Rapport de stage de première année de Master
Optimisation de cache d'instructions

Benoit Boissinot
Benoit.Boissinot@ens-lyon.fr
Université Lyon 1

sous la direction de

Fabrice Rastello
Fabrice.Rastello@ens-lyon.fr
E.N.S. Lyon

Éric Thierry
Eric.Thierry@ens-lyon.fr
E.N.S. Lyon

15 juin 2005

Contexte

Ce stage a été effectué du 8 avril au 30 juin 2005 dans l'équipe Inria Compsys à l'E.N.S. de Lyon. L'objectif de l'équipe Compsys est le développement de techniques d'optimisation pour les systèmes embarqués. Dans le cadre de ce stage, nous avons cherché une heuristique pour le placement des procédures minimisant les conflits. Une implémentation en C de cette heuristique a été implémentée. Par ailleurs des résultats de complexité sur la minimisation des conflits dans le cache d'instructions ont été trouvées.

1 Introduction

Idéalement, nous voudrions avoir beaucoup de mémoire dotée d'une vitesse d'accès très élevée. Cependant, les mémoires assez rapides pour le processeur sont très chères et ne peuvent pas être très grandes (registres). Pour faire la liaison entre la vitesse du processeur et la vitesse des mémoires à forte capacité (disques durs, CD-Roms, ...), nous avons besoin de mémoires faisant des compromis entre vitesse d'accès, taille et coût (mémoire cache puis mémoire vive). On parle de mémoire hiérarchique [7]. Le processeur n'a accès qu'aux registres : les informations dont il a besoin sont rapatriées depuis les mémoires plus lentes quand il en a besoin ; il y a donc tout intérêt à ce qu'une information utilisée fréquemment

Niveau	1	2	3	4
Nom	registre	cache	mémoire principale	stockage disque
Taille	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Temps d'accès (ns)	0.25-0.5	0.5-25	80-250	5000000
Bande passante (MB/sec)	20000-100000	5000-10000	1000-5000	20-150

TAB. 1 – Hiérarchie des mémoires. Les valeurs données sont des valeurs typiques pour 2001.

reste au niveau des mémoires rapides, il faut donc éviter qu'elle soit écrasée par d'autres informations.

On parle de défaut de cache (*cache miss*) lorsqu'un programme a besoin d'accéder à un bloc qui n'est pas dans le cache. Il faut alors le copier de la mémoire vers le cache, ce qui est plus coûteux qu'un accès direct au cache (*cache hit*).

Souvent, il y a deux caches différents, le cache de données et le cache d'instructions, l'un contient les données accédées par le programme et l'autre contient les instructions du programme. Dans notre cas nous nous intéresserons uniquement à l'optimisation du cache d'instructions.

Dans le cas du processeur ST200 de STMicroelectronics, le cache d'instructions est à *adressage direct* (*direct-mapped*), c'est-à-dire que si L est la taille du cache, les instructions situées à l'adresse mémoire i sont chargées dans le cache à l'adresse $\bar{i}^L = i \mod L$. Pour éviter les conflits, il faut donc éviter de placer aux mêmes adresses modulo L des instructions appelées de manière entrelacées (en alternance). De plus dans le cas de ce processeur particulier, le coût d'un défaut de cache est particulièrement élevé (de l'ordre de 150 cycles). Un paramètre dont il faut tenir compte lors de l'optimisation est l'expansion de code : le rapport entre la taille finale du programme et la somme des tailles des procédures. En effet s'autoriser à rajouter des trous entre procédures peut fournir un placement avec moins de conflits. La minimisation de l'expansion de code fait donc partie de la fonction objective de notre problème.

Dans le cas de systèmes embarqués, la taille de la mémoire est une donnée critique (notamment le coût du support).

Notre étude s'intéresse donc au placement des procédures afin de minimiser le nombre de conflits dans le cache. Cette optimisation s'effectue lors de la phase d'édition des liens, la granularité est importante comparée à une optimisation au niveau génération de code, dans notre cas, on manipule des procédures et non des instructions.

Le travail effectué au cours du stage a consisté en une implémentation en C d'un algorithme de placement de code. Le programme que nous avons développé à une complexité largement inférieur au programme précédemment utilisé. Grâce à l'utilisation de structures de données avancées (tableaux de bits, partitionnement par arbres, graphe avec tables de hachage, ...), nous avons obtenu un speedup de 3 sur l'exécution de l'algorithme. De plus, nous avons trouvé des résultats de complexité (non-approximabilité et NP-complétude) sur

les problèmes étudiés.

