Heuristiques d'utilisation du cache secondaire

Rapport de stage de DEA
Fevrier - Juin 2004

Thomas PIQUET
Sous la direction d'André SEZNEC (IRISA, projet CAPS)

IRISA

IFSIC
INSTITUT DE FORMATION SUPÉRIEURE en INFORMATIQUE et COMMUNICATION

Table des matières

Table des figures

Introduction

Actuellement, la fréquence des processeurs ne cesse d'augmenter. Parallèlement, les performances de la mémoire n'ont pas connu une telle croissance. Ceci a fait apparaître une augmentation du temps d'accès relatif à la mémoire. Afin de réduire ces temps d'accès, la hiérarchie mémoire a été introduite. Des mémoires sont insérées entre la mémoire principale du calculateur et le processeur. Ces mémoires appelées mémoires caches sont de taille inférieure à la mémoire principale et ont un temps d'accès plus faible. La raison principale de l'efficacité des mémoires caches est simple : un programme n'utilise pas à tout instant la totalité de ses données. Donc en réalisant une "sélection" des données en fonction de leur utilisation par le programme, on peut améliorer la vitesse d'éxécution de celui-ci.

Les mémoires caches utilisent de manière simple les propriétés de localité temporelle et spatiale qu'exhibent la plupart des applications. La même structure de cache gère toutes les données qu'elles exhibent de la localité temporelle ou de la localité spatiale. De plus, aucune trace de l'utilisation des données n'est gardée en mémoire. C'est à partir de ces deux points qu'ont été menées des études sur les techniques de "prefetch" [CB94, CB92, FP92] et sur les caches multi-latéraux [GAV95, TRS$^+$, RD96].

Dans cette étude, nous nous intéressons à une nouvelle dimension : la "pollution" des caches ou plus exactement de la hiérarchie mémoire. On peut qualifier une donnée de polluante pour le cache L2 si cette donnée est amenée dans ce cache puis éjectée avant sa réutilisation par le cache L1. Il est possible que cette donnée ait éjecté du cache L2 une donnée présentant une localité temporelle plus forte. En réalisant des études sur les "working sets" (ensembles de travail) et sur la localité temporelle des données dans un programme, on va essayer de caractériser au mieux l'utilisation des données d'un programme. Cette caratérisation a pour but de sélectionner les données pouvant polluer le cache et ainsi éviter l'éjection de données utiles au profit de données "polluantes".
La figure 1 montre comment on souhaite exploiter la propriété des working sets. Dans cet exemple, les working sets WS0 et WS1 tiennent entièrement dans le cache L2. Le processeur a maintenant besoin d'une donnée appartenant au working set WS2. La donnée est alors mise en cache L1 sans la faire passer par le cache L2 (bypass). Les données appartenant au working

set WS2 ont une localité temporelle plus faible que celles appartenant à WS0 et WS1. C'est cette raison qui motive le bypass du cache L2. Dans un fonctionnement de cache "classique", il est possible que cette donnée ne soit réutilisée qu'après son éviction des deux caches. Elle est stockée inutilement dans le cache L2. Ce qui fait que cette donnée a éjecté du cache L2 une donnée appartenant à WS0 ou WS1 (exhibant une localité temporelle plus forte), elle "pollue" le cache L2. Il est préférable de faire résider le plus près possible du processeur les données exhibant une localité temporelle forte. Donc en réalisant ce bypass on espère garder dans une mémoire proche du processeur (cache L2) les données ayant une localité temporelle forte et de ce fait augmenter la vitesse d'exécution des programmes.

Ce rapport est divisé en deux. Dans la première partie, l'étude bibliographique, nous allons voir comment est réalisée une hiérarchie mémoire, le principe de la localité et les différentes implémentations de mémoires caches. Puis nous allons nous consacrer à la caractérisation de l'utilisation des données, en voyant tout d'abord la réutilisation de l'information passée, une étude sur les working sets et une étude de la localité temporelle dans un programme. La seconde partie est le rapport sur les travaux et études réalisés durant le stage. Nous allons d'abord essayer de vérifier l'existence des working sets puis de quantifier la pollution du cache L2. Ensuite nous allons tenter de caractériser ces données polluantes d'une manière un peu plus précise.

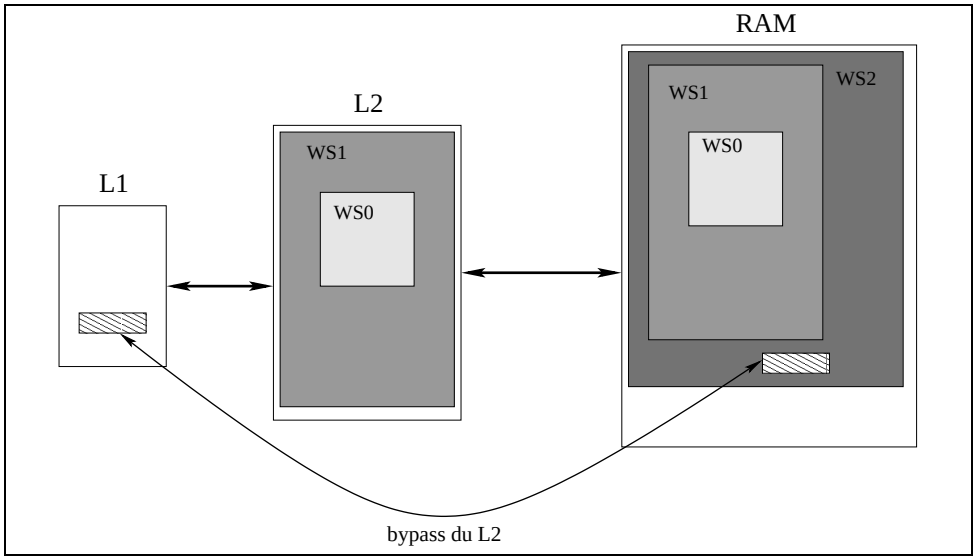

FIG. 1 – Principe de fonctionnement.

Première partie

Bibliographie

Chapitre 1

La hiérarchie mémoire

La nécessité d'avoir une hiérarchie mémoire tient au fait que les programmeurs désireraient une quantité de mémoire "infinie" et rapide d'accès. C'est de ce constat que la hiérarchisation de la mémoire est apparue. Du point de vue technologique, il n'est pas possible de réaliser une mémoire de grande capacité avec un temps d'accès de l'ordre d'un cycle processeur. La localité spatiale et temporelle des applications expliquent le succès de l'utilisation des hiérarchies mémoires.

1.1 La localité

La localité des données accédées apparaît sous deux formes principales : la localité temporelle, et la localité spatiale.

1.1.1 La localité temporelle

Une variable qui exhibe de la localité temporelle est une variable qui est accédée de manière fréquente durant une section de l'exécution d'un programme.
Exemple de code C :

```
for(i=0 ; i<10000 ; i++){
   s = s + a[i] ;
}
```

Ici `s` est une variable montrant une localité temporelle forte : elle est accédée pour chaque opération de cette itération. Il est donc intéressant d'essayer de minimiser le temps d'accès à cette variable mémoire.

On peut essayer d'organiser les variables en différents "niveaux" de localité temporelle. C'est-à-dire que certaines variables ont une utilisation plus intensive que d'autres.
Exemple de code C :

```
for(i=0 ; i<100 ; i++){
   k = k + b[i] ;
   for(j=0 ; j<10000 ; j++){
      s = s + a[i+j] ;
   }
}
```

Dans cet exemple, la variable `k` est utilisée 100 fois, alors que la variable `s` est utilisée 1000000 fois. Il est donc préférable de faire résider la variable `s` très près du processeur. En cas de conflit (plus de place disponible), il est plus judicieux de laisser la variable `s` plutôt que de l'éjecter au profit de la variable `k`.

1.1.2 La localité spatiale

La localité spatiale concerne principalement un groupe de variables ayant des adresses proches, tel un tableau. Un programme exhibant ce type de localité est un programme qui accède à ce tableau de manière régulière et dans un laps de temps assez court.
Exemple de code C :

```
for(i=0 ; i<10000 ; i++){
   a[i] = a[i] + b[i];
}
```

Dans cet exemple, les tableaux a et b exhibent de la localité spatiale : tous les éléments de chaque tableau sont accédés les uns à la suite des autres. On peut donc rapatrier les variables par blocs dans la mémoire cache. Ces blocs sont des zones de mémoire continue.

1.2 Implémentation

L'implémentation d'une hiérarchie mémoire peut être étudiée soit d'un point de vue technologique, avec la vitesse d'accès mémoire, soit d'un point de vue purement architectural, avec les différents principes de gestion des niveaux de mémoire. Ici, on va se concentrer sur les différentes manières de réaliser ces caches.

1.2.1 La vitesse d'accès aux mémoires

Comme le montre la figure 1.1, la vitesse d'accès à une mémoire est étroitement liée à sa taille. Cela conduit à une réalisation de hiérarchie où des

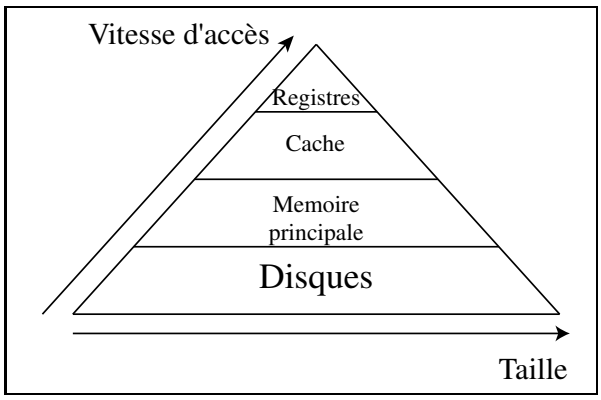

FIG. 1.1 – Rapport entre la vitesse et la taille d'une mémoire

mémoires de tailles et de vitesses différentes communiquent afin de minimiser les temps de latence et de fournir une mémoire de taille importante.

C'est entre les registres et la mémoire principale (RAM) que se trouve la mémoire cache. En fonction du type de technologie cette mémoire peut se trouver directement sur la puce du microprocesseur ou à côté. Actuellement, les mémoires caches sont elles-même composées de plusieurs niveaux. Très souvent les deux premiers niveaux se situent sur la même puce que le microprocesseur et éventuellement un troisième niveau vient se positionner entre le composant du microprocesseur et la RAM. Les niveaux de caches ont des capacités et des temps de latence croissants. De plus, le premier niveau de cache est scindé en deux parties : le cache d'instruction et le cache de données. Comme montré dans la figure 1.2.

