer au mieux [9].

Alternativement à cette approche centrée sur le contrôle, le *data-shackling* propose de raisonner sur les données [10]. Son principe est d'agir directement sur le transfert des données plutôt que par le biais de manipulations des structures de contrôle.

Tous ces algorithmes reposent en général sur des approches heuristiques. Par exemple, considérons deux accès à la même cellule mémoire. Il paraît vraisemblable que plus ces accès seront proches dans le temps, plus le second aura de chances d'être un succès. On cherchera donc à transformer les boucles pour que ces deux accès soient les plus proches possible. Nous voulons au contraire nous appuyer sur une évaluation au moins approximative du trafic entre cache et mémoire principale, et trouver le programme qui minimise ce trafic. La méthode proposée consiste en un découpage du programme en *chunks*. Nous en présentons le principe en section 2. La section 3 est consacrée à la construction de chunks respectant les dépendances et minimisant le trafic. La section 4 montre comment on génère le programme objet quand le système de chunks est donné. Nous terminerons en présentant quelques résultats de notre approche et en indiquant les problèmes qui restent en suspens.

2. Principe

Le problème de l'évaluation du trafic en mémoire est difficile à résoudre exactement pour un modèle détaillé de cache. Nous proposons donc les simplifications suivantes :

- le cache est supposé totalement associatif ; cette hypothèse est presque satisfaite par les caches modernes à grand degré d'associativité, et on peut corriger cette approximation en utilisant une taille de cache effective un peu inférieure à la taille physique ;
- la modélisation du mécanisme de remplacement est très complexe ; nous allons donc le mettre hors jeu par le mécanisme du *chunking* qui sera expliqué plus loin ;
- enfin, nous nous contenterons d'estimations asymptotiques du trafic ; nous réservons pour des travaux futurs la question de savoir si une meilleure estimation peut influencer l'allure du programme résultant.

Le principe de notre méthode est de partitionner l'ensemble des opérations d'un programme en sous-ensembles plus petits qui n'utilisent pas plus de données que le cache ne peut en contenir : les *chunks*. Le programme ainsi découpé est ensuite exécuté chunk par chunk, en considérant que la mémoire cache est vidée avant chacun d'eux. Ces sous-ensembles doivent être tels que leur exécution suivant un ordre déterminé soit équivalente à l'exécution du programme original. En pratique, on numérotera les chunks de manière croissante, suivant l'ordre dans lequel ils devront être exécutés. Pour résoudre ce problème, nous cherchons pour chaque instruction S, une *fonction de chunking* θ_S qui, à un vecteur d'itération x, associe un numéro de chunk $\theta_S(x)$. Un exemple d'application du chunking est présenté en figure 1 sur

Programme original	Programme transformé

`do i=1, n`
`  a(i) = i            /*S1*/`
`  do j=1, m`
`    b(j) = b(j) + a(i) /*S2*/`
`  enddo`
`enddo`
	`do c=1, n`
	`  i = c`
	`  a(i) = i            /*S1*/`
	`enddo`
	`do c=n+1, n+m`
	`  do i=1, n`
	`    j = c - n`
	`    b(j) = b(j) + a(i) /*S2*/`
	`  enddo`
	`enddo`

FIG. 1 – Exemple de transformation par chunking

un problème simple. Dans cet exemple, l'ordre des opérations a été bouleversé pour une exploitation maximale de la localité temporelle. Dans le programme résultat, la variable `c` donne à tout instant le

numéro de chunk dans lequel on se trouve. Les fonctions de chunking utilisées sont $\theta_{S1}\left(\left[\begin{array}{c} i \end{array}\right]\right) = \left[\begin{array}{c} i \end{array}\right]$ et $\theta_{S2}\left(\left[\begin{array}{c} i \\ j \end{array}\right]\right) = \left[\begin{array}{c} j+n \end{array}\right]$. Nous détaillerons cet exemple tout au long de l'article.