Dans une première partie nous reprendrons les algorithmes de placement de code existants, ensuite nous énoncerons un résultat de non-approximabilité que nous avons obtenu. Puis nous décrirons l'algorithme que nous avons implémenté.

2 État de l'art

Différentes heuristiques de placements des procédures ont déjà été proposées, elles sont toutes basées sur un algorithme glouton qui réduit un graphe dérivé de la trace des instructions utilisées par une exécution du programme. Ce graphe est une approximation des conflits entre procédures. Nous décrirons plus en détail les méthodes exposées par Pettis et Hansen [8], Gloy et al. [5] et Bouchez et al. [6].

2.1 Algorithme de Pettis et Hansen

L'algorithme de Pettis Hansen [8] utilise le graphe d'appel des procédures. Le graphe d'appel est un graphe construit à partir de la trace, où les sommets sont des procédures et les arêtes représentent les suites d'appels. Par exemple, si la procédure A est appelée par la procédure B dans la trace, alors on place une arête de B vers A, et on incrémente le poids de cette arête. Le graphe est élagué pour ne garder que les procédures les plus populaires.

L'algorithme consiste à prendre les arêtes par ordre de poids décroissant, et à fusionner les sommets reliant ces arêtes. Les fusions engendrent des chaînes de procédures (ou agrégats). Lors de la fusion, si deux procédures ne sont pas encore dans le même agrégat, on teste différentes possibilités de placement qui minimisent la distance entre ces procédures (utilisation du principe de localité). Si l'on note $\overline{A}$ la chaîne A où les procédures ont été inversées ($p_1, \ldots, p_n$ devient $p_n, \ldots, p_1$), les différentes positions testées sont AB,BA $\overline{A}B$ et $B\overline{A}$ ainsi que les combinaisons inverses ($\overline{AB}$, $\overline{BA}$,...).

2.2 Algorithme de Gloy et al.

L'algorithme de Gloy et al. [5] utilise un «graphe de relation temporelle» ou graphe de conflit (*Temporal Relationship Graph* ou *TRG*). Ce graphe est construit à partir d'une trace des instructions utilisées par l'exécution d'un programme. Intuitivement, en prenant comme granularité les procédures (en pratique celle-ci est plus faible), on peut le décrire de la façon suivante : les noeuds sont les procédures et, étant donnés deux noeuds P et Q, le poids de l'arête les joignant est le nombre de fois que deux occurrences de P sont séparées par une occurrence de Q dans la trace ou vice versa. Ce nombre est une approximation du nombre de défauts de cache qu'il y aurait pour P si les deux procédures avaient pour taille une ligne de cache et étaient placées au même endroit (si P n'a pas d'autres conflits) ; le *TRG* est donc un meilleur indicateur de conflits que le graphe d'appel utilisé par Pettis et Hansen.

L'algorithme de placement de code travaille sur un graphe dérivé du *TRG*, dans lequel chaque noeud contient une liste de couples dont les premiers éléments sont des procédures et les seconds un décalage relatif qui exprime la position modulaire de la procédure correspondante par rapport aux autres procédures du noeud. Initialement, chaque noeud contient un seul couple dont la procédure est celle du noeud correspondant dans le *TRG* et le décalage est 0. Puis l'algorithme recherche à chaque étape l'arête de poids maximal et fusionne les noeuds. Parmi tous les décalages possibles, on choisit celui qui provoque le moins de conflit. Une fois que deux noeuds sont fusionnés, leur décalage relatif (modulo la taille du cache) restera identique. Une fois que le graphe est entièrement réduit, chaque procédure possède un décalage modulo la taille cache. La deuxième phase de l'algorithme consiste à ordonnancer les procédures avec la contrainte que leur décalage modulaire ne change pas. Un algorithme glouton est donc effectué sur les procédures : on choisit une procédure de départ, puis tant qu'il reste des procédures non agrégées, on agrége à droite la procédure qui minimise le décalage entre la chaîne existante et elle-même pour former une nouvelle chaîne.

L'un des problèmes de cette approche est que la première phase de placement modulaire ne se préoccupe pas de l'expansion du code. Aussi dans la phase finale de l'algorithme, l'algorithme utilisé n'est pas optimal en terme d'expansion de code. Ainsi l'application de l'algorithme de Gloy et al. peut parfois quadrupler la taille du code du programme après optimisation !