1.2.2 Les différents types de cache

Pour masquer les temps de latence, lorsqu'une donnée est accédée on vérifie sa présence dans le premier niveau de cache. Si la donnée est présente, elle est directement fournie au processeur, on parle alors d'un *hit*. Dans le cas contraire, on vérifie si la donnée est présente dans le niveau de mémoire supérieur, dans ce cas on parle d'un *miss*. Lorsqu'un *miss* apparaît, la donnée est rapatriée du niveau de mémoire supérieur. Il existe trois types de miss (classification donnée dans [HS89]) :

Miss de "coldstart" : lorsque la mémoire cache est vide, aucune donnée n'est présente en mémoire.

Miss de capacité : lorsque la mémoire cache est pleine et que la donnée n'est pas présente en mémoire.

Miss de conflit : lorsque deux données doivent cohabiter dans la même zone mémoire du cache.

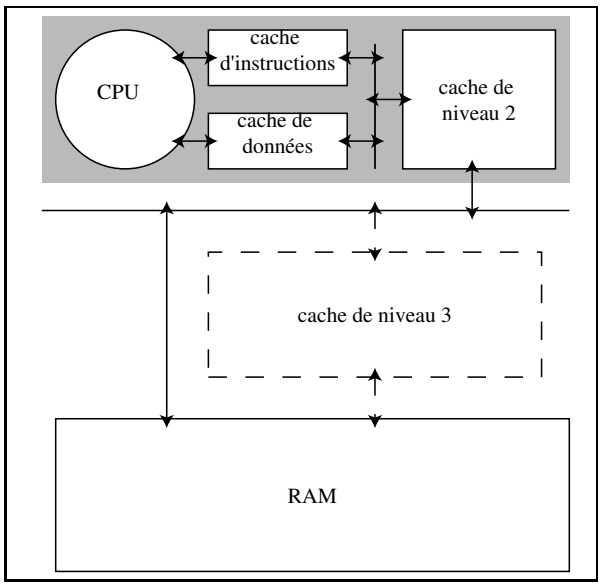

FIG. 1.2 – Exemple d'une hiérarchie mémoire actuelle

A partir de cela, on peut en déduire différentes stratégies pour organiser un niveau de mémoire cache. Principalement deux paramètres interviennent :

1. L'associativité de la mémoire : ce paramètre peut permettre de réduire les miss de conflit. Il exite différents types d'associativité pour les caches : totalement associatifs, à n-voies associatifs et à correspondance directe. La figure 1.3 montre ces différentes organisations de cache.
2. La politique de remplacement : lorsqu'un miss de capacité intervient, une ligne doit être éjectée du cache. On doit choisir quel banc va éjecter sa ligne. Couramment, un banc est désigné au hasard ou on sélectionne le banc qui a la ligne la plus ancienne (LRU).

D'autres paramètres comme la taille d'une ligne et la taille totale du cache sont aussi à prendre en compte. Cependant, rien ne peut permettre de réduire les miss de "coldstart".

Plus de détails sur la réalisation d'une hiérarchie mémoire peuvent être trouvés dans [HP90], au chapitre "Réalisation d'une hiérarchie mémoire".

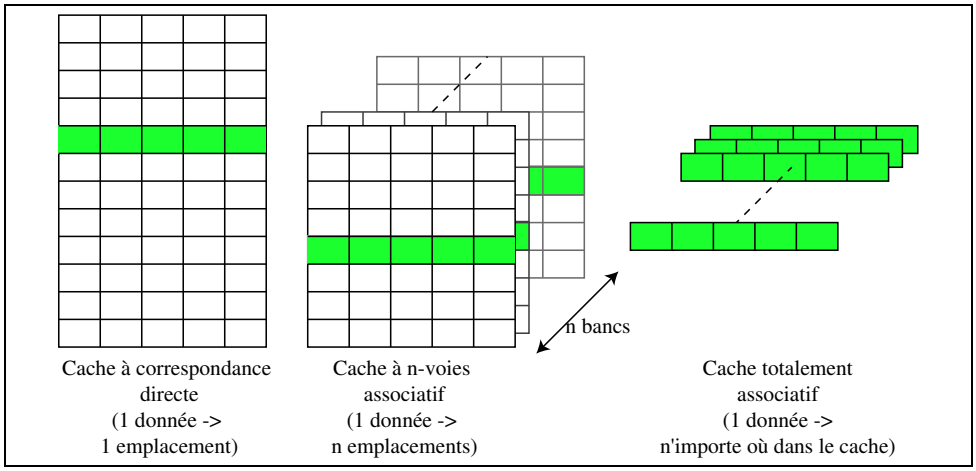

FIG. 1.3 – Les différentes organisations de cache

Chapitre 2

Caractérisation de l'utilisation des données

Les mémoires caches sont efficaces car l'utilisation des données dans un programme n'est pas aléatoire dans le temps et l'espace (d'adressage). On peut classifier les variables suivant qu'elles exhibent de la localité temporelle, spatiale. Ce sont ces catégories qui dirigent la réalisation de caches efficaces. C'est pourquoi, il est important de bien caractériser l'utilisation des données dans un programme. Nous allons d'abord voir comment sont traités ces biais d'une manière architecturale, ensuite nous verrons comment on peut qualifier les "working sets", puis nous appronfondirons le principe de la localité temporelle afin d'en tirer un modèle.

2.1 Réutilisation de l'information passée

C'est en observant le passé des données que l'on peut essayer d'en déterminer un comportement. C'est-à-dire que l'on veut déterminer soit le moment où ces données seront nécessaires, soit le type de localité exhibée par la variable. C'est sur ces deux points que s'appuient les deux concepts architecturaux suivants.

2.1.1 Les techniques de prefetch

Le prefetch (ou préchargement) est une méthode qui consiste à aller chercher les données en mémoire avant que le processeur n'en ait besoin. Il existe deux manières pour mettre en œuvre le prefetch :

Méthode logicielle : elle consiste à réaliser une étude préalable du code au moment de la compilation et à générer des instructions de prefetch à l'intention de la mémoire cache. Cette technique montre de bonnes performances pour des accès très réguliers à la mémoire, cependant certaines informations ne peuvent pas être prises en compte au moment

de la compilation. Cette méthode est dite *statique*. Une étude complète d'un algorithme réalisant l'insertion d'instructions de prefetch peut être trouvée dans [MLG92].

Méthode matérielle : il s'agit d'une approche *dynamique*. C'est-à-dire que les informations sont "captées" au moment de l'exécution du programme. L'avantage de cette technique est que le code généré n'est pas modifié par le compilateur ce qui garantie une plus grande compatibilité. En revanche, le coût matériel supplémentaire n'est pas négligeable et des données inutiles peuvent être préchargées.

Ici nous allons principalement nous concentrer sur la méthode matérielle. La méthode la plus simple pour réaliser du prefetch, consiste, lors d'un miss, à aller chercher en mémoire la ligne concernée par le miss ainsi que la ligne suivante. Dans ce cas de prefetch, on profite principalement de la localité spatiale pour des accès mémoires ayant un *pas* faible.

`for(i=0 ; i<10000 ; i++){`
`  s = s + a[i] ;`
`}`
	`for(i=0 ; i<10000 ; i++){`
	`  s = s + a[i*N] ;`
	`}`

exemple d'accès à un tableau avec un *pas* unitaire.	exemple d'accès à un tableau avec un *pas* de N.

Cet exemple montre ce qu'est un *pas*. Dans le cas où le pas est important (par exemple lorsque N est supérieur à la taille d'une ligne de cache), la technique de prefetch citée ci-dessus est assez inefficace. Premièrement parce que la ligne de cache préchargée ne sera pas utilisée et qu'elle peut éjecter du cache une donnée plus utile. C'est pour cela qu'a été introduit le prefetch dirigé par le pas présenté dans [FP92, CB94, CB92]. L'approche présentée dans [CB94, CB92] est un mécanisme basé sur la prédiction du flot d'instructions alors que dans [FP92], le choix de la donnée à précharger est fait lors d'un accès mémoire.

La technique de prefetch présentée dans [FP92] est basée sur une table de prédiction du pas *SPT* (Stride Prediction Table). La figure 2.1 montre l'organisation de cette table. Cette table est accédée lorsqu'une instruction réalise un accès à la mémoire. Les commandes de prefetch peuvent être réalisées dans trois cas : lorsque l'accès mémoire génère un hit, un miss, ou dans les deux cas. Cette table est constituée de trois champs :

1. Une adresse d'instruction utilisée pour repérer une entrée dans la table.
2. L'adresse de la donnée chargée précédemment (associée à l'adresse d'instruction).
3. Un bit de validation utilisé pour déterminer si le prefetch doit être réalisé.

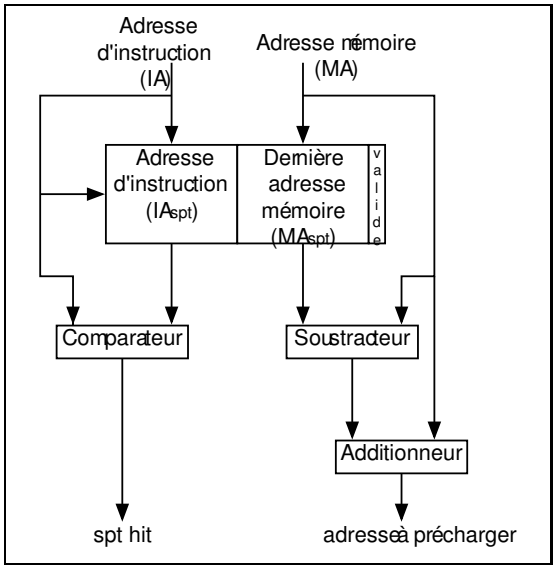

FIG. 2.1 – Schéma de la table de prédiction de prefetch présenté dans [FP92]

Le fonctionnement du mécanisme est le suivant : soit IA l'adresse de l'instruction réalisant une référence en mémoire, et MA l'adresse référencée. Si une entrée de la table, IA_{spt}, correspondant à IA est trouvée, on vérifie la validité de MA_{spt} (valeur associée à IA_{spt}). Le pas est calculé par $MA - MA_{spt}$ puis l'adresse à précharger par $MA + pas$. Le prefetch est réalisé seulement lorsque le pas est strictement positif.