3. Recherche des fonctions de chunking

On peut décrire un système de chunks à l'aide de deux grandeurs. Tout d'abord l'*empreinte*, qui est pour chaque chunk l'ensemble des cellules mémoire accédées par les opérations de ce chunk. Ensuite le *trafic*, le nombre de va-et-vient entre cache et mémoire centrale. Nous cherchons à construire un système optimal. Ce sera le cas lorsque d'une part chaque empreinte de chunk pourra tenir dans la mémoire cache, et d'autre part quand chaque cellule apparaîtra dans le moins d'empreintes possible, c'est à dire lorsque le trafic sera minimal. Afin d'être capable de reconstruire le code, nous chercherons des fonctions de chunking affines. Pour une opération $S[x]$, instance de l'instruction S pour un vecteur d'itération x dans le domaine d'itération D_S, le numéro de chunk s'écrira :

$$\theta_S(x) = Tx + d.$$

T est la matrice de chunking de dimension $g \times \rho(S)$ avec g le plus grand nombre de dimensions des tableaux accédés dans l'instruction considérée et $\rho(S)$ le nombre de boucles englobant cette instruction ; d est quant à lui un vecteur constant.

3.1. Évaluation asymptotique

Dans notre modèle, il est possible d'estimer les grandeurs des empreintes et du trafic. Pour une instruction S, un tableau A et une fonction d'index f, l'empreinte d'un chunk t est l'ensemble des données que l'on peut accéder durant l'exécution de ce chunk :

$$E_{S,A,f}(t) = \left\{ f(x) \mid x \in D_S, \theta_S(x) = t \right\}.$$

On considère que le cache est vide avant l'exécution de chaque chunk : une même donnée accédée dans différents chunks comptera donc pour autant de fois dans le calcul du trafic. On peut alors voir le trafic comme le nombre de couples ⟨donnée, numéro de chunk⟩ possibles :

$$\mathcal{T}_{S,A,f} = \text{Card}\left\{ \left\langle f(x), \theta_S(x) \right\rangle \mid x \in D_S \right\}.$$

Nous nous plaçons dans l'hypothèse classique où le programme original est à contrôle statique [7], la fonction d'index est donc affine et s'écrit : $f(x) = Fx + a$, où F est la matrice d'index de dimension $\rho(A) \times \rho(S)$, avec $\rho(A)$ le nombre de dimensions du tableau A, et a un vecteur constant qu'on ignorera. Les ordres de grandeur des cardinaux des ensembles décrivant empreinte et trafic sont alors connus. On sait que si chaque élément de x est un entier appartenant à un segment de longueur m, alors :

$$\begin{aligned} \text{Card } E_{S,A,f}(t) &= O\left(m^l\right), \text{ avec } l = \text{rang} \left(\begin{array}{c} T \\ F \end{array}\right) - \text{rang } T, \\ \mathcal{T}_{S,A,f} &= O\left(m^k\right), \text{ avec } k = \text{rang} \left(\begin{array}{c} T \\ F \end{array}\right), \end{aligned}$$

où $\left(\begin{array}{c} T \\ F \end{array}\right)$ est une matrice constituée pour ses premières lignes de la matrice T et pour les lignes suivantes de la matrice F. La matrice F peut être obtenue par analyse du code source, T quant à elle est l'inconnue qu'il s'agit de découvrir.

3.2. Construction des matrices de chunking

On dispose grâce aux évaluations d'un moyen de quantifier la qualité d'un chunking. Dans le cas d'une instruction comportant n références, les matrices d'index F_i pour $1 \leq i \leq n$ sont connues, on cherche alors à construire la matrice de chunking T ayant les meilleures propriétés. Cette construction est guidée par les évaluations. Nous effectuons tout d'abord une énumération dans l'ordre du trafic croissant des tuples $\left\langle \text{rang } T, \text{rang} \left(\begin{array}{c} T \\ F_i \end{array}\right) \text{ pour } 1 \leq i \leq n \right\rangle$ tels que chaque empreinte générée puisse tenir dans le cache. Nous tentons ensuite de construire T sous les conditions de rang du meilleur tuple possible.

Pour une référence isolée, construire une matrice T de rang v telle que rang $\begin{pmatrix} T \\ F \end{pmatrix} = w$ est toujours possible pourvu que v et w soient des valeurs acceptables et compatibles entre elles. On constitue pour cela une matrice génératrice avec en particulier $\rho(S) - w$ vecteurs d'une base de $\ker F$, puis on en calcule l'inverse. T sera alors composée de v lignes convenablement choisies de la matrice résultat puis complétée de lignes nulles si nécessaire.