2.3 Algorithme de Bouchez et al.

L'algorithme de Bouchez et al. [6] est une variante de l'algorithme de Gloy et al. Dans la première phase, les auteurs utilisent une estimation de l'expansion pour chaque ensemble de procédures afin de minimiser celle-ci et de prendre en compte le rapport du gain en terme de conflits par rapport à la perte au niveau de l'expansion.

Une fois que l'on a réduit le graphe de conflit, l'algorithme propose une méthode optimale pour placer les procédures en conservant également les décalages dans le cache. Cette méthode génère une expansion de code bien moins importante.

3 Complexité du problème de minimisation des conflits

Dans [6], le problème suivant est démontré comme étant NP-Complet.

Problème Min-Cache-Miss **:** Soit un taille de cache L fixé, $\mathcal{T}$ la trace des instructions utilisées par un programme, et k un entier, peut-on trouver une allocation de manière à ce que le nombre de défauts de cache lors de l'exécution de $\mathcal{T}$ soit inférieur à k ?

Nous avons démontré (voir annexe A) que ce problème, en plus d'être NP-complet, est également non-approximable à un rapport $N^{1-\epsilon}$ près, où ϵ est fixé et où N est la taille de la trace.

4 Une heuristique pour Min-Cache-Miss

L'algorithme de Gloy et al. génère une expansion de code très importante, nous avons développé une heuristique qui donne une expansion de code quasi-nulle[1]. Elle utilise un technique mixte entre Pettis et Hansen (placement contiguës des procédures) et Gloy et al. (utilisation du *TRG*, évaluation du coût de placement et choix du plus petit coût).

L'heuristique qui a été implémentée pour l'instant n'est pas l'algorithme final que nous souhaitons produire, elle nous permet d'évaluer quelles sont les options à envisager et quels degrés de liberté nous disposons.

4.1 Définitions

Chunk Un chunk est un partie de procédure de taille inférieure ou égale à une ligne de cache. Si $\|A\|$ est la taille de la procédure A, et s la taille d'une ligne de cache, alors la procédure sera constituée de $\lceil n = \|A\|/s \rceil$ chunks. Les chunks de A seront notés $A_0, \ldots, A_{n-1}$.

Graphe de conflit ou *TRG* Le graphe de conflit est un graphe construit à partir de la trace. Les sommets sont les chunks et les arêtes représentent le nombre de fois où deux occurrences d'un chunk A sont séparées par un chunk B dans la trace (et *vice versa*). Ce nombre est le nombre exact de défauts de cache qu'il y aurait si les deux chunks se retrouvaient sur la même ligne de cache. Le graphe de conflit est une représentation compressée et partielle de la trace.

Agrégat Un agrégat est un ensemble de procédure où le positionnement des unes par rapport aux autres est fixé. Le coût engendré par les conflits des procédures contenues dans l'agrégat ne dépend pas du décalage de la première procédure.

4.2 Algorithme de placement

Données du problème : Afin de simplifier l'explication, on omettra le fait que les procédures doivent respecter des contraintes d'alignement

- L : taille du cache d'instructions
- $I_1, I_2, \ldots I_n$: intervalles à placer ; chaque intervalle correspond à une procédure. $\|I_i\|$ est la longueur de l'intervalle I_i et $\mathit{offset}(I_i)$ est l'adresse de la première case mémoire du cache d'instruction utilisée par cet intervalle ; c'est à dire que quand la procédure i est chargée dans le cache, elle occupe les cases $\mathit{offset}(I_i)$, $\overline{\mathit{offset}(I_i)+1}^{L}, \ldots,$ $\overline{\mathit{end}(I_i)}^{L}$ du cache d'instruction (avec $\mathit{end}(I_i) = \overline{\mathit{offset}(I_i) + \|I_i\| - 1}^{L}$).

L'algorithme 1 utilise un graphe de conflit construit à partir de la trace :

[1]Par souci de clarté, nous tairons dans ce rapport certaines contraintes matérielles (alignement) qui complexifient le problème et rendent en général l'expansion de code nulle inatteignable.