Dans [CB94, CB92] la technique de préfetch est une version améliorée de [FP92]. La figure 2.2 montre l'organisation de l'unité qui réalise le prefetch. Comme on le voit, le mécanisme utilise un compteur ordinal spéculatif (LA-PC) qui se sert d'une table de prédictions de branchements. Ce compteur est maintenu à jour de la même manière qu'un compteur utilisant un prédicteur de branchement. L'autre différence de cette technique de prefetch réside dans la *Reference Prediction Table* (appelée SPT précédemment). Cette table contient maintenant quatre entrées :

1. Une adresse d'instruction utilisée pour repérer une entrée dans la table.
2. L'adresse de la donnée chargée précédemment (associée à l'adresse d'instruction).
3. Le pas mis à jour à chaque accès.
4. Un compteur 2 bits utilisé pour déterminer si le prefetch doit être réalisé.

La présence d'un compteur 2 bits et la sauvegarde du pas permettent d'éviter les prefetch inutiles. Par exemple, pour les accès aléatoires à un tableau et pour l'incrémentation de l'index le plus extérieur d'un nid de

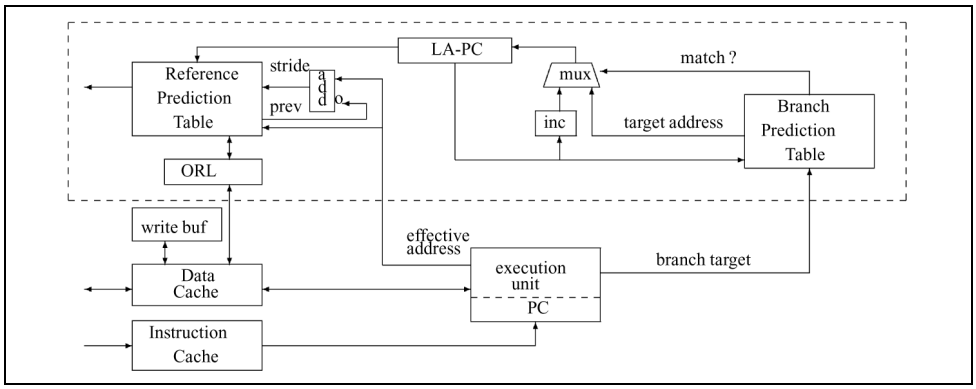

FIG. 2.2 – Schéma de l'unité de support de cache présenté dans [CB94, CB92]

boucles.

2.1.2 Les caches multi-latéraux

Comme nous l'avons vu précédemment, les différents types de localité sont généralement exploités de la même manière à l'intérieur d'un cache. Cela est principalement dû à la structure du cache, c'est-à-dire que c'est le même cache qui gère les données qui exhibent de la localité temporelle ou de la localité spatiale. C'est sur ce constat que se basent les travaux présentés dans [GAV95, TRS+, RD96].

La technique présentée dans [GAV95] sépare le cache de données en deux parties :

Le cache temporel : mémoire ayant les même caractéristiques qu'un cache classique mais possédant des lignes de petite taille (quelques octets).

Le cache spatial : identique au cache temporel mais possédant de grandes lignes de cache (quelques dizaines d'octets).

La détermination du type de localité de la donnée est faite de manière dynamique. Le mécanisme réalisant la prédiction du type de localité est basé sur des techniques de prefetch comme vu précédemment ([CB92, FP92] par exemple). Le prédicteur peut donner trois types de localités : spatiale, temporelle et aucune. Lorsqu'un miss apparaît, la table de prédiction est inerrogée, puis la donnée est mise dans le cache correspondant à la prédiction : temporel, dans le cache temporel ; spatiale, dans le cache spatial ; aucun, la donnée n'est pas mise en mémoire cache. C'est l'étude de l'évolution du pas d'une instruction réalisant une référence qui permet la détermination du type de localité. Lorsque le pas est nul, la localité choisie est temporelle. Pour un pas de taille fixe, il s'agit d'une localité spatiale et pour un pas variable, aucune localité n'est prédite.

Il existe différentes façons de gérer les caches multi-latéraux. L'architecture générale d'une mémoire de ce type est présentée dans la figure 2.3. Sur cette figure, il est à noter que les caches A et B sont deux caches de nature différente (taille de ligne, nombre de lignes, associativité...). De plus une unité de détection (DU) est utilisée pour déterminer dans quel cache doit être placée la donnée. L'unité de détection est composée d'un tableau indexé par adresses (de données ou d'instructions). Ce tableau contient les informations nécessaires au choix du placement de la donnée dans le cache. Dans [TRS+] principalement trois techniques différentes de caches multi-latéraux sont étudiées :

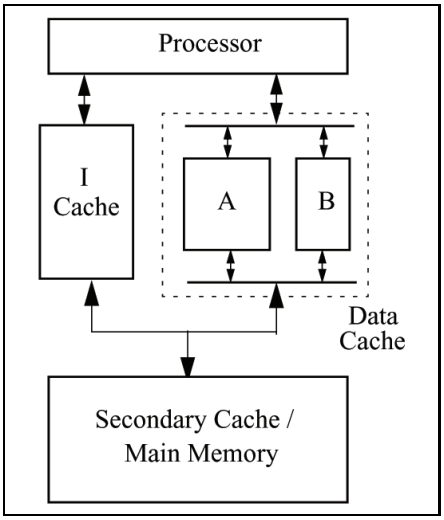

FIG. 2.3 – Schéma général d'une mémoire cache multi-latérale.

Non-temporal streaming (NTS) : tiré de [RD96] cette technique se base uniquement sur la localité temporelle. Une ligne de cache peut être marquée comme temporelle (T) ou non temporelle (NT) à l'intérieur de la DU. Les lignes NT sont mises dans un cache de petite taille et très associatif, alors que les lignes T sont mises dans le cache "principal". Durant sa présence en cache une ligne qui n'est pas réutilisée est marquée comme NT et sera mise dans le cache "secondaire" lors de sa prochaine apparition.

Memory address table (MAT) : cette approche est aussi basée sur la localité temporelle. Le fonctionnement et la structure du cache sont très proches du modèle NTS, à la différence qu'ici ce sont des macroblocs qui sont désignés comme "fréquemment" ou "non fréquemment" utilisés dans la DU. Un macrobloc est une zone mémoire continue (par exemple 1ko). Lorsqu'un miss intervient, on vérifie à quel macrobloc appartient la ligne de cache concernée. Ensuite, on compare la fréquence d'utili-

sation de ce macrobloc avec celle du macrobloc en conflit dans le cache principal (informations fournies par la DU). Si la fréquence d'utilisation du macrobloc "entrant" est supérieure à celle du macrobloc en conflit, la ligne présente en cache est alors remplacée. Sinon la ligne de cache ayant généré le miss est mise dans le cache secondaire.

Program counter selective (PCS) : contrairement aux deux modèles précédents, les décisions de placement des données en cache sont prises à partir du compteur ordinal plutôt qu'à partir de l'adresse effective de la donnée (indexation de la DU par l'adresse de l'instruction ayant généré le miss). Cependant, le principe utilisé est très proche du modèle NTS. C'est-à-dire qu'à une instruction d'accès mémoire on associe une valeur qui peut être T ou NT et qui est mise à jour de la même manière que le model NTS.

2.2 Les working sets

Un working set est un sous-ensemble des données nécessaires au fonctionnement d'un programme. Dans [WOT+95] une suite de programmes est caractérisée pour faciliter l'étude de machines multi-processeurs à mémoire partagée. Une partie de cette caractérisation se consacre aux working sets et à la localité temporelle. Une méthode est proposée pour déterminer la taille des working sets d'une application. La figure 3.2 montre le pourcentage de miss par rapport à la taille du cache pour quatre types de mémoires caches. Les working sets indiqués sont déterminés à l'aide des points d'inflexions de la courbe. Un point d'inflexion suivit d'une région assez plate, signifie qu'un working set se trouve entièrement en cache. On remarque sur la figure qu'il existe plusieurs working sets pour une même application, ce qui donne plusieurs paliers sur la courbe. Cela peut nous amener à une hiérarchisation des working sets comme le montre la figure 2.4. Il est à noter que les petits working sets exhibent une localité temporelle plus forte que ceux de taille supérieure.

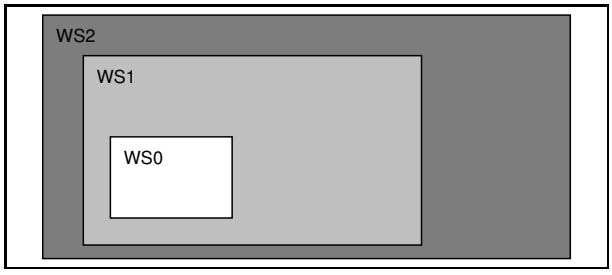

FIG. 2.4 – Hiérarchisation des working sets.

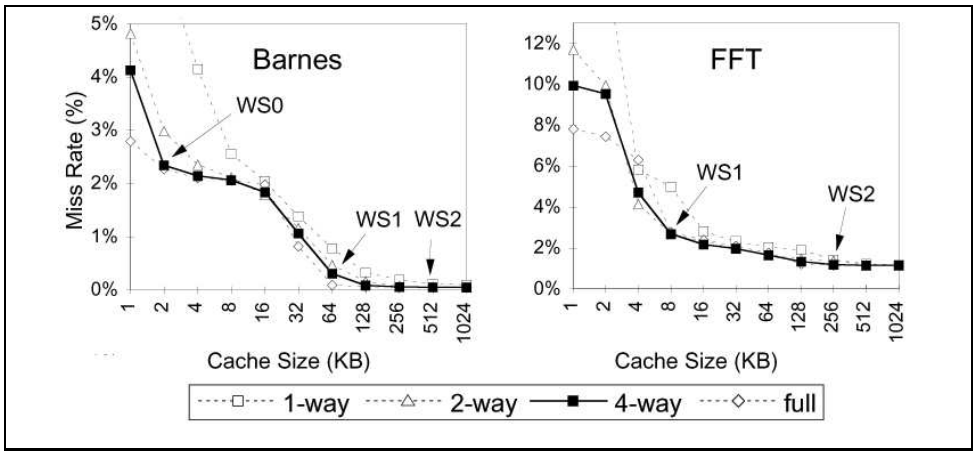

FIG. 2.5 – Evolution du taux de miss par rapport à la taille du cache pour deux applications parallèles.