Pour généraliser à n références, on doit combiner les n exigences rang $\begin{pmatrix} T \\ F_i \end{pmatrix} = w_i$ pour $1 \leq i \leq n$. La matrice génératrice doit posséder pour chaque référence exactement $\rho(S) - w_i$ vecteurs d'une base de $\ker F_i$ pour un total d'au plus v vecteurs. Une telle matrice n'existe pas toujours. Le choix des vecteurs à inclure dans la matrice génératrice est déterminant. On peut le guider en préférant ajouter pour chaque référence le plus possible de vecteurs déjà présents dans la matrice. S'il n'existe pas de solution pour un tuple, alors il faut tenter d'en trouver une pour le prochain tuple le plus intéressant.

Il existe toujours une matrice de chunking possible telle que les empreintes tiennent dans le cache. En effet, la contrainte la plus dure pour les empreintes est d'avoir une taille en $O(m^0)$, et la dernière possibilité tentée sera le tuple $\langle \rho(S), w_i = \rho(S) \text{ pour } 1 \leq i \leq n \rangle$. Le chunking correspondant génère pour chaque référence une empreinte en $O\left(m_i^0\right)$ et le trafic maximum en $O\left(m_i^{\rho(S)}\right)$. Sa solution $T = Id$ existe toujours et correspond au chunking trivial où chaque opération est dans un chunk.

Exemple Soit le code original de la figure 1. On suppose que `a` est un tableau de n éléments capable de tenir dans le cache et que `b` est un tableau de `m` éléments ne pouvant pas tenir dans le cache. Les ordres de grandeur acceptables pour la taille des empreintes sont donc $O\left(n^1\right)$ et $O\left(m^0\right)$. Le programme est constitué de deux instructions :

- l'instruction S1 a une seule référence au tableau `a` avec pour matrice d'index $F_{S1,1} = \begin{bmatrix} 1 \end{bmatrix}$. La matrice T_{S1} ayant les meilleures propriétés correspond au tuple $\langle 1, 1 \rangle$, elle génèrera des empreintes de tailles en $O\left(n^1\right)$ et un trafic en $O\left(n^1\right)$. On construit $T_{S1} = \begin{bmatrix} 1 \end{bmatrix}$;
- l'instruction S2 a deux références, l'une au tableau `a` avec pour matrice d'index $F_{S2,1} = \begin{bmatrix} 1 & 0 \end{bmatrix}$ et l'autre au tableau `b` avec pour matrice d'index $F_{S2,2} = \begin{bmatrix} 0 & 1 \end{bmatrix}$. La matrice T_{S2} ayant les meilleures propriétés correspondrait au tuple $\langle 1, 2, 1 \rangle$, elle génèrerait des empreintes de tailles en $O\left(m^0 + n^1\right)$ et un trafic en $O\left(m^1 + n^2\right)$. La construction est possible et donne $T_{S2} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

■

3.3. Validité

Puisque le chunking est une technique de réordonnancement des opérations, on doit s'assurer qu'il ne viole aucune dépendance. Rappelons que les chunks sont numérotés dans l'ordre dans lequel ils doivent être exécutés. À l'intérieur de chacun d'eux, les opérations respectent l'ordre séquentiel original. Considérons $I_{\mathcal{P}}$ l'ensemble des instructions du programme $\mathcal{P}$, et $\delta_{\mathcal{P}}$ la relation de dépendance sur $\mathcal{P}$, un système de chunks est alors valide si et seulement si :

$$\forall S, R \in I_{\mathcal{P}}, \quad S[x] \, \delta_{\mathcal{P}} R[y] \Rightarrow \theta(S[x]) \leq \theta(R[y]).$$