– lignes 4 à 10, le graphe de travail (E_w, V_w est initialisé à partir du graphe de conflit (E_c, V_c). Les sommets de ce graphe sont des agrégats de procédures et les arêtes représentent une estimation du conflit possible entre deux procédures (somme du poids des arêtes entre deux agrégats ce qui représentent l'espérance du conflit entre les deux procédures).
– Ensuite, lignes 12 à 15, les noeuds sont fusionnés (algorithme 2). On choisit à chaque fois le couple de noeuds réuni par une arête avec le poids le plus fort. Lors de la fusion plusieurs politiques sont possibles, la première a été de tester les positions AB et BA et de choisir la moins coûteuse . Une variante de cette technique est de choisir les k plus grosses arêtes, de tester pour chaque couple les deux positions puis de prendre le plus grand ratio poids de l'arête sur coût de placement. Une autre variante est de tester tous les décalages pour lesquels il existe un ensemble de procédures qui sont des points isolés dans le graphe de travail qui remplissent ce décalage. L'algorithme 3 décrit comment construire l'ensemble des décalages possibles et l'algorithme 4 permet de retrouver les procédures à utiliser pour construire le décalage voulu.
– Finalement, lignes 17 à 19, lorsqu'il ne reste plus d'arêtes dans le graphes de travail les agrégats restants sont fusionnés deux par deux.

Algorithme 1 Algorithme glouton qui réduit le graphe de conflit.

```
PLACE_PROCEDURES ()
1   I : ensemble de procédures (intervalle)
2   (V_c, E_c) : Graphe de conflit
3   /* initialisation du graphe de travail */
4   V_w ← I
5   E_w ← ∅
6   for (a, b) ∈ E_c do
7      e ← (procedure(a), procedure(b))
8      E_w ← E_w ∪ e
9      poids(e)+=poids((a, b))
10  endfor
11  /* réduction des arêtes */
12  while E_w ≠ ∅ do
13     (a, b) = e ∈ E_w, poids(e) = max_{x∈E_w} poids(x)
14     FUSION(a, b)
15  endwhile
16  /* réduction des noeuds restants */
17  while V_w > 1 do
18     FUSION(a, b)
19  endwhile
```

L'algorithme 2 présente la fusion de deux sommets du graphes de travail. Le choix du placement des agrégats se fait ligne 2. Ensuite il s'agit de fusionner les noeuds A et B dans un nouveau sommet C et donc de modifier les voisins de A et B. Dans la section 4.3 nous décrirons les détails de l'implémentation notament l'utilisation de listes d'adjacence pour

diminuer la complexité.

Algorithme 2 Fusionne A et B dans un nouveau sommet C.

```
FUSION (A, B)
1   (V_w, E_w) : Graphe de travail
2   choisir la position de A par rapport à B
3   /* on enlève les sommets A et B */
4   V_w ← V_w/{A, B}
5   E_w ← E_w/{(A, B)}
6   /* on met à jour les voisins de A */
7   for (A, x) ∈ E_w do
8      E_w ← E_w ∪ {(C, x)}
9      poids(C, x)+=poids(A, x)
10     E_w ← E_w/{(A, x)}
11  endfor
12  /* on met à jour les voisins de B */
13  for (B, x) ∈ E_w do
14     E_w ← E_w ∪ {(C, x)}
15     poids(C, x)+=poids(B, x)
16     E_w ← E_w/{(B, x)}
17  endfor
18  /* on ajoute le sommet C */
19  V_w ← V_w ∪ C
```

4.2.1 Construction de décalages possibles à partir des agrégats isolés.

Soit L la taille du cache (en octets). On veut connaître les valeurs modulo L que l'on peut avoir à partir des agrégats isolés. On utilise donc de la programmation dynamique de dimension 2.

Les n agrégats isolés (non reliés à un autre noeud du graphe) sont stockés dans un tableau P.

`possible` est un tableau de taille L qui contient les valeurs possibles à chaque étape. Il est initialisé à `Faux` sauf pour 0 où l'on met `Vrai`.

`use` est une matrice de taille $n \times L$, on s'en sert pour retrouver quelles procédures on a utilisées à chaque étape pour construire un décalage donné.

L'algorithme consiste, pour chaque agrégat de P, à ajouter les valeurs que l'on peut atteindre en l'utilisant et à mettre à jour `use` pour indiquer que l'on a utilisé l'agrégat.

L'implémentation à partir d'un bitmap (tableau de bits) permet d'accélérer le calcul, en effet les lignes 8 à 11 devienne les instructions suivantes :

- Construire le bitmap `tmp`, qui est le bitmap `possible` ayant subi une rotation de $\|P[k]\|$.
- Assigner `tmp` à `use`$[k]$.
- `possible` devient le résultat du «ou» bit-à-bit avec `tmp`.