2.3 Etude de la localité temporelle dans un programme

Cette étude tirée de [Pha95] cherche à modéliser les intervalles de temps entre chaque référence d'une donnée. Soit t_0, t_1, t_2, t_3 les instants de référence à une donnée. Le flot d'IRG (Inter-Reference Gap) pour cette donnée est : $t_1 - t_0$, $t_2 - t_1$, $t_3 - t_2$. Ce flot est modélisé à l'aide d'une chaîne de Markov d'ordre k. On se sert des valeurs des IRG passés pour mettre à jour de manière dynamique le modèle. A partir de ce modèle une technique de prédiction permet d'estimer la valeur de l'IRG suivant. Le but de cette technique est de pouvoir déterminer la donnée qui sera référencée le plus tard. Pour une mémoire cache par exemple, cela permet de choisir au mieux la ligne de cache à éjecter lors d'un miss.

Deuxième partie

Rapport

Chapitre 3

Quantification de la pollution dans le cache de niveau 2 sur un ensemble d'applications

La première partie du stage a consisté à étudier le comportement mémoire de logiciels afin de déterminer si une optimisation peut être possible. La suite de logiciels choisie pour réaliser cette étude est le SPEC CPU 2000. Cette suite est couramment utilisée en architecture pour caractériser les performances de processeurs et de compilateurs. D'abord, nous allons présenter les outils que nous avons utilisés pour réaliser cette étude (la suite d'applications et le simulateur). Ensuite nous allons observer l'existence des workings sets de cette suite, puis nous allons tenter de caractériser les données polluantes et de voir dans quelles proportions elles sont présentes dans la suite SPEC CPU 2000.

3.1 Méthodologie d'évaluation

Dans cette partie, nous allons voir les deux éléments qui nous ont servis à réaliser toutes les études durant le stage. Il s'agit de la suite logicielle SPEC CPU 2000 et du simulateur d'architecture super scalaire SimpleScalar.

3.1.1 Présentation des SPEC CPU 2000

SPEC (Standard Performance Evaluation Corporation) est une organisation qui propose plusieurs suites de logiciels pour évaluer les performances de différents systèmes (processeurs, serveurs,...). Dans notre cas, nous cherchons à évaluer les performances du cache de niveau 2 d'un processeur. Pour ce faire, nous allons utiliser les SPEC CPU 2000. Cette suite de logiciels se décompose en deux :

- CINT2000 : ensemble de douze programmes pour mesurer les performances en calcul entier du processeur.
- CFP2000 : ensemble de quatorze programmes pour mesurer les performances en calcul flottant du processeur.

Le tableau 3.1 donne une brève description de tous les programmes du SPEC CPU 2000. Les programmes présents dans le SPEC CPU 2000 sont assez variés, car ils doivent êtres représentatifs de l'usage général d'un processeur (compression, compilation, calcul scientifique...). Cette suite va nous permettre d'avoir une vision assez large des comportements mémoire d'applications.
En plus de ces 26 applications, SPEC fournis trois jeux de données pour chacune des applications : *test*, *train*, *ref*. La différence principale entre ces jeux de données est leur taille, ce qui influe directement sur le temps d'exécution de l'application (*test* étant le plus petit et *ref* le plus gros).

3.1.2 Présentation du simulateur SimpleScalar

Pour réaliser cette étude, nous avons utilisé le logiciel SimpleScalar v3.0. Il s'agit d'un simulateur de processeur superscalaire. Ce simulateur propose plusieurs modules : simulateur de prédicteur de branchement, simulateur de cache, simulateur out-of-order complet. Dans le cadre du stage, nous allons utiliser le simulateur de cache compilé pour interpréter les binaires Alpha. La figure 3.1 montre le fonctionnement du simulateur. Les applications données en entrées du simulateur sont les SPEC CPU 2000.

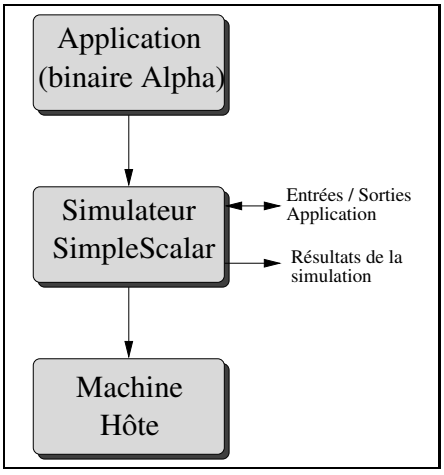

FIG. 3.1 – Principe du simulateur Simplescalar

Toutes les simulations présentées dans ce rapport ont été faites avec le jeu de données *train* des SPEC. Pour des raisons de temps de simulation, nous n'avons pas réalisé l'exécution complète des applications. Nous avons simulé

CINT2000		
Nom	Language	Description
164.gzip	C	Compression
175.vpr	C	Placement et routage de circuit FPGA
176.gcc	C	Compilateur de langage C
181.mcf	C	Optimisation combinatoire
186.crafty	C	Jeux d'echec
197.parser	C	Analyse syntaxique
252.eon	C++	Vision par ordinateur
253.perlbmk	C	Interpréteur pour langage PERL
254.gap	C	Interpréteur pour théorie des ensembles
255.vortex	C	Base de donnée orienté objet
256.bzip2	C	Compression
300.twolf	C	Simulateur de placement routage
CFP2000		
Nom	Language	Description
168.wupwise	Fortran 77	Physique chromodynamique quantique
171.swim	Fortran 77	Modélisation d'eau peu profonde
172.mgrid	Fortran 77	Multi-grid Solver : Champ de potentiel 3D
173.applu	Fortran 77	Equation différentiel partiellement parabolique / elliptique
177.mesa	C	Bibliothèque graphique 3D
178.galgel	Fortran 90	Calcul pour la dynamique des fluides
179.art	C	Reconnaissance d'image / Réseaux neuronaux
183.equake	C	Simulation de propagation d'onde sismique
187.facerec	Fortran 90	Traitement d'image : reconnaissance de visage
188.ammp	C	Chimie : outils de calculs
189.lucas	Fortran 90	Théorie des nombres / test de nombres premiers
191.fma3d	Fortran 90	Simulation de crash (éléments finis)
200.sixtrack	Fortran 77	Conception d'accélérateur nucléaire à haute énergie
301.apsi	Fortran 77	Météorologie

TAB. 3.1 – Liste des programmes de SPEC CPU 2000

1000000000 d'instructions en passant les 500000000 précédentes. Nous avons choisi cette solution car lorsque l'on simule complètement l'application, son comportement varie de manière négligeable. Le passage des 500000000 premières instructions sert à passer la phase d'initialisation de l'application, c'est-à-dire pour être directement dans le "régime permanent" de l'application.

3.2 Résultats préliminaires

Dans cette partie, nous allons d'abord vérifier que les SPEC CPU 2000 présentent bien le phénomène des working sets, puis nous allons voir dans quelles mesures les données polluantes sont présentes dans cette suite d'ap-

plications.

3.2.1 Etude des working sets

La méthode utilisée pour évaluer la taille des working sets des SPEC est celle présentée dans [WOT+95], c'est-à-dire que nous allons observer la variation du taux de miss d'une application en fonction de l'évolution de la taille de cache. Nous avons fait varier la taille du cache L2 entre 16ko et 64Mo. Nous avons réalisé des simulations avec des caches totalement associatif, 4 voies associatif et à correspondance directe ayant des ligne de 32 octets. Les résultats présentés dans la figure 3.2 ont été obtenus avec l'architecture mémoire suivante :

- Cache d'instructions L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache de données L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache L2 unifié (instructions + données) associativité et taille variable, taille de ligne de 32 octets, politique de remplacement LRU.

La figure 3.2 montre l'évolution du taux de miss de 6 applications du SPEC. Ces applications reflètent les comportements typiquement rencontrés. Sur cette figure, on constate principalement deux comportements :

- Soit la progression du taux de miss est logarithmique en fonction de la taille du cache (`197.parser`, `300.twolf`).
- Soit on observe bien la présence de points d'inflexions qui révèlent la présence de working sets (`171.swim`, `172.mgrid`, `177.mesa`, `179.art`).

Un autre point à constater est la faible différence que l'on observe entre les niveaux d'associativités. C'est pour rester dans des conditions réalistes que pour la suite des tests nous allons utiliser des caches 4 voies associatifs.

Maintenant, il reste à vérifier l'existence des données polluantes ainsi que leurs proportions.

3.2.2 Quantification des données polluantes

Une donnée est dite polluante pour le cache L2 si cette donnée est amenée dans ce cache à la suite d'un miss du L1, puis éjectée du L2 avant sa réutilisation par le cache L1. C'est-à-dire que cette donnée est entrée en cache L2 et n'a été utilisée qu'une seule fois par le cache L1. Il est possible que cette donnée ait éjecté du L2 une donnée plus fréquemment utilisée (localité temporelle plus forte). D'où le phénomène de pollution.
Avec cette définition, la quantité de données polluantes dépend directement de la structure de la hiérarchie mémoire. C'est-à-dire qu'en fonction de l'associativité, la taille de ligne, le nombre de lignes des différents caches, le nombre de données polluantes peut être plus ou moins important. Dans cette partie