La construction des matrices de chunking est réalisée pour chaque instruction indépendamment les unes des autres. À ce stade, rien n'interdit un entrelacement illégal des opérations. Il est possible de corriger les fonctions de chunking afin qu'elles respectent les dépendances. On dispose en effet sur ces fonctions de certains degrés de liberté que l'on peut exploiter. D'une part, on peut appliquer aux matrices de chunking toutes les transformations qui ne changent pas leurs propriétés de rang. Et d'autre part, on peut renseigner les éléments des vecteurs constants d des fonctions de chunking. Pour cela, nous calculons l'espace des transformations possible, dit espace de Farkas [8], puis nous résolvons un problème de programmation linéaire en nombres entiers dans cet espace. Ces opérations sont répétées autant de fois qu'il y a de lignes dans la plus grande matrice de chunking. La solution si elle existe donne pour chaque instruction les composantes du vecteur d et les transformations à effectuer sur chaque matrice de chunking. Si aucune solution n'existe, nous devons retourner à la construction des matrices de chunking et tenter la proposition suivante.

Un chunking valide tel que les empreintes tiennent dans le cache existe toujours. Il correspond à la dernière proposition tentée, dans laquelle toutes les matrices de chunking sont des matrices identité. On retrouve alors le programme original, avec un chunk par opération et un trafic maximal.

Exemple Reprenons l'exemple en section 3.2. Si on se contentait d'utiliser les matrices trouvées, nous obtiendrions les fonctions de chunking suivantes :
- $\theta_{S1}\left(\left[\begin{array}{c} i \end{array}\right]\right) = \left[\begin{array}{c} 1 \end{array}\right]\left[\begin{array}{c} i \end{array}\right] + \left[\begin{array}{c} 0 \end{array}\right] = \left[\begin{array}{c} i \end{array}\right]$,
- $\theta_{S2}\left(\left[\begin{array}{c} i \\ j \end{array}\right]\right) = \left[\begin{array}{cc} 0 & 1 \\ 0 & 0 \end{array}\right]\left[\begin{array}{c} i \\ j \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \end{array}\right] = \left[\begin{array}{c} j \\ 0 \end{array}\right]$.

Ces fonctions ne décrivent pas un chunking valide. En effet, la dépendance de S1 vers S2 est violée. Par exemple, l'opération S2 $\left[\begin{array}{c} 2 \\ 1 \end{array}\right]$ est exécutée dans le chunk numéro 1 alors que l'opération S1 $\left[\begin{array}{c} 2 \end{array}\right]$ dont elle dépend est exécutée après, dans le chunk numéro 2. Notre méthode permet de corriger ce chunking afin que toutes les dépendances soient respectées et les propriétés du chunking conservées. La correction proposée par notre prototype est la suivante :
- $\theta_{S1}\left(\left[\begin{array}{c} i \end{array}\right]\right) = \left[\begin{array}{c} 1 \end{array}\right]\left[\begin{array}{c} i \end{array}\right] + \left[\begin{array}{c} 0 \end{array}\right] = \left[\begin{array}{c} i \end{array}\right]$,
- $\theta_{S2}\left(\left[\begin{array}{c} i \\ j \end{array}\right]\right) = \left[\begin{array}{cc} 0 & 1 \\ 0 & 0 \end{array}\right]\left[\begin{array}{c} i \\ j \end{array}\right] + \left[\begin{array}{c} n \\ 0 \end{array}\right] = \left[\begin{array}{c} j+n \\ 0 \end{array}\right]$.

Pour homogénéiser les fonctions de chunking, on peut ajouter des dimensions nulles ou en enlever si elles sont nulles pour toutes les fonctions, puisque cela ne change pas les rangs. On a finalement $\theta_{S1}\left(\left[\begin{array}{c} i \end{array}\right]\right) = \left[\begin{array}{c} i \end{array}\right]$ et $\theta_{S2}\left(\left[\begin{array}{c} i \\ j \end{array}\right]\right) = \left[\begin{array}{c} j+n \end{array}\right]$. ■