Toutes les opérations se font par groupe de 32 sur une architecture classique (32 bits) et les opérations booléennes sont les opérations bit-à-bit correspondantes.

La complexité de l'algorithme est en $O(L \times n)$, en effet on fait deux parcours de P, un pour construire les possibles et un autre pour retrouver les agrégats utilisés, en pratique (résultats avec `gprof`), la complexité est négligeable.

Algorithme 3

DECALAGE_POSSIBLE ()

$P[1 \dots n]$: tableaux des procédures qui sont des points isolés
L : taille du cache en octets
$possible[0 \dots L-1]$: tableau contenant les décalages possibles
$use[1 \dots n][0 \dots L-1]$: tableau pour retrouver les procédures utilisées pour chaque décalage
$possible[0] \leftarrow$ **true**
for $k \leftarrow 1 \dots n$ **do**
  for $i \leftarrow 0 \dots L-1$ **do**
    if $possible[i]$ **then**
      $possible[\overline{i + \|P[k]\|}^{L}] \leftarrow$ **true**
      $use[k][\overline{i + \|P[k]\|}^{L}] \leftarrow$ **true**
    endif
  endfor
endfor

Pour retrouver les agrégats à utiliser à partir d'un décalage donné il suffit de revenir en arrière dans le tableau des `use`. On commence par les derniers agrégats, si il y a `Vrai` dans le tableau (ligne correspondant à l'agrégat, position correspondant au décalage) alors on ajoute l'agrégat et on retranche $\|P[k]\|$ au décalage. Sinon on continue avec la procédure d'avant.

Algorithme 4

TROUVE_PROCEDURES ()

$P[1 \dots n]$: tableaux des procédures qui sont des points isolés
L : taille du cache en octets
s : taille du décalage que l'on souhaite obtenir
$use[1 \dots n][0 \dots L-1]$: tableau pour retrouver les procédures utilisées pour chaque décalage
$A \leftarrow \emptyset$
for $k \leftarrow n \dots 1$ **do**
  if $use[k][s]$ **then**
    $A \leftarrow A \cup P[k]$
    $s \leftarrow \overline{s - \|P[k]\|}^{L}$
  endif
endfor
return A

4.3 Complexité de l'algorithme

Décrivons l'analyse de la complexité de l'algorithme décrit au dessus : On décompose le problème en MAX (chercher l'arête de poids maximal), BEST_SHIFT (trouver le meilleur emplacement) et MERGE (fusionner les agrégats). On note m et n respectivement le nombre d'arêtes et de sommets du graphe de conflit et M et N le nombre d'arêtes et de sommets du graphe de travail.

- MAX : à chaque étape il faut connaître l'arête de poids max. Le graphe étant à priori quelconque, la complexité amortie en utilisant une structure de tas est de $O(M \log(M))$ (il y a M mises à jour du tas qui est de profondeur $\log(M)$).
- BEST_SHIFT : chaque fois que l'on a choisit l'arête à traiter, il faut trouver le meilleur placement respectif des deux agrégats A et B. Tout d'abord, les décalages des chunks sont stockés dans une structure de*partition* [3], chaque agrégat forme un arbre où le premier chunk est le représentant. Lorsqu'on cherche le décalage d'un chunk, on met à jour l'arbre (on l'aplatit). Lors des fusions des agrégats, il suffit de reparenter l'un des arbres (opération en $O(1)$). Comme indiqué dans [3], la complexité totale du maintien des décalages est en $O(M + m \cdot (1 + \log_{2+m/M} M))$. Si les deux agrégats ont pour taille $\|A\|$ et $\|B\|$, pour chaque décalage possible (L), on calcule le coût de ce décalage ($\|A\| \times \|B\|$). Cela fait une complexité amortie de $O(n^2 L)$. On peut faire mieux : pour chaque arête entre A et B ($\|A\| \times \|B\|$) on regarde pour quel décalage elle serait comptée. Puis on parcourt l'ensemble des décalages accessibles (tels qu'il existe un ensemble de procédures qui comblent le décalage, voir les algorithmes 3 et 4) et on prend celui de poids minimum. La complexité amortie est de $O(m)$.
- MERGE : une fois le meilleur décalage trouvé, il faut fusionner l'ensemble des arêtes d'adjacence ($M_a + M_b$) des deux agrégats (et sommer les poids). Avec des listes d'adjacence triées, on peut le faire en $M_a + M_b$. La complexité amortie est de $O(N^2)$ où N est le nombre de sommet initial du graphe de travail.