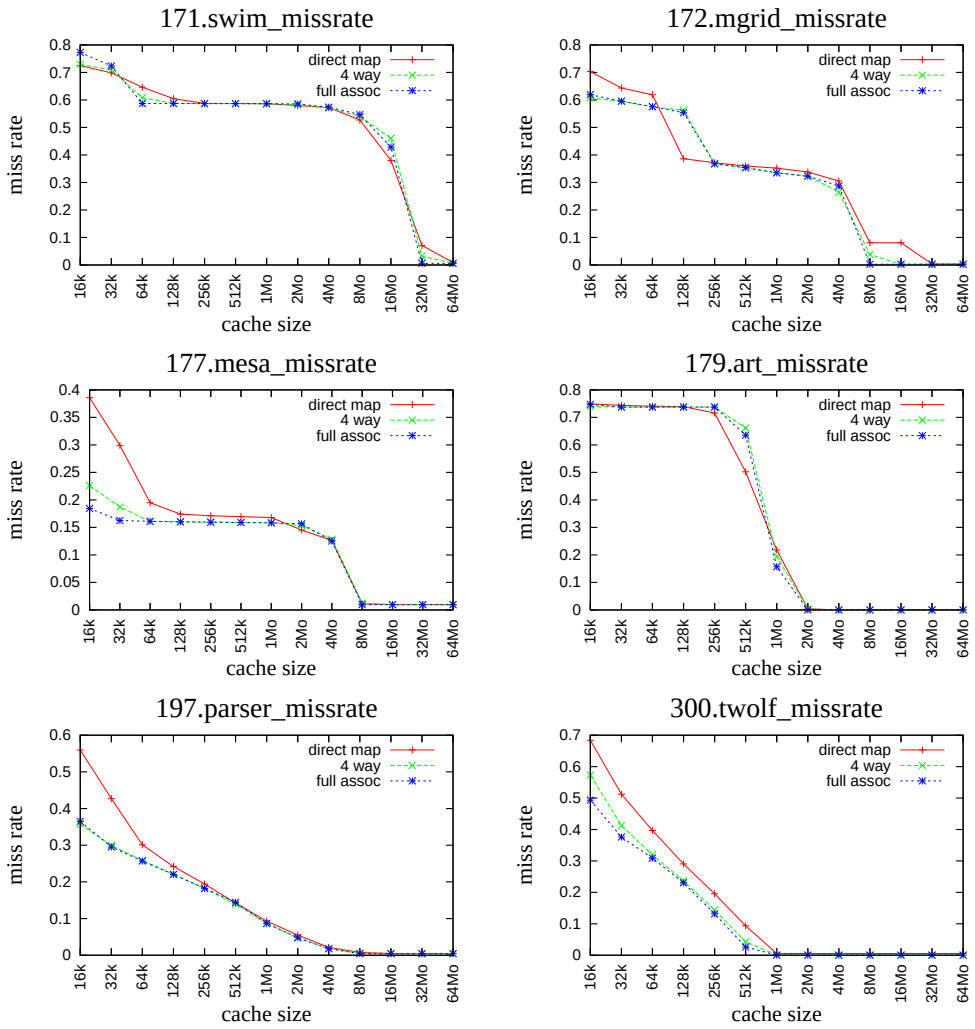

FIG. 3.2 – Evolution du taux de miss du cache L2 en fonction de la taille de cache

du rapport, nous avons surtout étudié l'influence de la structure du L2 sur les données polluantes. Mais il ne faut pas négliger le cache L1, car en fonction de sa structure il ne va pas réaliser les mêmes miss et de ce fait changer les accès réalisés vers le L2.

D'un point de vue pratique, il suffit d'ajouter une information *retouche* à chaque ligne du cache L2. Il s'agit d'un bit qui est positionné à 0 lorsqu'une ligne entre en cache et qui passe à 1 lorsqu'elle est accédée de nouveau par le cache L1 (hit L2). Ce mécanisme implique que la granularité de mesure des données polluantes est la ligne de cache (ou bloc).

Les résultats présentés dans la figure 3.4 ont été obtenus avec la hiérarchie mémoire suivante :

- Cache d'instructions L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache de données L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache L2 unifié (instructions + données) associativité et taille variable, taille de ligne de 32 octets, politique de remplacement LRU.

Nous avons fait varier la taille du cache L2 entre 16ko et 64Mo pour trois associativité différentes : correspondance directe, 4 voies associatif et totalement associatif.
Le ratio représenté correspond à :
$pratio = \frac{nbAccesPolluant}{nbAccesL2}$

On remarque que l'évolution de ce ratio en fonction de la taille du cache L2 correspond à celle du taux de miss pour les différentes configurations de cache. La notion de donnée polluante est bien dépendante de la structure du cache. Sur la figure 3.4 on observe ce comportement sur la variation de l'associativité et du nombre de lignes du cache L2.

On peut supposer que cette pollution engendre principalement des miss de capacité. La figure 3.3 met en rapport le ratio (*pratio*)de pollution d'un cache totalement associatif et le taux de miss de capacité de deux applications du SPEC CPU 2000.
Le taux de miss de capacité est calculé avec la méthode donnée dans [HS89]. Le taux de miss de capacité est égale au taux de miss d'un cache totalement associatif moins le taux de miss d'un cache infini (ou suffisamment grand pour ne pas réaliser de remplacement de ligne).
Sur cette figure, on peut constater que ces deux taux sont étroitement liés. On remarque aussi que la courbe du taux de miss capacité domine celle du ratio de pollution. Ce qui nous permet de confirmer que la pollution engendre une bonne partie des miss de capacité.

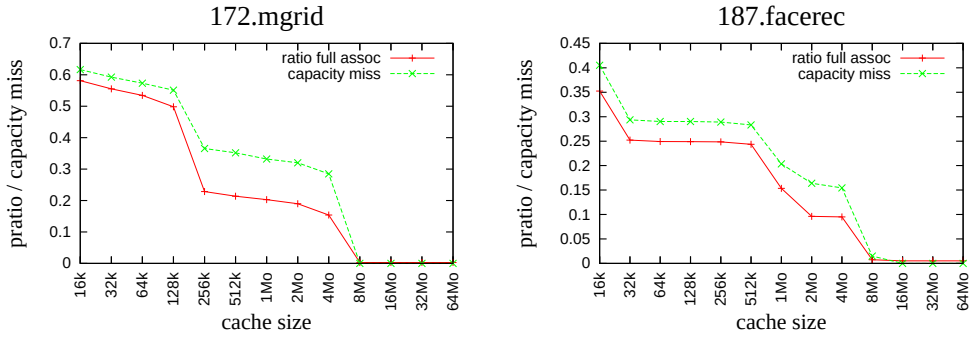

FIG. 3.3 – Comparaison entre l'évolution du ratio de pollution et l'évolution du taux de miss de capacité

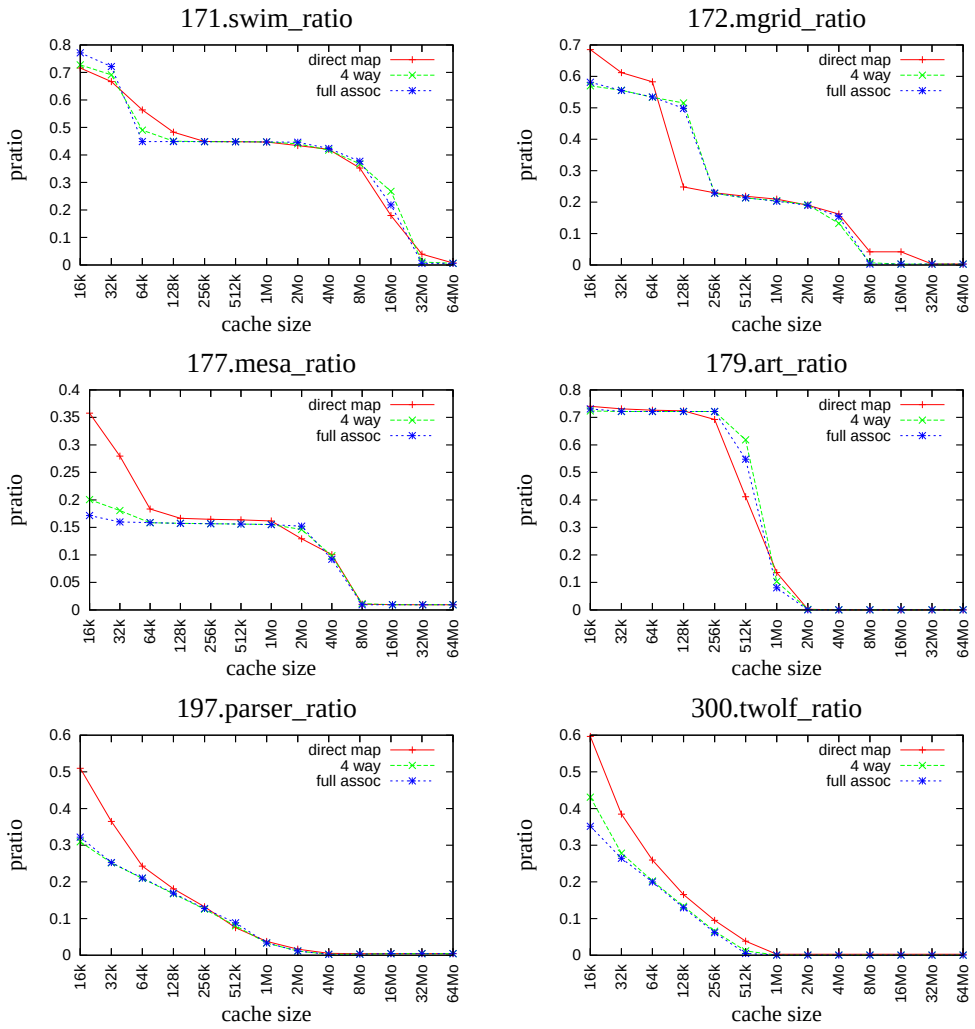

FIG. 3.4 – Evolution du ratio de pollution en fonction de la taille du cache L2 et de l'associativité

Chapitre 4

Caractérisation des données polluantes

Maintenant que nous avons montré que de nombreux blocs polluent le cache L2, nous allons essayer de déterminer au moment du chargement si la donnée accédée est polluante ou non et ensuite décider si cette donnée doit être mise en cache L2. La technique utilisée est issue des méthodes de prédictions que l'on retrouve par ailleurs en architecture (branchement, dépendance,...). Il existe plusieurs sources possibles de prédiction, l'adresse de l'instruction qui a généré l'accès mémoire, l'adresse effective de la donnée en mémoire, par exemple. Dans un premier temps, nous allons présenter le prédicteur utilisé. Ensuite nous allons voir les gains de performances (en termes de réduction du taux de miss) que peut apporter ce prédicteur et en même temps affiner la caractérisation des données polluantes.

4.1 Outil pour la caractérisation

Dans cette partie, nous allons voir de quel prédicteur nous sommes partis et dans quelles mesures nous l'avons modifié pour qu'il devienne un prédicteur de données polluantes.