4. Génération de code

La génération de code est la dernière étape vers l'obtention du programme final. Elle est souvent ignorée en dépit de son influence sur la qualité du code produit. Il faut en particulier éviter qu'une mauvaise gestion des structures de contrôle ne détériore les performances. Comme le programme original est à contrôle statique, le domaine d'exécution de chaque instruction peut être décrit par un polyèdre [11]. Dans le cas d'un chunking, on ajoutera à ce polyèdre autant de dimensions et de contraintes que le numéro de chunk en possède. La génération de code est ensuite un problème bien connu de parcours de polyèdre [1] dont la solution la plus aboutie est celle de Quilleré et al. [15]. Cette technique génère chaque niveau de boucle par séparation des polyèdres de manière à ce qu'ils soient disjoints sur la dimension courante, puis récursion sur chacun d'eux pour générer le niveau supérieur, et enfin triage afin de respecter l'ordre lexicographique. Nous avons complété cette méthode pour la production de code non parallèle, de sorte que l'étude peut se faire sur des nids de boucles non parfaits pouvant contenir des instructions à tout niveau d'imbrication. On garantit alors la production d'un code particulièrement efficace pour le chunking donné.

Exemple Reprenons l'exemple en section 3.3. Les polyèdres décrivant les domaines d'exécution de S1 et S2 se déduisent de l'étude du code original. On les complète avec l'unique dimension du chunking c et les contraintes qu'elle porte. Les systèmes de contraintes décrivant les polyèdres sont alors :

Système de contraintes pour S1 Système de contraintes pour S2

$$\left\{\begin{array}{rcrcrl} c & - & i & & & = 0 \\ & - & i & + & n & \geq 0 \\ & & i & - & 1 & \geq 0 \end{array}\right. \qquad \left\{\begin{array}{rcrcrl} c & - & j & - & n & = 0 \\ & - & i & + & n & \geq 0 \\ & & i & - & 1 & \geq 0 \\ & - & j & + & m & \geq 0 \\ & & j & - & 1 & \geq 0 \end{array}\right.$$

Sur la première dimension, c, les deux polyèdres sont disjoints : le premier décrit $1 \leq c \leq n$ et le second $n+1 \leq c \leq n+m$. Il y aura donc un nid de boucle pour chaque instruction. La récursion sur ces nids est ensuite triviale puisqu'ils ne contiennent qu'une instruction chacun. Il s'agit enfin d'ordonner les nids de boucles pour respecter l'ordre lexicographique. On peut facilement remarquer que le premier polyèdre doit précéder le second. Le code produit est celui du programme transformé en figure 1. ■

5. Implémentation et résultats

De la recherche des fonctions de chunking à la génération de code, notre méthode a été complètement automatisée. Le prototype *Chunky* implémente l'ensemble du processus en langage C à l'exception du

calcul des dépendances et des espaces de Farkas où il utilise encore un code Maple. La résolution des dépendances et la génération de code font une utilisation intensive d'opérations sur les polyèdres. Nous avons pour cela utilisé la PolyLib [16] et PIP [6].

Cette automatisation nous a permis d'effectuer des tests sur différents problèmes non triviaux et ainsi d'évaluer l'intérêt de notre méthode. Nous avons choisi une évaluation aussi précise que possible en utilisant les compteurs matériels pour comparer les nombres de défauts de cache [3]. La machine de test utilisée possède un cache de niveau 1 de 16Ko et un cache de niveau 2 de 256Ko. Nous présentons en figure 2 l'évolution du nombre de défauts de cache des programmes en figure 1 en fonction de m. Nous avons fixé le rapport m/n à 64 de manière à mettre les phénomènes en évidence. On observe que dans