On obtient donc finalement, (M est en $O(m)$ et N est en $O(n)$) une complexité de $O(n^2 + m \log m)$.

5 Autres travaux

Une autre méthode que nous avons exploré est la méthode suivante basée sur du *diviser pour régner*. Si on a un ensemble de procédures tels que la somme des tailles des procédures est inférieure à la taille du cache, alors on peut les placer de façon à ce qu'il n'y ait pas de conflit entre elles (placement contiguë). L'idée est donc de partitionner les procédures en sous-ensembles tenant dans le cache, tout en minimisant les conflits entre ces ensembles [1]. Ensuite on fusionne les parties.

Le problème du partitionnement du graphe en sous-ensemble de taille L minimisant les poids entre parties est connu comme étant NP-complet [2]. Nous avons démontré, que la fusion des sous-ensembles est également NP-complet.

En effet le problème suivant, qui est équivalent au problème lors de la fusion est NP-

complet.

Problème K-Merge-Cost **:** Soit un ensemble de procédures $P = \{p_1, \ldots, p_n\}$ telles que $\sum_{p \in P} \|p\| = L$, L étant la taille du cache. Pour tout $p \in P$, il existe une fonction de coût $\widehat{C}_p : [\![0, L - size(p)]\!] \to \mathbb{N}$ telle que $\widehat{C}_p(\lambda)$ qui représente le coût (nombre de défaut de cache) de placer la procédure p en $[\![\lambda, \lambda - \|p\|[\![$. Existe-t-il une partition de $[\![0, L[\![$ à l'aide des n procédures tel que le coût total soit inférieur à k ?

La preuve de la NP-complétude ainsi que de la non-approximabilité à $2W$ (W plus grande valeur prise par les fonctions de coût) près sont énoncés dans l'annexe B.

6 Conclusion

Nous avons donc implémenté une variante de l'algorithme de Gloy et al. ayant une complexité acceptable pour une utilisation dans un compilateur. Le temps d'éxécution de l'algorithme est passé de 30 secondes à 10 secondes sur un éxécutable type (`mpeg2decode.exe`). L'implémentation de cette algorithme nous a permis de tester quelques pistes pour la conception d'un algorithme efficace d'optimisation du cache d'instruction.

Les résultats de complexité que nous avons obtenu (NP-complétude et non-approximabilité), mettent en valeur le fait que quelque soit l'algorithme que nous produirons, nous ne serons jamais assuré de la qualité des résultats obtenables.

Une autre piste que nous avons explorée est celle de l'affinité entre procédures. En effet, lors d'un défaut de cache, le processeur charge une ligne entière, de plus il y a un phénomène de *pré-chargement* (*prefetch*) qui fait que les lignes suivantes sont également chargées. On peut donc jouer sur ce fait et placer les procédures dans le même ordre que l'ordre d'appel.

A Preuve de non-approximabilité de Min-Cache-Miss

Le but de cette annexe est de prouver le théorème 1.

Théorème 1 *Pour tout $\epsilon > 0$, approximer* Min-Cache-Miss *avec un facteur de $O(n^{1-\epsilon})$ où n est la taille de la trace est NP-dur.*

Preuve : On réduit le problème à partir de Graph K-Colorability [3]. L'idée est que étant donné un graphe G et k couleurs, on peut construire un programme avec autant de procédures que de noeuds dans G et tel que si deux noeuds sont voisins dans G alors le poids entre les deux procédures dans le graphe de conflit est fort. Si on considère une mémoire cache avec $k + 1$ lignes de cache (chaque procédure étant de taille une ligne, et on a une autre ligne pour la procédure principale), alors un placement optimal colorie le graphe G avec k couleurs.

Considérons le graphe $G = (V, E)$ avec $V = \{x_1, \ldots, x_n\}$. Soit $G' = (V', E')$ avec $V' = V \cup \{p\}$ et $E' = E \cup (\cup_{v \in V} \{p, v\})$.

Soit P le programme suivant :

```
P ()
1   for i ← 1 ... n − 1 do
2     for j ← i + 1 ... n do
3       if (x_i, x_j) ∈ E then
4         for tmp ← 1 ... W do
5           X_i()
6           X_j()
7         endfor
8       endif
9     endfor
10  endfor
```

$X_1, \ldots, X_n$ sont des procédures de taille 1. W est un nombre suffisamment grand, on prendra $W = \text{card } E^l$ où l sera déterminé plus tard (card E est le nombre d'arêtes dans G).