Pour réaliser ce prédicteur de données polluantes nous sommes partis d'un prédicteur de branchment *skewed* tiré de [MSU97]. Le schéma 4.1 montre le principe général. Ce prédicteur est constitué de trois tables de compteurs à saturation (compteur 2 bits). Ces tables sont indexées par des fonctions *skewed*. On réalise un vote à la majorité entre les trois résultats donnés par les tables pour obtenir le verdict final. Nous avons choisis ce type de prédicteur car il a l'avantage d'être relativement insensible aux conflit de mapping.

Dans l'utilisation d'un prédicteur, il y a principalement deux phases : l'interrogation et la mise à jour. Lorsque le cache L2 réalise un miss, nous

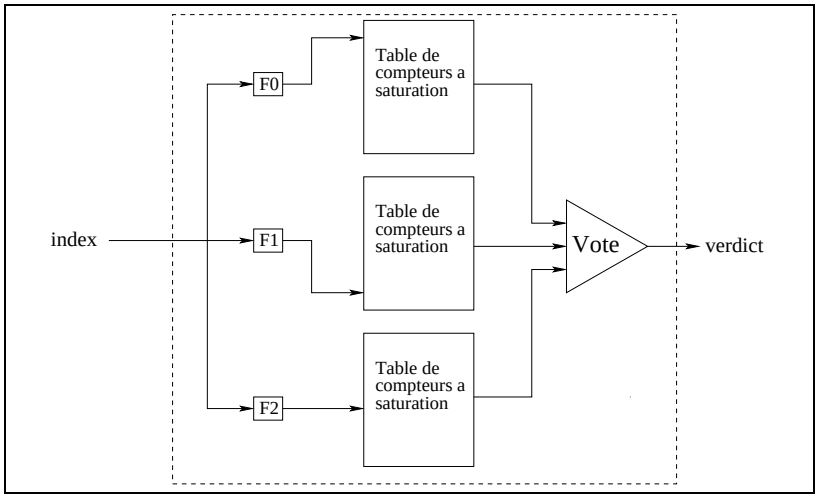

FIG. 4.1 – Schéma du prédicteur

voulons savoir si la donnée qui a généré ce miss est une donnée polluante (pour décider de sa mise en cache L2). C'est à ce moment où l'on interroge le prédicteur. Pour ce qui est de la mise à jour du prédicteur, elle doit être fait lorsqu'une donnée est éjectée du cache L2. On doit signaler au prédicteur si la donnée éjecté à été réutilisée par le cache L1. Pour ce faire, nous gardons le mécanisme de bit *retouche* mis en place dans 3.2.2. Cependant cette technique est incomplète.
Prenons un cas de figure où le cache L2 réalise un miss, on interroge le prédicteur, celui-ci fourni le verdict polluant, la donnée n'est pas mise en cache. Mais il se peut que ce verdict soit une mauvaise prédiction, c'est à dire que cette donnée n'est pas polluante. Il est possible que cette donnée ne soit jamais mise en cache car le prédicteur ne sera jamais mis à jour pour cette donnée et donc le prédicteur fournira toujours le verdict polluant. C'est pour cela qu'il faut ajouter un ensemble de tag de cache "fantôme" qui doit être maintenu avec une politique de gestion de cache classique (par exemple LRU). C'est lors d'un miss "fantôme" que nous devons mettre à jour le prédicteur.

A partir de là, les compteurs 2 bits ne vont plus représenter le biais global d'un branchement, mais la proportion qu'a une donnée à être polluante. La figure 4.2 montre la nouvelle signification du compteur.

Sur ce compteur, on peut influer sur 2 paramètres :

- L'etat du compteur pour la prédiction. C'est à dire que l'on peut donner le verdict de polluant à partir d'un seuil ou quand l'automate se trouve dans un état fort.
- Nombre de bits du compteur. C'est à dire passer d'un compteur à 4 états à un compteur à 8 états, 16 états, 32 états...

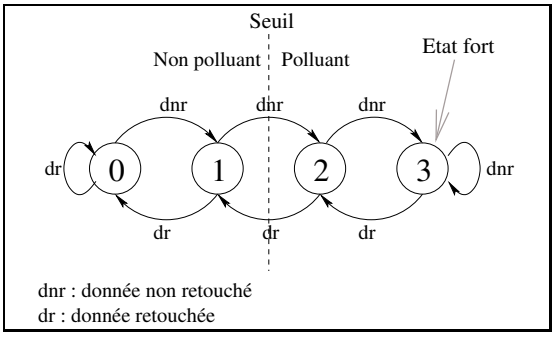

FIG. 4.2 – Schéma d'un compteur 2bits modifié pour la prédiction de données polluantes.

On peut déjà fixer le paramètre de l'état du compteur pour la prédiction. Comme pour tous les prédicteurs, la mauvaise prédiction a un coût. Ici il y a deux types de mauvaises prédictions, soit on prédit polluant alors que la donnée ne l'est pas, ou on prédit non polluant alors que la donnée l'est. Ces deux types de mauvaises prédictions n'ont pas le même coût. Dans le premier cas, il est fort probable que cette donnée soit rappelée peu de temps après cette mauvaise prédiction et cela va générer une fois de plus un miss jusqu'à la mémoire principale. Alors que dans le deuxième cas, on a comportement de cache normal (i.e. la donnée polluante entre en cache L2). Il est donc préférable de donner le verdict polluant seulement quand on est dans un état fort.

Au niveau de l'implémentation du prédicteur, nous avons choisis les fonctions *skewed* issues de [MSU97]. Plus d'informations sur ces fonctions peuvent être trouvées dans [BS97] (propriétés, implémentation matérielle). Ce sont ces trois fonctions qui vont nous servir à adresser chacune des trois tables à 2^n entrées du prédicteur. Ces fonctions nous permettent d'influer sur deux nouveaux paramètres :

- Le nombre d'entrées de chacune des tables de compteurs.
- Le choix de la méthode d'indexation du prédicteur : tag de cache, compteur ordinal (PC), index de page.

Pour conclure sur l'implémentation du prédicteur de données polluantes, voici un récapitulatif de son utilisation :

1. Accès au cache L2.
2. Observation des tags de cache "fantôme" :
   - Hit : passage à 1 du bit de retouche concerné.
   - Miss : Mise à jour du prédicteur (avec le bit de retouche), mise à jour du tag "fantôme"
3. Si occurrence d'un miss en L2, interrogation du prédicteur :

- Si le verdict est polluant : la donnée n'est pas mise en cache L2.
- Si le verdict est non polluant : on sélectionne la donnée à éjecter du cache L2 (LRU, random, ...), puis on met en cache L2 la donnée ayant générée le miss.

4. Envoi de la donnée en L1.

La figure 4.3 montre le principe du prédicteur. Sur ce schéma on voit que les tags fantomes sont mis à jour à chaque fois qu'un accès est réalisé vers le cache L2. Dans cet exemple le cache L2 réalise un miss sur le bloc *a*. On interroge alors le prédicteur qui fournit le verdict polluant, donc le bloc ne sera pas mise en cache L2. On obtient ce verdict car le bloc se situ dans le WS2 ce qui signifie qu'il exhibe une localité temporelle plus faible que ceux présents dans WS0 et WS1. On évite alors la pollution du cache L2 qui contient WS0 et WS1.

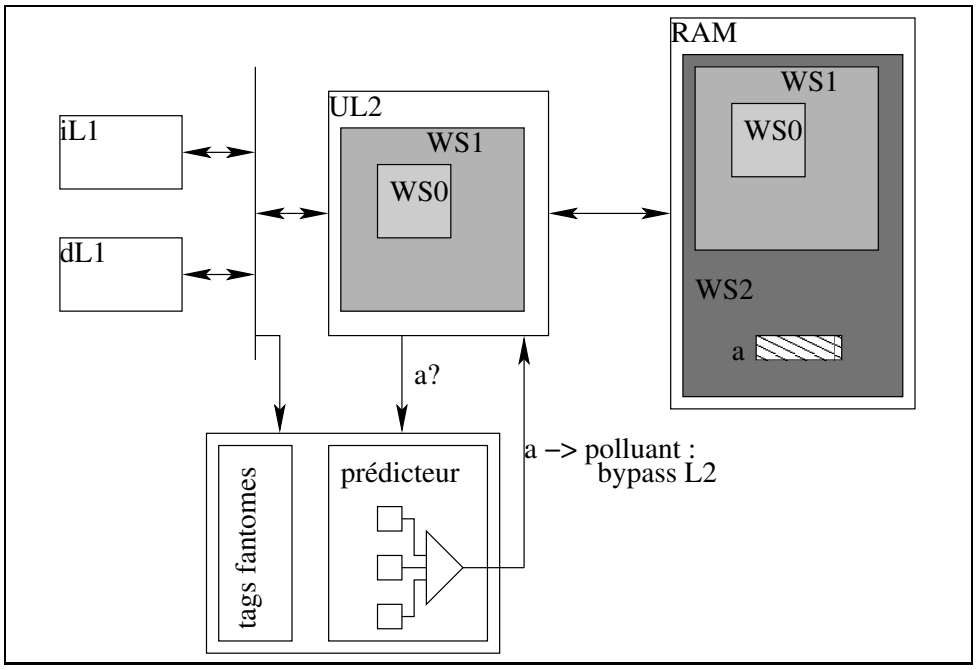

FIG. 4.3 – Schéma de principe du prédicteur.

Cette implémentation nous laisse trois paramètres que l'on peut faire varier :

- Le nombre de bits du compteur.
- Méthode d'indexation des tables de compteurs (tag de cache, index de page, compteur ordinal).
- Nombre d'entrées par table de compteurs. *Dans la suite du rapport lorsque nous indiquons un nombre d'entrées n, il s'agit du nombre d'entrées pour une table, soit un nombre total de $3 * n$ entrées pour*

le prédicteur.

4.2 Résultats

Dans cette partie, nous allons voir quelle est l'influence des différents paramètres du prédicteur. Nous allons pouvoir grâce à cela caractériser d'une manière un peu plus fine les données polluantes.

Le premier paramètre que nous allons étudier est le nombre de bits du compteur à saturation. Pour cette étude, nous avons utilisé l'architecture mémoire suivante :

- Cache d'instructions L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache de données L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache L2 unifié (instructions + données) 4 voies associatif de 128ko, taille de ligne de 32 octets, politique de remplacement LRU.