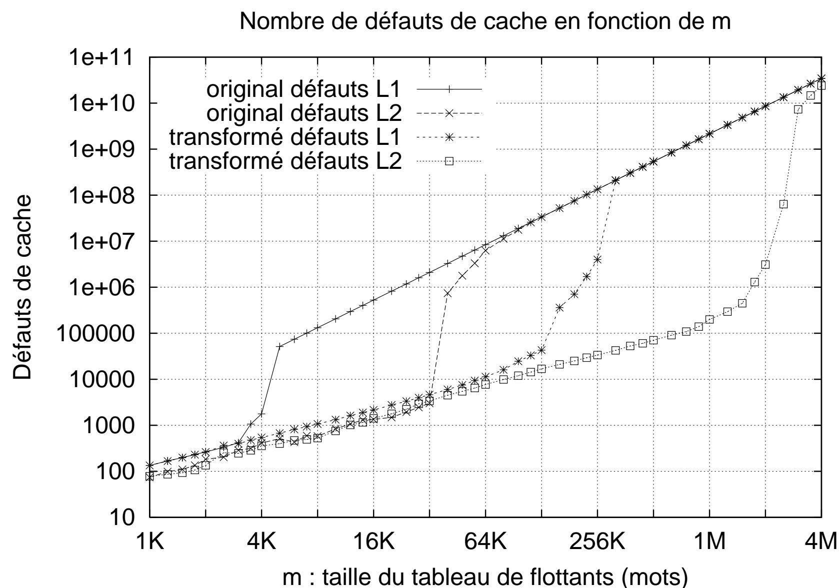

FIG. 2 – Comparaison en défauts de cache des programmes en figure 1

le cas du programme original, dès que le tableau b devient plus grand qu'un niveau de cache, le nombre de défauts sur ce niveau grandit brusquement. Le programme transformé a un meilleur comportement, puisqu'il réagit plus tard, quand c'est au tour du tableau a de ne plus tenir dans un niveau de cache. Nous avons pu observer le même type de comportement sur la plupart des programmes que nous avons testés. Quelques résultats sur des problèmes bien connus sont résumés en figure 3. On y montre le nombre de défauts de cache et les performances pour des tailles de tableaux $m \times m$ tels qu'ils ne tiennent plus en cache de niveau 1 ou 2. Pour les comparaisons, l'option de compilation est O3 pour les programmes originaux, et O1 pour les programmes transformés, afin d'éviter que le compilateur ne perturbe le chunking. Comme précédemment, les défauts de cache sont prévenus jusqu'au delà d'un ordre de grandeur. On peut observer la répercusion positive sur les performances, bien qu'elle ne soit pas garantie. En effet, malgré le soin apporté à la génération de code, il est parfois difficile d'éviter la présence de structures de contrôle très lourdes ; c'est le cas du programme Gauss - Jordan pour $m = 70$. Cependant, la pénalité pour un défaut de cache L2 étant de l'ordre de 10 fois plus grande que celle de L1, la réduction des défauts de cache L2 conduit presque systématiquement à une amélioration des performances.