La trace d'exécution de P est :

$$\mathcal{T} = P \prod_{(x_i, x_j) \in E} (X_i P X_j P)^W$$

Pour tout $W = \text{card } E^l$, on notera $\mathcal{T}_l$ la trace générée par cette valeur de W et $Opt(\mathcal{T}_l)$ le nombre optimal de défaut de cache pour $\mathcal{T}_l$. Notons que la taille de la trace $\mathcal{T}_l$ est égale à $4W \cdot \text{card}(E) + 1 = 4 \cdot \text{card}(E)^{l+1} + 1$.

Tout d'abord, on doit montrera que la taille de l'instance de Min-Cache-Miss construite à partir de G est polynomiale en la taille de $G = \text{card}(E) \cdot \log(\text{card}(V))$. C'est le cas si on choisit l ne dépendant que de ϵ. On peut également noter que G' est L-coloriable si et seulement si G est $(L-1)$-coloriable.

A présent le lemme 1 nous assure que décider si $Opt(\mathcal{T}_l) \leq 2 \cdot \text{card}(E) + 1$ ou $Opt(\mathcal{T}_l) \geq 2 \cdot \text{card}(E)^l$ est NP-complet (car cela nous permettrait de décider si G est $L-1$-coloriable ce qui est NP-complet pour $L \geq 4$).

Supposons qu'il existe un algorithme polynomial qui approxime $Opt(\mathcal{T}_l)$ à un facteur $(2 \cdot \text{card}(E)^l)/(2 \cdot \text{card}(E) + 1)$, on peut alors determiner si $Opt(\mathcal{T}_l) \leq 2 \cdot \text{card}(E) + 1$ ou si $Opt(\mathcal{T}_l) \geq 2 \cdot \text{card}(E)^l$.

Si l'on peut approximer à un facteur $(2 \cdot \text{card}(E)^l)/(3 \cdot \text{card}(E))$, alors on peut approximer à un facteur $(2 \cdot \text{card}(E)^l)/(2 \cdot \text{card}(E) + 1)$.

Donc approximer $Opt(\mathcal{T}_l)$ à un facteur $f(N) = (2 \cdot \text{card}(E)^l)/(3 \cdot \text{card}(E))$ est NP-dur (avec N la taille de $\mathcal{T}_l$). Comme $N = 4 \cdot \text{card}(E)^{l+1} + 1$, on a $f(N) = \frac{2}{3}(\frac{N-1}{4})^{\frac{l-1}{l+1}}$. On peut prendre une valeur arbitraire pour l : pour $\epsilon > 0$ fixé, on prend l tel que $\frac{l-1}{l+1} \geq 1 - \epsilon$. Alors $f(N) \geq \frac{2}{3}(\frac{N-1}{4})^{1-\epsilon} \geq \frac{2}{3}(\frac{N}{8})^{1-\epsilon}$.

Finalement, pour tout $\epsilon > 0$, approximer MIN-CACHE-MISS à un facteur $\frac{2}{3\times 8^{1-\epsilon}} N^{1-\epsilon}$ est NP-dur.

Lemme 1 *G' est L-coloriable si et seulement si $Opt(\mathcal{T}_l) \leq 2 \cdot card(E) + 1$. G' n'est pas L-coloriable si et seulement si $Opt(\mathcal{T}_l) \geq 2 \cdot card(E)^l$.*

Preuve D'abord prouvons que si G' est L-coloriable, alors on peut construire une allocation qui coûte moins que $2 \cdot \text{card}(E) + 1$.

Soit $c : V \to \{0, \ldots, L-1\}$ une coloration de G'. Notre allocation est directement dérivée de la coloration : $\mathit{offset}(P) = c(p)$ et pour tout i, $\mathit{offset}(X_i) = c(x_i)$.

A présent, comme p est connecté à tous les noeuds du graphe, aucune autre procédure que P n'est placée sur la première ligne du cache. De plus, comme deux sommets voisins ont une couleur différente, il n'y pas de conflit dans la boucle for indexée par *tmp*. Donc les seuls défauts de cache possibles sont ceux où on change d'arêtes dans le programme. Pour chaque transition, il y a au plus deux défauts de cache (les deux procédures n'étaient pas chargées). On ajoute un pour le chargement initial de P.