Pour le prédicteur, nous avons choisi la configuration suivante :

- Verdict sur état fort du compteur.
- Indexation des tables par le compteur ordinal (PC).
- 1024 entrées par tables.
- Nombre de bits par compteur variable, entre 2 bits et 7 bits.

172.mgrid

	2bits	3bits	4bits	5bits	6bits	7bits
nb bypass	31259038	30848890	30611687	30456821	30187112	29899046
miss rate	0.5721	0.5694	0.5679	0.5671	0.5654	0.5640

197.parser

	2bits	3bits	4bits	5bits	6bits	7bits
nb bypass	11482632	9772205	9311686	8354228	7617000	7028098
miss rate	0.2587	0.2446	0.2414	0.2330	0.2267	0.2219

301.apsi

	2bits	3bits	4bits	5bits	6bits	7bits
nb bypass	19814504	14937975	14004844	13777403	13546588	13097417
miss rate	0.2246	0.1999	0.1958	0.1949	0.1938	0.1917

TAB. 4.1 – Evolution du nombre de données non mises en cache L2 et du missrate en fonction du nombre de bits du compteur

Le tableau 4.1 montre l'évolution du nombre de données non mises en cache L2 et du missrate en fonction du nombre de bits du compteur. On peut remarquer qu'en même temps que le taux de miss diminue, le nombre de données considérées comme polluantes diminue, ce qui signifie que le prédicteur réalise de meilleures prédictions (meilleure sélection des données polluantes).

Ceci nous confirme bien l'existence des données polluantes, c'est-à-dire que le bypass de certaines données bien précise diminue encore le taux de miss. Il est possible qu'en affinant plus cette sélection on arrive à réduire le taux de miss.
Pour la suite des tests, nous allons prendre des compteurs à 4 bits avec verdict sur état fort. Un compteur de cette taille offre de bonnes performances ainsi qu'un coût matériel plus raisonnable qu'un compteur à 7 bits.

Maintenant, nous allons observer l'influence du nombre d'entrées par table du prédicteur. Pour cela, nous gardons la même architecture mémoire que précédemment. Pour le prédicteur, nous faisons maintenant varier le nombre de compteurs par table, mais on garde la même indexation (PC), les compteurs sont à 4 bits et donnent leur verdict sur l'état fort.
La figure 4.4 montre cette évolution pour deux SPEC. Nous avons fait varier le nombre d'entrées entre 512 et 1M (2^{20}). On remarque que l'influence de ce paramètre est quasiment nulle ou vraiment infime. Cette influence est à peu près identique pour tous les SPEC. Ceci nous suggère que seulement peu d'instructions génèrent de la pollution.

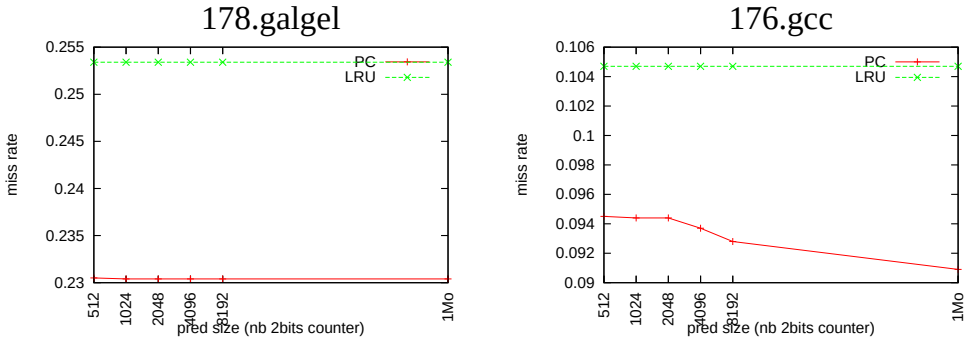

FIG. 4.4 – Evolution du taux de miss en fonction du nombre de compteurs par table

Le dernier paramètre important à étudier est la méthode d'indexation des tables. Nous avons le choix entre trois possibilités : indexation par le tag de cache, par l'adresse de l'instruction réalisant l'accès mémoire (PC) ou par l'index de la page où se situe la donnée. La figure 4.6 compare les performances de deux de ces techniques (PC et tag de cache) avec une gestion LRU du cache L2. Ces simulations ont été réalisées avec le predicteur suivant : tables de 1024 entrées, compteurs 4 bits avec verdict sur état fort. Avec la hiérarchie mémoire :

- Cache d'instructions L1 à correspondance directe de 8ko, lignes de 32 octets, politique de remplacement LRU.
- Cache de données L1 à correspondance directe de 8ko, lignes de 32

octets, politique de remplacement LRU.
- Cache L2 unifié (instructions + données) 4 voies associatif de taille variable, taille de ligne de 32 octets, politique de remplacement LRU.

On remarque que globalement l'indexation des tables par le compteur ordinal donne de meilleures performances que l'indexation faite avec le tag de cache. Une autre chose à remarquer est l'endroit où le gain de performance (réduction du taux de miss) est plus important. Cet endroit correspond au seuil du working set, c'est-à-dire le moment où le working set commence a tenir complètement en cache.
Si l'indexation des tables par index de pages n'est pas présentée ici, c'est parce que cette technique donne les mêmes résultats que l'indexation des tables par tags de cache.

La figure 4.5 met en rapport le taux de miss de capacité avec le gain de performance réalisé avec le prédicteur précédent indexé par le PC. Ce que l'on peut remarquer, c'est que le gain de performance le plus important se fait lorsque la chute de miss de capacité est importante. Ce qui signifie que le gain de performance est possible grâce au potentiel de miss de capacité disponible entre deux tailles de cache.

Maintenant que nous avons vu ce que peut apporter ce prédicteur du point de vue performance de la hiérarchie mémoire, on peut s'interroger sur ses performances, c'est-à-dire voir son taux de bonnes prédictions.

Dans la partie précédente nous nous sommes arrêtés sur le prédicteur suivant :
- Indexation des tables par le PC.
- 1024 entrées par table de prédiction.
- Compteur à 4 bits avec verdict sur état fort.

Ici, nous allons essayer d'évaluer la qualité de ses prédictions. D'un point de vue pratique, nous avons utilisé une gestion LRU du cache L2, lorsqu'une donnée est éjectée, nous effectuons une prédiction et nous comparons le résultat avec le bit de retouche. On se retrouve avec quatre cas de figures :
- Verdict polluant, retouche à 1 : mauvaise prédiction.
- Verdict polluant, retouche à 0 : bonne prédiction.
- Verdict non polluant, retouche à 1 : bonne prédiction.
- Verdict non polluant, retouche à 0 : mauvaise prédiction.

La figure 4.7 montre l'évolution du taux de mauvaise prédiction en fonction de la taille du cache. Ce que l'on peut remarquer c'est que globalement le prédicteur donne moins de 10% de mauvaises prédictions. Ce taux a tendance à diminuer avec l'augmentation de la taille de cache, ce qui est logique, car le prédicteur est moins souvent sollicité. Ces deux tendances sont présentes sur à peu près tout les SPEC CPU 2000, à quelques exceptions où le taux de mauvaise prédiction peut aller jusqu'à 30%.

La technique d'indexation du prédicteur par tag de page donne des ré-

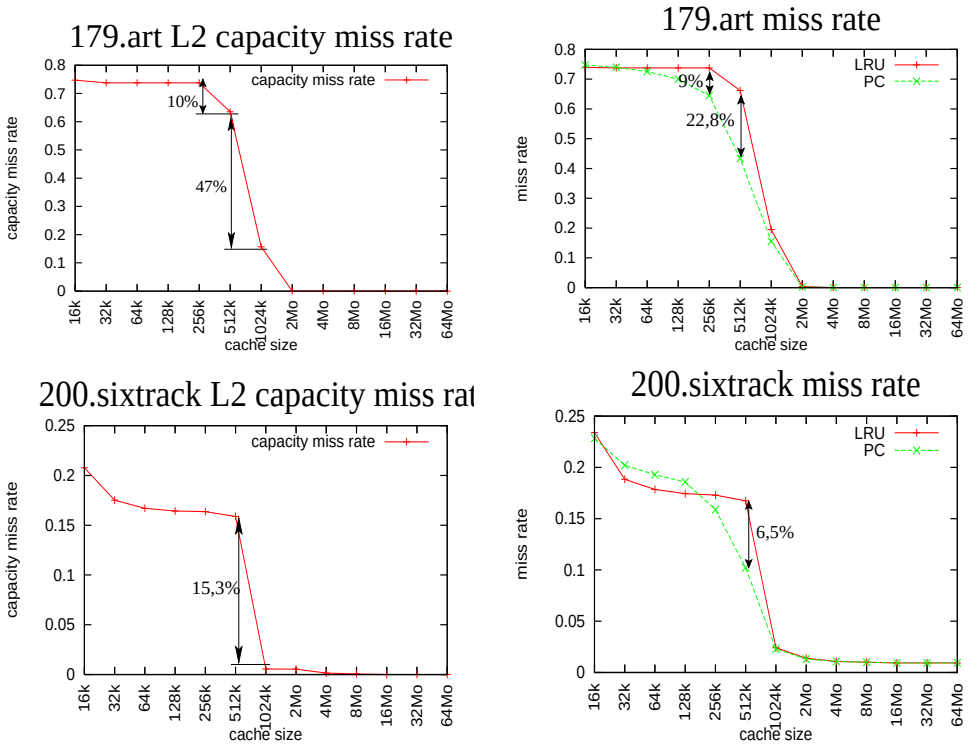

FIG. 4.5 – Comparaison entre l'évolution des miss de capacité et du gain de performance réalisé avec un prédicteur

sultats similaires à l'indexation du prédicteur par tag de cache au niveau du taux de miss du L2. Cependant, on peut s'interroger sur la qualité des prédictions au niveau d'une page. C'est-à-dire est-ce que sur une page un prédicteur indexé par tag de page donne de meilleurs résultats qu'un prédicteur indexé par le PC ? La figure 4.8 montre l'évolution des taux :

$$taux1 = \frac{nb(misspredpage > misspredPC)}{nbpages}$$

$$taux2 = \frac{nb(misspredPC > misspredpage)}{nbpages}$$

Où $nb(misspredpage > misspredPC)$ représente le nombre de pages où le prédicteur indexé par tag de page réalise plus de mauvaises prédictions que le prédicteur indexé par le PC. C'est-à-dire le nombre de pages où le prédicteur indexé par tag de page est le moins performant.
La figure 4.8 montre que sur une page les prédictions réalisées avec une indexation par PC donnent de meilleurs résultats. Par exemple, pour l'application `equake` un prédicteur indexé par tag de page donne de meilleurs résultats seulement dans 15% des cas. On va donc préférer garder une in-

dexation des tables par PC.