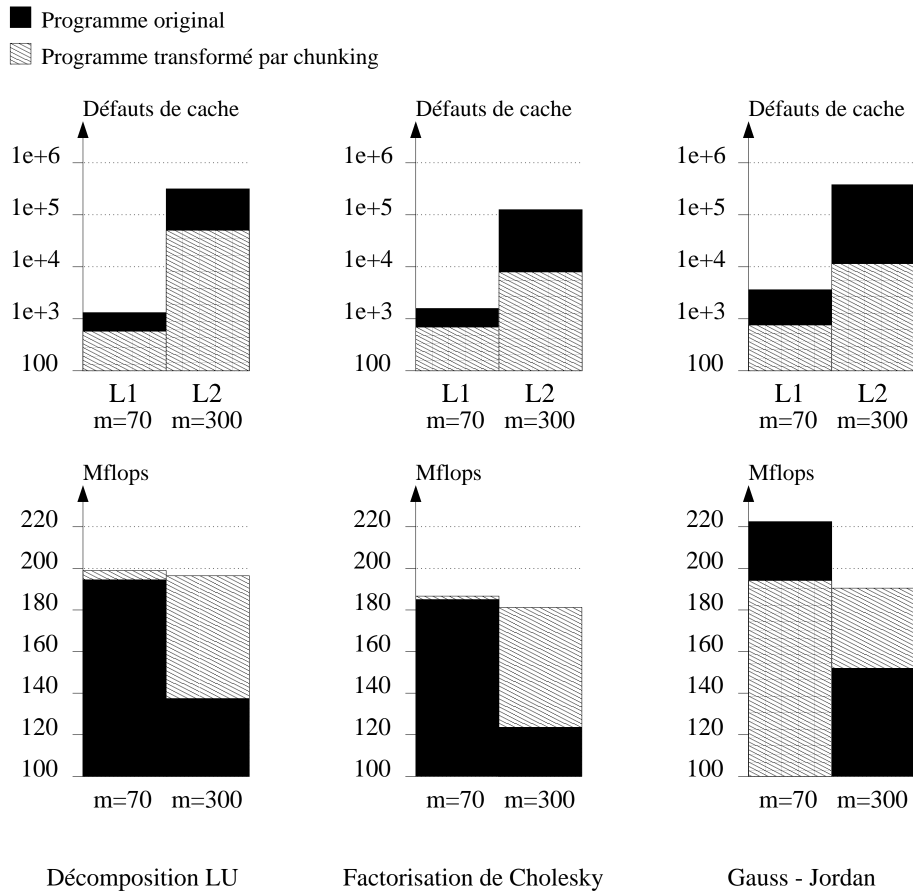

FIG. 3 – Résultats sur quelques problèmes courants

6. Conclusion et travaux futurs

Dans cet article, nous proposons une méthode d'amélioration de la localité basée sur des estimations du trafic. Ses avantages sont multiples. Tout d'abord, elle ne repose pas sur des heuristiques, et les transformations proposées garantissent la non altération de la localité temporelle, au pire en laissant le code original intact. Ensuite, elle s'adapte très bien face au problème des dépendances, car il est souvent possible de corriger une transformation sans diminuer sa qualité. Enfin, on peut l'appliquer sur tout programme à contrôle statique, sans autres restrictions. Cette méthode est entièrement automatique, et ne nécessite hormis le code original que les tailles relatives du cache et des données, les optimisations proposées restant stables sur de larges plages de tailles. Le temps de traitement est raisonnable (de l'ordre de la dixaine de secondes pour la factorisation de cholesky sur une machine à 900MHz) et sera fortement amélioré quand nous aurons reprogrammé les calculs des dépendances et des espaces de Farkas, qui occupent l'essentiel du temps.

Malgré des premiers résultats très encourageants, plusieurs extensions restent à apporter avant de rivaliser en pratique avec les techniques d'optimisation classiques. Nous travaillons actuellement à l'intégration de l'amélioration de la réutilisation de groupe et du pavage, qui semblent les prolongements naturels de notre approche. En effet, il s'agit intuitivement pour le premier d'ajouter des contraintes à la création du système de chunks, et pour le second, d'agréger des petits chunks ou de découper des gros. Pour être complet, il s'agira enfin d'aborder le problème de la localité spatiale, pour lequel des travaux basés sur des estimations fines des accès existent déjà [4, 13]. Nous envisageons ensuite de nous poser le problème de la reconnaissance de portions de code à contrôle statique dans un programme

quelconque, puis d'adaptater notre méthode aux systèmes disposant de mémoires locales.

Bibliographie

1. C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.
2. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide, Third Edition*. SIAM, 1999.
3. R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library : a common interface to access hardware performance counters on microprocessors. Technical Report FZJ-ZAM-IB-9816, Forschungszentrum Jlich, 1998.
   PCL est librement téléchargeable à l'adresse http ://www.kfa-juelich.de/zam/PCL.
4. Ph. Clauss and B. Meister. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. *ACM SIGARCH, Computer Architecture News*, 28(1), march 2000.
5. S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, june 1995.
6. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3) :243–268, 1988.
   PIP est librement téléchargeable à l'adresse http ://www.prism.uvsq.fr/~paf.
7. P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1) :23–53, february 1991.
8. P. Feautrier. Some efficient solutions to the affine scheduling problem, part I : one dimensional time. *International Journal of Parallel Programming*, 21(5) :313–348, october 1992.
9. I. Kodokula. *Data-centric compilation*. PhD thesis, Cornell University, 1993.
10. I. Kodokula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, june 1997.
11. D. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.
12. W. Li. *Compiling for NUMA parallel machines*. PhD thesis, Cornell University, 1993.
13. V. Loechner, B. Meister, and Ph. Clauss. Precise data locality optimization of nested loops. *Journal of Supercomputing*, 21(1) :37–76, january 2002.
14. K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4) :424–453, july 1996.
15. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5) :469–498, october 2000.
16. D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.
    La PolyLib est librement téléchargeable à l'adresse http ://icps.u-strasbg.fr/PolyLib.
17. M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, New York, june 1991.
18. M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.