Il y a donc moins que $2 \cdot \text{card}(E) + 1$ défauts de cache.

Ensuite si G' n'est pas L-coloriable. Quelque soit l'allocation choisie, pour la L-coloration correspondante ($c : V \to \mathit{offset}(P)$ tel que $c(p) = \mathit{offset}(P)$ et $x_i \to \mathit{offset}(X_i)$), il existe $(x_i, y) \in E$ avec $y = p$ ou $y = x_j$, tel que $c(x_i) = c(y)$.

Soit $Y = X_j$ si $y = x_j$, $Y = P$ sinon.

Comme x_i et y ont la même couleur, $\mathit{offset}(X_i) = \mathit{offset}(Y)$ et donc la trace $(X_i P X_j P)^W$ va créer au moins $2 \times W - 2$ défauts de cache.

On peut ajouter un pour le chargement initial de P ainsi que $\text{card}(V)$ pour charger tous les X_i la première fois.

Il y a donc plus de $2W = 2 \times \text{card}(E)^l$ défauts de cache.

B Preuve de non-approximabilité de K-MERGE-COST

Le but de cette annexe est de prouver le théorème 2.

Théorème 2 *Approximer* K-MERGE-COST *avec un facteur $2W$, où W est la plus grande valeur prise par les fonctions de coût, est NP-dur.*

Preuve : On réduit le problème à partir de 4-PARTITION qui est fortement NP-complet [4].

Rappelons la définition du problème de 4-PARTITION dans sa version forte. L'instance consiste en $n = 4m$ nombres $s_1, \ldots, s_n$ tels que $\sum_{i=1}^{n} s_i = m \cdot B$ et pour tout i, $B/5 < s_i < B/3$, B étant polynomial en n (plus précisement $B \leq cn^3$ pour une constante c). La question est existe-t-il $S_1, \ldots, S_m$ partition de $\{s_1, \ldots, s_n\}$ telle que pour tout i, $\sum_{s \in S_i} s = B$. On remarquera que nécessairement $\text{card}(S_i) = 4$.

Considérons un cache de taille $L = m \cdot (B+1)$. On prend n procédures qui ne conflictent pas (procédures de coût nulles) $p_1, \ldots, p_n$ telles que $\|p_i\| = s_i$ et m procédures $p'_1, \ldots, p'_m$

telles que $\|p'_i\| = 1$ et pour tout i la fonction de coût est nulle en $(B+1)\cdot i$ et égale à W ailleurs ($\widehat{C}_{p'_i}((B+1)\cdot i) = 0$, et $\widehat{C}_{p'_i}(\lambda) = W$ si $\lambda \neq (B+1)\cdot i$).

La taille de l'instance est de $O(L\cdot(n+m) + m\log W + m + n + L + n\log(\max s_i))$ (taille des fonctions de coût, plus taille des procédures et du cache, plus taille des longueur des procédures). La taille est donc en $O(n^2\cdot B)$ (car $\max s_i \leq B$), donc polynomiale en la taille de l'instance de départ ($\Theta(n\log B)$).

Avec cette construction, clairmenent, il existe une partition de $p_1,\ldots,p_n,p'_1,\ldots,p'_m$ de coût total égal à zéro si et seulement si il existe une bonne partition pour l'instance de 4-Partition. K-Merge-Cost est donc NP-dur.

Soit *Opt* le nombre minimal de conflits que l'on peut obtenir à partir du problème. Supposons qu'il existe un algorithme polynomial qui approxime *Opt* à un facteur $2W$. On peut alors déterminer si $Opt > 0$ ou si $Opt = 0$ (s'il y a un conflit, le coût total approximé est supérieur à W, sinon il est inférieur). Finalement pour tout $W \geq 1$, approximer K-Merge-Cost à un facteur W est NP-dur.

Références

[1] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning : a survey. *Integr. VLSI J.*, 19(1-2) :1–81, 1995.

[2] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *SPAA '04 : Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124, New York, NY, USA, 2004. ACM Press.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[4] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[5] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 303–313, Los Alamitos, December 1–3 1997. IEEE Computer Society.

[6] Christophe Guillon, Fabrice Rastello, Thierry Bidault, and Florent Bouchez. Procedure placement using temporal-ordering information : dealing with code size expansion. *Journal of Embedded Computing*, 2004. to be published.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach (3rd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[8] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6) :16–27, June 1990.

Table des matières