A la vue de ces résultats, on peut tirer quelques conclusions sur les données polluantes :

- Grâce aux études menées sur les différentes techniques d'indexations (PC, tag de cache, index de page), il semblerait qu'il existe une corrélation forte entre la donnée polluante et l'instruction qui réalise l'accès à cette donnée.
- La variation du nombre de bits d'un compteur à saturation nous a permis de confirmer l'existance des données polluantes.
- La variation du nombre d'entrées dans une table de compteurs nous a permis de voir que peu d'instruction génèrent des accès polluant.

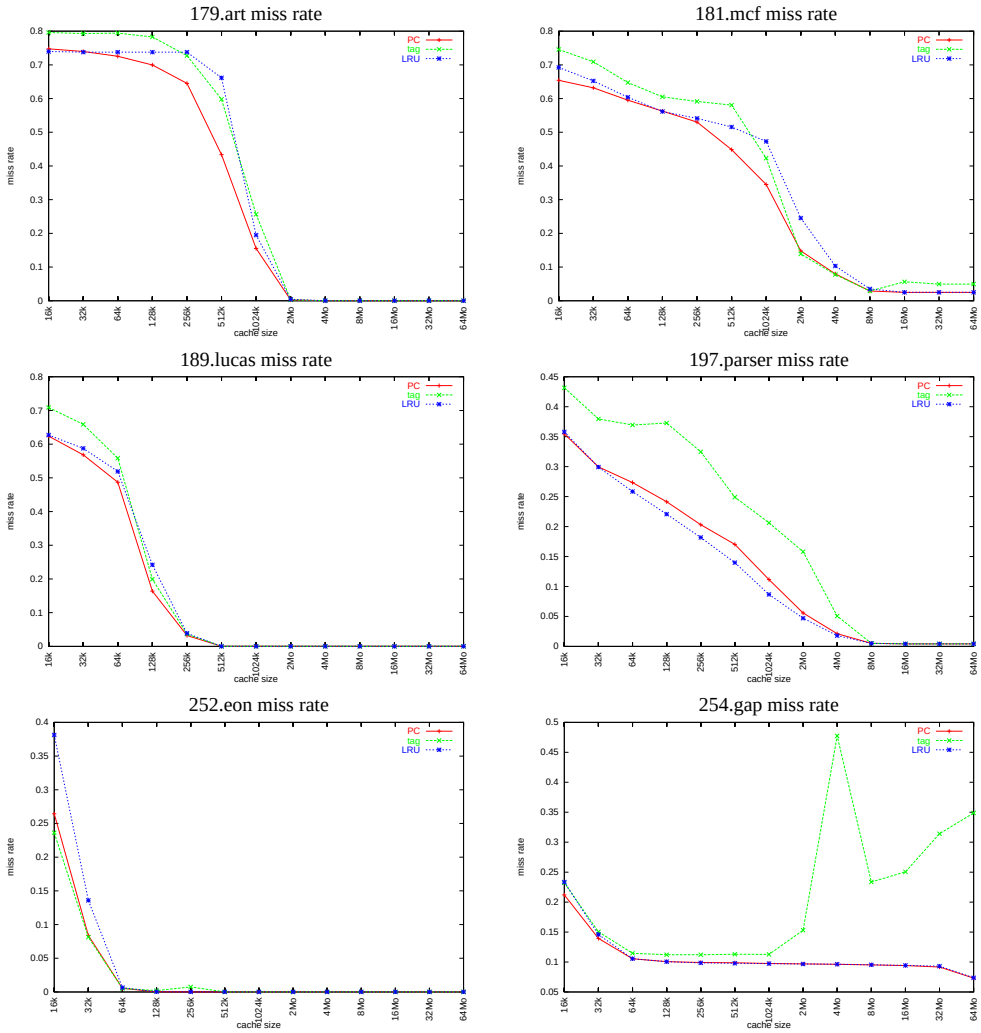

FIG. 4.6 – Evolution du taux de miss en fonction de la taille de cache pour trois politiques de gestion de cache

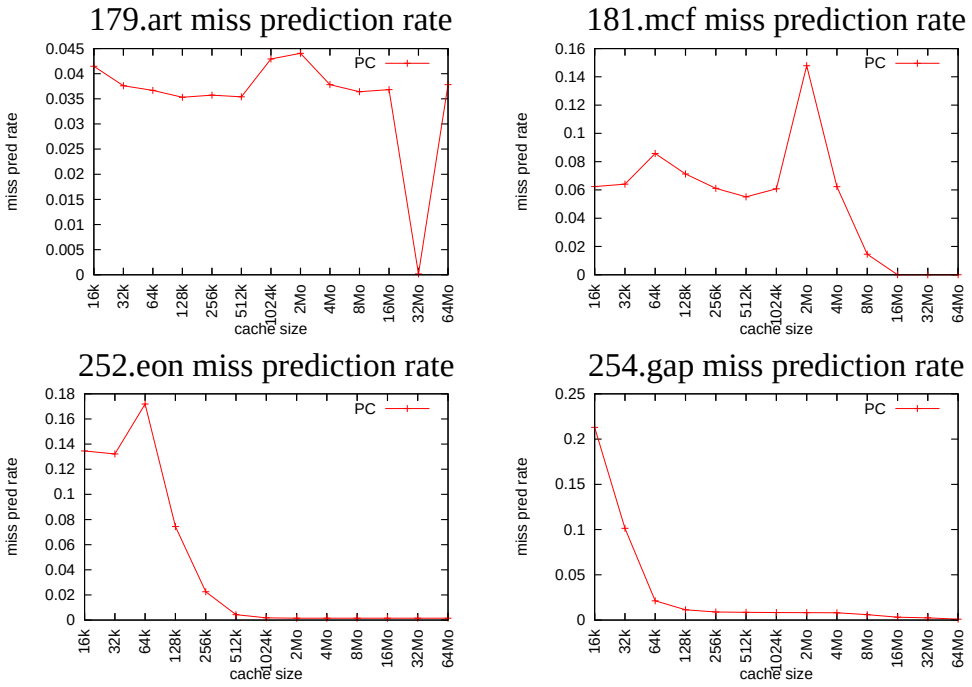

FIG. 4.7 – Evolution du taux de mauvaise prédiction en fonction de la taille de cache.

FIG. 4.8 – Evolution du taux de mauvaise prédiction par page en fonction de la taille de cache

Conclusion

Cette étude est un premier pas vers une gestion active du cache de niveau 2. Nous avons vu une nouvelle manière de gérer les remplacements dans les caches de niveau 2. Cette gestion est possible grâce à la mise en évidence de la notion de "donnée polluante".

Nous avons tout d'abord donné une première définition de cette notion. Cependant, elle nécessite d'être précisée. Nous avons principalement étudié ce phénomène de pollution dans une hiérarchie mémoire classique, il serait intéressant de caractériser ce phénomène dans sa globalité. Cela nous mène plusieurs questions, dans quelles mesures les "données polluantes" dépendent de la forme de la hiérarchie mémoire ? Existe-t-il des données polluantes indépendantes de la hiérarchie ?

Nous avons modifié un prédicteur utilisé pour les branchements afin qu'il puisse réaliser la prédiction de "données polluantes". Ce prédicteur nous a permis, en plus de fournir une nouvelle politique de remplacement, d'approfondir notre connaissance sur les "données polluantes". Nous avons vu que des gains de performances étaient possibles principalement lorsque l'on adresse les tables de prédictions avec le PC. Cela implique une forte corrélation entre l'instruction réalisant le premier accès à un bloc et les "données polluantes". Cette remarque peut nous amener à nous demander si une approche "compilateur" est possible ? C'est-à-dire réaliser la détection de ces instructions au moment de la compilation et ainsi forcer le bypass du cache L2 de manière statique.

Cette approche peut aussi apporter une amélioration aux techniques de "prefetch". En combinant ces deux techniques, on peut espérer ramener en cache des blocs utiles à temps et éviter une pollution de la hiérarchie mémoire. Par exemple lorsque l'on réalise un accès à un tableau de manière linéaire, on peut réaliser du "prefetch" au niveau du cache L1 sans faire entrer ce tableau en cache L2. On profite ainsi de la localité spatiale du tableau sans pour autant polluer le cache L2.

Bibliographie

[BS97] Francois Bodin and André Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, 46(5) :530–544, 1997.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992. Also available as U. Washington CS TR 92-06-03.

[CB94] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proc. of the 21st Symp. on Computer Architecture (21st ISCA'94), Computer Architecture News*, pages 223–232, Chicago, April 1994. ACM SIGARCH. Published as Proc. of the 21st Symp. on Computer Architecture (21st ISCA'94), Computer Architecture News, volume 22, number 2.

[FP92] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th MICRO*, pages 102–110, 1992.

[GAV95] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In ACM, editor, *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, CONFERENCE PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SUPERCOMPUTING 1995 ; 9th, pages 338–347, New York, NY 10036, USA, 1995. ACM Press.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[HS89] Mark D. Hill and Alan J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12) :1612–1629, December 1989.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth Inter-*

national Conference on Architectural Support for Programming Languages and Operating Systems, pages 62–73, October 1992.

[MSU97] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 292–303, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[Pha95] Bhaskarpillai Gopinath Vidyadhar Phalke. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 291–300, New York, NY, USA, May 1995. ACM Press.

[RD96] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proceedings of the 25th International Conference on Parallel Processing*, volume I, Architecture, pages I :154–163, Boca Raton, FL, August 1996. CRC Press. Michigan.

[Sez93] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178. ACM Press, 1993.

[TRS+] Edward S. Tam, Jude A. Rivers, Vijayalakshmi Srinivasan, Gary S. Tyson, and Edward S. Davidson. Active management of data caches by exploiting reuse information. In *IEEE Transactions on Computers*.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs : Characterization and methodological considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95), ACM SIGARCH Computer Architecture News*, pages 24–36, Santa Margherita, Italy, June 1995. Published as Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95), ACM SIGARCH Computer Architecture News, volume 23, number 6.